CFD General Notation System
CGIO User's Guide

Document Version 3.1.2

CGNS Version 3.1.3

Contents

1 Introduction

The CGIO interface provides low-level access to the database manager which underlies CGNS. The original database manager for CGNS was ADF (Advanced Data Format), and as such much of the CGIO interface routines are patterned after this. Starting with CGNS library version 2.xx, a new database manager HDF5 (Hierarchical Data Format) was introduced. At that time only one of these database managers could be used at a time, and this was selected at build time.

In CGNS library version 3.xx, the CGIO interface was developed to support both database managers simultaneously, and in a fashion transparent to the application code. This is now the preferred way to access the database manager.

This document defines the general structure of a database file, but not the specific implementation details. See the ADF and HDF5 Implementations for the details. The CGIO core routines used to store and retrieve data from the database manager are also described.

2 The CGIO Software Library

2.1 Node - The Building Block

A database is a hierarchical 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, may be connected to an arbitrary number of children, each of which is itself a node. In this system, a 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 (node locator)
- 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 database. This structure allows the construction of a hierarchical database as shown in Figure 1 on p. 3. As illustrated in the figure, it is possible to reference nodes in a second file (*File_Two*) from the original file (*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 database files start with a root node, which is created automatically when a new file is opened. There is only one root node in a database file, and may be referenced by the database Root ID or by name as “`/`”.

2.2 Node Attributes

Each node in the database 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 hierarchical database system.

Data	The data associated with a node.
Data Type	A 2-byte character field, blank filled, case sensitive. Specifies the type of data (e.g., real, integer, character) associated with this node. The supported data types are listed in Table 1 on p. 4.
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).

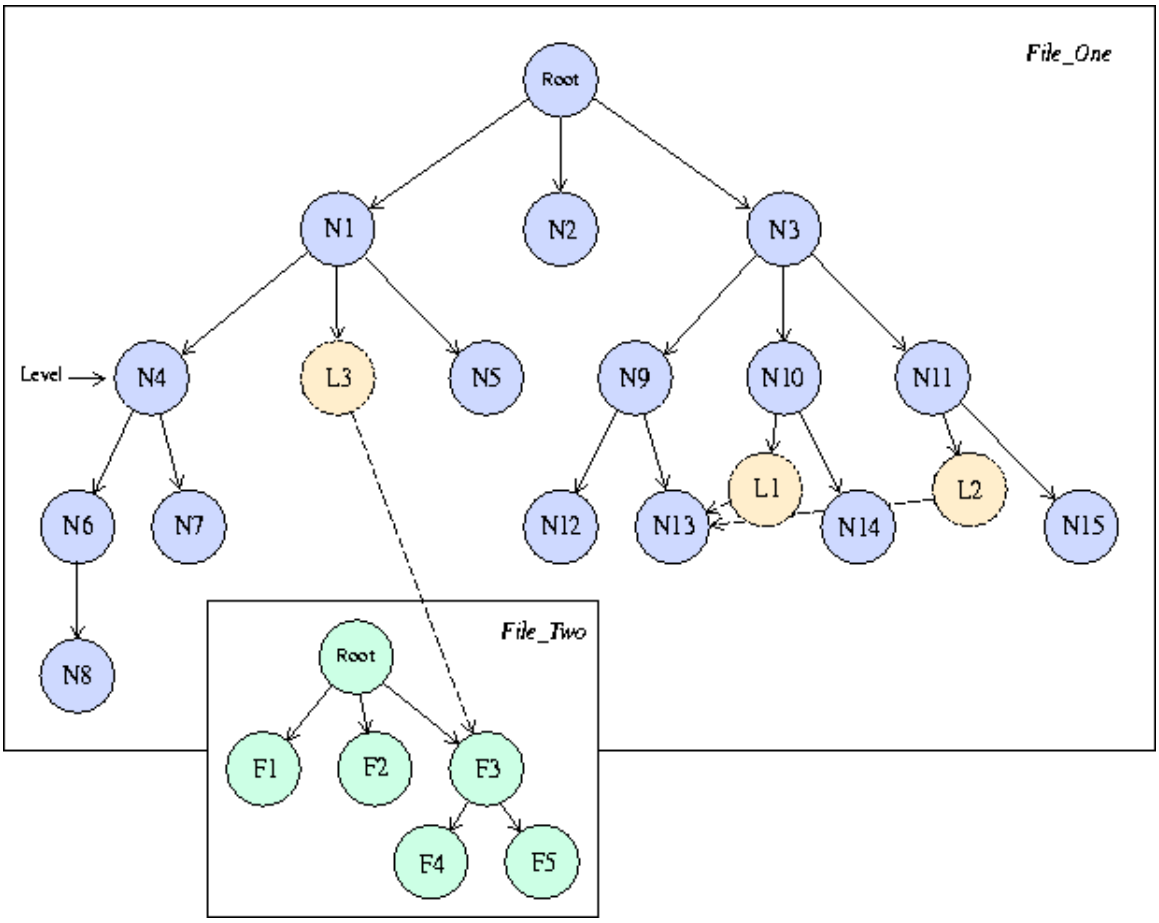

Figure 1: Example Database Hierarchy of Nodes

ID	A unique identifier to access a given node within a file. This field contains sufficient information for the database manager 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 may be different. Within different programs, the node ID for the same node may also 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

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, the databse manager checks the requested name against the other names of the child nodes of the specified parent. If the requested name is not unique, an error is returned.

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 from C must have the null "\0" character appended to them. Names returned 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.

2.3 Supported Data Types

Table 1: Data Types

Notation	Data Type	C Type	Fortran Type
`MT`	No Data		
`I4`	32-bit Integer	`int`	`integer*4`
`I8`	64-bit Integer	`cglong_t`	`integer*8`
`U4`	32-bit Unsigned Integer	`unsigned int`	`integer*4`
`U8`	64-bit Unsigned Integer	`cgulong_t`	`integer*8`
`R4`	32-bit Real	`float`	`real*4`
`R8`	64-bit Real	`double`	`real*8`
`C1`	Character	`char`	`character`

B1	Byte (unsigned byte)	unsigned char	character*1
LK	Link		

The MT node contains no data, and is typically used as a container for subnodes (children).

A link is denoted by LK, 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 database file. The link parallels a soft link in the UNIX operating system in that it does not guarantee that the referenced node exists. The database manager will "resolve" the link only when information is requested about the linked node or it's children.

2.4 Glossary of Terms

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 The representation of a hierarchy of nodes on disk files. By use of links, it may physically span multiple files.

File An database file, which a single root node and its underlying structure.

ID A unique identifier to access a given node within a database file. This field contains sufficient information for the database manager 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 may be different. Within different programs, the node-ID for the same node may also be different. The ID is never actually written into a file.

Link-Node A special type of node. Links are created using the cgio_create_link (Section 5.1.3) 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. 3, 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 a 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`"

2.5 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. *cgns_io.h* has a number of variables defined. An example declaration would be:

```
char name[CGIO_MAX_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 CGIO_MAX_NAME_LENGTH=32
CHARACTER*(CGIO_MAX_NAME_LENGTH) NAME
```

or include the Fortran header file *cgns_io_f.h* which defines these parameters.

ID — A unique identifier to access a given node within a database. 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 may be different. Within different programs, the node ID for the same node may also 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.

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 `cgio_read_data` (Section 7.1.1), `cgio_write_data` (Section 7.1.4), `cgio_children_names` (Section 4.1.6), and `cgio_children_ids` (Section 4.1.7).

The user should be aware of the differences in array indexing between Fortran and C. The subroutines `cgio_read_all_data` (Section 7.1.2) and `cgio_write_all_data` (Section 7.1.5) 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 `cgio_write_all_data` to store the array on disk. Then a Fortran program might use `cgio_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 `cgio_write_data` and `cgio_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 the database, there will not be any problems.

return codes

The CGIO routines return an integer code indicating whether they were successfull or not. On success, 0 (`CGIO_ERR_NONE`) is returned. A non-zero return indicates an error. Return codes < 0 indicate an error at the CGIO level; codes > 0 indicate an error in the database manager. See Section 8.2 for a list of error codes and mesages.

2.6 Limits and Sizes

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

Table 2: Default Values and Sizes

Define	Value	Attribute
`CGIO_MAX_DATATYPE_LENGTH`	2	Data type length
`CGIO_MAX_DIMENSIONS`	12	Maximum dimensions
`CGIO_MAX_NAME_LENGTH`	32	Name length
`CGIO_MAX_LABEL_LENGTH`	32	Label length
`CGIO_MAX_VERSION_LENGTH`	32	Version length
`CGIO_MAX_DATE_LENGTH`	32	Date length
`CGIO_MAX_ERROR_LENGTH`	80	Maximum length of error string
`CGIO_MAX_LINK_DEPTH`	100	Maximum link depth
`CGIO_MAX_FILE_LENGTH`	1024	File name length
`CGIO_MAX_LINK_LENGTH`	4096	Maximum link data size

3 Database-Level Routines

Functions	Modes
ier = cgio_is_supported(int file_type);	- - -
ier = cgio_check_file(const char *filename, *int *file_type*);	- - -
ier = cgio_open_file(const char *filename, int file_mode, int file_type, *int *cgio_num*);	r w m
ier = cgio_close_file(int cgio_num);	r w m
ier = cgio_get_file_type(int cgio_num, *int *file_type*);	r w m
ier = cgio_get_root_id(int cgio_num, *double *rootid*);	r w m
call cgio_is_supported_f(file_type, *ier*)	- - -
call cgio_check_file_f(filename, *file_type*, *ier*)	- - -
call cgio_open_file_f(filename, file_mode, file_type, *cgio_num*, *ier*)	r w m
call cgio_close_file_f(cgio_num, *ier*)	r w m
call cgio_get_file_type_f(cgio_num, *file_type*, *ier*)	r w m
call cgio_get_root_id_f(cgio_num, *rootid*, *ier*)	r w m

Input/***Output***

`file_type`	Type of database file. acceptable values are `CGIO_FILE_NONE`, `CGIO_FILE_ADF`, `CGIO_FILE_HDF5` and `CGIO_FILE_ADF2`.
`filename`	Name of the database file, including path name if necessary. There is no limit on the length of this character variable.
`file_mode`	Mode used for opening the file. The supported modes are `CGIO_MODE_READ`, `CGIO_MODE_WRITE`, and `CGIO_MODE_MODIFY`.
`cgio_num`	Indentifier for the open database file.
`rootid`	Ndeo identifier for the root node of the database.
`ier`	Error status.

3.1 Function Descriptions

3.1.1 `cgio_is_supported`

Determines if the database type given by `file_type` is supported by the library. Retuns 0 if supported, else `CGIO_ERR_FILE_TYPE` if not. `CGIO_FILE_ADF` is always supported; `CGIO_FILE_HDF5` is supported if the library was built with HDF5; and `CGIO_FILE_ADF2` is supported when built in 32-bit mode.

3.1.2 `cgio_check_file`

Checks the file `filename` to determine if it is a valid database. If so, returns 0 and the type of database in `file_type`, otherwise returns an error code and `file_type` will be set to `CGIO_FILE_NONE`.

3.1.3 cgio_open_file

Opens a database file of the specified type and mode. If successfull, returns 0, and the database identifier in `cgio_num`, otherwise returns an error code. The database identifier is used to access the database in subsequent function calls.

The mode in which the database is opened is given by `file_mode`, which may take the value `CGIO_MODE_READ`, `CGIO_MODE_WRITE`, or `CGIO_MODE_MODIFY`. New databases should be opened with `CGIO_MODE_WRITE`, while existing databases are opened with either `CGIO_MODE_READ` (for read-only access) or `CGIO_MODE_MODIFY` (for read/write access).

A specific database type may be specified by `file_type`, which may be one of `CGIO_FILE_NONE`, `CGIO_FILE_ADF`, `CGIO_FILE_HDF5`, or `CGIO_FILE_ADF2`. When opening a database in write mode, `CGIO_FILE_NONE` indicates that the default database type should be used, otherwise the specified database type will be opened. When opening in read or modify mode, `CGIO_FILE_NONE` indicates that any database type is acceptable, otherwise if the database type does not match that given by `file_type` an error will be retuned.

3.1.4 cgio_close_file

Closes the database given by `cgio_num`. Returns 0 for success, else an error code.

3.1.5 cgio_get_file_type

Gets the type of the database given by `cgio_num`. Returns 0 and the type in `file_type` if successfull, else an error code.

3.1.6 cgio_get_root_id

Gets the unique node identifier for the root node in the database given by `cgio_num`. Returns 0 and the identifier in `rootid` if successfull, else an error code.

4 Data Structure Management Routines

Functions	Modes
ier = cgio_create_node(int cgio_num, double pid, const char *name, double *id);	- w m
ier = cgio_new_node(int cgio_num, double pid, const char *name, const char *label, const char *data_type, int ndims, const cgsize_tt *dims, const void *data, double *id);	- w m
ier = cgio_delete_node(int cgio_num, double pid, double id);	- w m
ier = cgio_move_node(int cgio_num, double pid, double id, double new_pid);	- w m
ier = cgio_number_children(int cgio_num, double id, int *num_child);	r w m
ier = cgio_children_names(int cgio_num, double id, int start, int max_ret, int name_len, int *num_ret, char *child_names);	r w m
ier = cgio_children_ids(int cgio_num, double id, int start, int max_ret, int *num_ret, char *child_ids);	r w m
call cgio_create_node_f(cgio_num, pid, name, id, ier)	- w m
call cgio_new_node_f(cgio_num, pid, name, label, data_type, ndims, dims, data, id, ier);	- w m
call cgio_delete_node_f(cgio_num, pid, id, ier)	- w m
call cgio_move_node_f(cgio_num, pid, id, new_pid, ier)	- w m
call cgio_number_children_f(cgio_num, id, num_child, ier)	r w m
call cgio_children_names_f(cgio_num, id, start, max_ret, name_len, num_ret, child_names, ier)	r w m
call cgio_children_ids_f(cgio_num, id, start, max_ret, num_ret, child_ids, ier)	r w m

Input/*Output*

`cgio_num`	Database identifier.
`pid`	Parent node identifier.
`id`	Node identifier.
`name`	Node name (max length 32).
`label`	Node label (max length 32).
`data_type`	Type of data contained in the node. One of "MT", "I4", "I8", "U4", "U8", "R4", "C1", or "B1".
`ndims`	Number of dimensions for the data (max 12).
`dims`	Data dimension values (`ndims` values).
`data`	Data array to be stored with the node.
`new_pid`	New parent node identifier under which the node is to be moved.

`num_child`	Number of children of the specified node.
`start`	Starting index for returned child names or ids (1 <= `start` <= `num_child`).
`max_ret`	Maximum child names or ids to be returned (1 <= `max_ret` <= `num_child-start+1`).
`name_len`	Length reserved for each returned child name.
`num_ret`	Number of returned values of child names or identifiers.
`child_names`	Child node names (`num_ret` values). This array should be dimensioned at least (`name_len * max_ret`).
`child_ids`	Child node identifiers (`num_ret` values). This array should be dimensioned at least (`max_ret`).
`ier`	Error status.

4.1 Function Descriptions

4.1.1 `cgio_create_node`

Creates a new empty node in the database given by `cgio_num` as a child of the node identified by `pid`. The name of the new node is given by `name`, and must not already exist as a child of the parent node. The node will contain no label, dimensions, or data. Use the Node Management Routines (Section 6) to change the properties of the node, and the Data I/O Routines (Section 7) to add data. Returns 0 and the identifier of the new node in `id` on success, else an error code.

4.1.2 `cgio_new_node`

Creates a new node in the database given by `cgio_num` as a child of the node identified by `pid`. The name of the new node is given by `name`, and must not already exist as a child of the parent node. The node label is given by `label`, the type of data by `data_type`, the dimensions of the data by `ndims` and `dims`, and the data to write to the node by `data`. This is equivalent to calling the routines: `cgio_create_node`, `cgio_set_label`. `cgio_set_dimensions`, and `cgio_write_all_data`. Returns 0 and the identifier of the new node in `id` on success, else an error code.

4.1.3 `cgio_delete_node`

Deletes the node identified by `id` below the parent node identified by `pid` in the database given by `cgio_num`. All children of the deleted node will also be deleted unless a link is encountered. The link node will be deleted but nothing below it. Returns 0 on success, else an error code.

4.1.4 `cgio_move_node`

Moves the node indentified by `id` below the parent node identified by `pid` to below the new parent node identified by `new_pid` in the database given by `cgio_num`. A node by the same name as that that for `id` must not already exist under `new_pid`. A node may only be moved if it and the parent nodes all reside in the sane physical database. Returns 0 on success, else an error code.

4.1.5 cgio_number_children

Gets the number of children of the node identified by `id` in the database given by `cgio_num`, Returns 0 and the number of children in `num_child` on success, else an error code.

4.1.6 cgio_children_names

Gets the names of the children of the node identified by `id` in the database given by `cgio_num`. The starting index for the array of names is given by `start`, and the maximum number of names to return by `max_ret`. Both `start` and `max_ret` should be between 1 and `num_child`, inclusively. The size reserved for each name in `child_names` is given by `name_len`. The array `child_names` should be dimensioned at least (`name_len * max_ret`). Since node names are limited to a length of `CGIO_MAX_NAME_LENGHT` (32), `name_len` should be at least 32 to ensure the returned names are mot truncated. In C, an additional byte should be added to `name_len` allow for the terminating '0' for each name. If successfull, the function returns 0; the actual number of returned names is given by `num_ret`, and the array of names in `child_names`. In C, the names are '0'-terminated within each name field. In Fortran, any unused space is padded with blanks (space character).

4.1.7 cgio_children_ids

Gets the node identifiers of the children of the node identified by `id` in the database given by `cgio_num`. The starting index for the array of ids is given by `start`, and the maximum ids to return by `max_ret`. Both `start` and `max_ret` should be between 1 and `num_child`, inclusively. The array `child_ids` should be dimensioned at least (`max_ret`). If successfull, the function returns 0; the actual number of returned ids is given by `num_ret`, and the array of identifiers in `child_ids`.

5 Link Management Routines

Functions	Modes
ier = cgio_is_link(int cgio_num, double id, *int *link_len*);	r w m
ier = cgio_link_size(int cgio_num, double id, *int *file_len*, *int *name_len*);	r w m
ier = cgio_create_link(int cgio_num, double pid, const char *name, const char *filename, const char *name_in_file, *double *id*);	- w m
ier = cgio_get_link(int cgio_num, double id, *char *filename*, *char *name_in_file*);	r w m
call cgio_is_link_f(cgio_num, id, *link_len*, *ier*)	r w m
call cgio_link_size_f(cgio_num, id, *file_len*, *name_len*, *ier*)	r w m
call cgio_create_link_f(cgio_num, pid, name, filename, name_in_file, *id*, *ier*)	- w m
call cgio_get_link_f(cgio_num, id, *filename*, *name_in_file*, *ier*)	r w m

Input/*Output*

cgio_num	Indentifier for the open database file.
id	Node identifier.
pid	Parent node identifier.
link_len	Total length of the link information (file_len + name_len).
file_len	Length of the name of the linked-to file. This will be 0 if this is an internal link.
name_len	Length of the pathname of the linked-to node.
name	Name of the link node.
filename	Name of the linked-to file. If creating an internal link, then this should be NULL or an empty string. When reading an internal link, this will be returned as an empty string.
name_in_file	Pathname of the linked-to node.
ier	Error status.

5.1 Function Descriptions

5.1.1 cgio_is_link

Determines if the node indentified by id in the database given by cgio_num is a link or not. The function returns 0 if successfull, else an error code. If this node is a link, then the total length of the linked-to file and node information in returned in link_len. If the node is not a link, link_len will be 0.

5.1.2 cgio_link_size

Gets the size of the linked-to file name in `file_len` and the node pathname length in `name_len` for the node identified by `id` in the database given by `cgio_num`. The function returns 0 for success, else an error code. If this is an internal link (link to a node in the same database), then `file_len` will be returned as 0.

5.1.3 cgio_create_link

Creates a link node as a child of the parent node identified by `pid` in the database given by `cgio_num`. The name of the node is given by `name`, the name of the linked-to file by `filename`, and the pathname to the linked-to node by `name_in_file`. If this is an internal link (link to a node in the same database), then `filename` should be defined as `NULL` or an empty string. The function returns 0 and the indentifier of the new node in `id` on success, otherwise an error code is returned.

5.1.4 cgio_get_link

Gets the link information for the node identified by `id` in the database given by `cgio_num`. If successfull, the function returns 0 and the linked-to file name in `filename` and the node pathname in `name_in_file`. These strings are '0'-terminated, and thus should be dimensioned at least (`file_len + 1`) and (`name_len + 1`), respectively If this is an internal link (link to a node in the same database), then `filename` will be an empty string. The maximum length for a file name is given by `CGIO_MAX_FILE_LENGTH` (1024) and for a link pathname by `CGIO_MAX_LINK_LENGTH` (4096).

6 Node Management Routines

Functions	Modes
ier = cgio_get_node_id(int cgio_num, double pid, const char *pathname, double *id);	r w m
ier = cgio_get_name(int cgio_num, double id, char *name);	r w m
ier = cgio_set_name(int cgio_num, double pid, double id, const char *name);	- w m
ier = cgio_get_label(int cgio_num, double id, char *label);	r w m
ier = cgio_set_label(int cgio_num, double id, const char *label);	- w m
ier = cgio_get_data_type(int cgio_num, double id, char *data_type);	r w m
ier = cgio_get_dimensions(int cgio_num, double id, int *ndims, cgsize_t *dims);	r w m
ier = cgio_set_dimensions(int cgio_num, double id, const char *data_type, int ndims, const cgsize_t *dims);	- w m
call cgio_get_node_id_f(cgio_num, pid, name, *id*, *ier*)	r w m
call cgio_get_name_f(cgio_num, id, *name*, *ier*)	r w m
call cgio_set_name_f(cgio_num, pid, id, name, *ier*)	- w m
call cgio_get_label_f(cgio_num, id, *label*, *ier*)	r w m
call cgio_set_label_f(cgio_num, id, label, *ier*)	- w m
call cgio_get_data_type_f(cgio_num, id, *data_type*, *ier*)	r w m
call cgio_get_dimensions_f(cgio_num, id, *ndims*, *dims*, *ier*)	r w m
call cgio_set_dimensions_f(cgio_num, id, data_type, ndims, dims, *ier*)	- w m

Input/***Output***

`cgio_num`	Database identifier.
`pid`	Parent node identifier.
`id`	Node identifier.
`pathname`	Absolute or relative path name for a node.
`name`	Node name (max length 32).
`label`	Node label (max length 32).
`data_type`	Type of data contained in the node. One of “MT”, “I4”, “I8”, “U4”, “U8”, “R4”, “C1”, or “B1”.
`ndims`	Number of dimensions for the data (max 12).
`dims`	Data dimension values (`ndims` values).
`ier`	Error status.

6.1 Function Descriptions

6.1.1 cgio_get_node_id

Gets the node identifier for the node specified by `pathname` in the database given by `cgio_num`. if `pathname` starts with "/", then it is taken as an absolute path and is located based on the root id of the database, otherwise it is taken to be a relative path from the parent node identifed by `pid`. The function returns 0 and the node identifier in `id` on success, else an error code.

6.1.2 cgio_get_name

Gets the name of the node identified by `id` in the database given by `cgio_num`. The name is returned in `name`, and has a maximum length of `CGIO_MAX_NAME_LENGTH` (32). In C, `name` should be dimensioned at least 33 to allow for the terminating '0'. The function returns 0 for success, else an error code.

6.1.3 cgio_set_name

Sets (renames) the node identied by `id` in the database given by `cgio_num` to `name`. The parent node identifier is given by `pid`. There must not already exist a child node of `pid` with that name. The function return 0 on success, else an error code.

6.1.4 cgio_get_label

Gets the label of the node identified by `id` in the database given by `cgio_num`. The label is returned in `label`, and has a maximum length of `CGIO_MAX_LABEL_LENGTH` (32). In C, `label` should be dimensioned at least 33 to allow for the terminating '0'. The function returns 0 for success, else an error code.

6.1.5 cgio_set_label

Sets the label of the node identied by `id` in the database given by `cgio_num` to `label`. The function return 0 on success, else an error code.

6.1.6 cgio_get_data_type

Gets the data type of the data associated with the node identified by `id` in the database given by `cgio_num`. The data type is returned in `data_type`, and has a maximum length of `CGIO_MAX_DATATYPE_LENGTH` (2). In C, `data_type` should be dimensioned at least 3 to allow for the terminating '0'. The function returns 0 for success, else an error code.

6.1.7 cgio_get_dimensions

Gets the dimensions of the data associated with the node identified by `id` in the database given by `cgio_num`. The number of dimensions is returned in `ndims` and the dimension values in `dims`. Since the maximum number of dimensions is `CGIO_MAX_DIMENSIONS` (12), `dims` should be dimensioned 12,

unless the actual number of dimensions is already known. The function returns 0 for success, else an error code.

6.1.8 `cgio_set_dimensions`

Sets the data type and dimensions for data associated with the node identified by `id` in the database given by `cgio_num`. The data type (`data_type`) as one of:

- "`MT`" An empty node containing no data
- "`I4`" 32-bit integer (int or integer*4)
- "`I8`" 64-bit integer (cglong_t or integer*8)
- "`U4`" 32-bit unsigned integer (unsigned int or integer*4)
- "`U8`" 64-bit unsigned integer (cgulong_t or integer*8)
- "`R4`" 32-bit real (float or real*4)
- "`R8`" 64-bit real (double or real*8)
- "`C1`" character (char or character)
- "`B1`" unsigned bytes (unsigned char or character*1)

The number of dimensions is given by `ndims` (maximum is 12), and the dimension values by `dims`. Note that any existing data for the node will be destroyed. To add the data to the node, use one of the data writing routines (Section 7). The function returns 0 for success, else an error code.

7 Data I/O Routines

Functions	Modes
ier = cgio_read_data(int cgio_num, double id, const cgsize_t *s_start, const cgsize_t *s_end, const cgsize_t *s_stride, int m_num_dims, const cgsize_t *m_dims, const cgsize_t *m_start, const cgsize_t *m_end, const cgsize_t *m_stride, *void *data*);	r w m
ier = cgio_read_all_data(int cgio_num, double id, *void *data*);	r w m
ier = cgio_read_block_data(int cgio_num, double id, cgsize_t b_start, cgsize_t b_end, *void *data*);	r w m
ier = cgio_write_data(int cgio_num, double id, const cgsize_t *s_start, const cgsize_t *s_end, const cgsize_t *s_stride, int m_num_dims, const cgsize_t *m_dims, const cgsize_t *m_start, const cgsize_t *m_end, const cgsize_t *m_stride, void *data);	- w m
ier = cgio_write_all_data(int cgio_num, double id, void *data);	- w m
ier = cgio_write_block_data(int cgio_num, double id, cgsize_t b_start, cgsize_t b_end, void *data);	- w m
call cgio_read_data_f(cgio_num, id, s_start, s_end, s_stride, m_num_dims, m_dims, m_start, m_end, m_stride, *data*, *ier*)	r w m
call cgio_read_all_data_f(cgio_num, id, *data*, *ier*)	r w m
call cgio_read_block_data_f(cgio_num, id, b_start, b_end, *data*, *ier*)	r w m
call cgio_write_data_f(cgio_num, id, s_start, s_end, s_stride, m_num_dims, m_dims, m_start, m_end, m_stride, data, *ier*)	- w m
call cgio_write_all_data_f(cgio_num, id, data, *ier*)	- w m
call cgio_write_block_data_f(cgio_num, id, b_start, b_end, data, *ier*)	- w m

Input/***Output***

cgio_num	Database identifier.
id	Node identifier.
s_start	Starting indices for data in the database. Fortran indexing is used (starting at 1).
s_end	Ending indices for data in the database. Fortran indexing is used (starting at 1).
s_stride	Step increment for data in the database.
m_num_dims	Number of dimensions for data in memory.
m_dims	Dimension values for data in memory.
m_start	Starting indices for data in memory. Fortran indexing is used (starting at 1).
m_end	Ending indices for data in memory. Fortran indexing is used (starting at 1).
m_stride	Step increment for data in memory.

`data`	Array of data to be read or written.
`b_start`	Starting offset (index) for the data in the database. Fortran indexing is used (starting at 1).
`b_end`	Ending offset (index) for the data in the database. Fortran indexing is used (starting at 1).
`ier`	Error status.

7.1 Function Descriptions

7.1.1 `cgio_read_data`

This routine provides general purpose read capabilities from the node identified by `id` in the database given by `cgio_num`. It allows for a general specification of the starting location within the data as 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 vectors (`s_start`, `s_end` and `s_stride`) are used to describe the mapping of the data within the node, and a second set of vectors (`m_start`, `m_end` and `m_stride`) are used to describe the mapping of the desired data within memory.

The memory dimensions are given by `m_num_dims` and `m_dims`. There is no requirement that the node dimensions and memory dimensions match, only that the total number of values to be read are the same for the node and memory specifications.

The data are stored in both memory and on disk in "Fortran ordering." That is, the first index varies the fastest, and indexing starts at 1. Negative indexing is not allowed.

Be careful when writing data using `cgio_write_all_data` and then using `cgio_read_data` to randomly access the data. `cgio_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. `cgio_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 `cgio_write_all_data`. Then another program might use `cgio_read_data` to access a subsection of the data, and the routine would not return what was expected.

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

The function returns 0 on success, else an error code.

7.1.2 `cgio_read_all_data`

Reads all the data from the node identified by `id` in the database given by `cgio_num`. On success, the function returns 0 and the data in `data`, else an error code is returned. *Note*: Data is returned in Fortran indexing order.

7.1.3 cgio_read_block_data

Reads a contiguous block of data from the node identified by `id` in the database given by `cgio_num`. On success, the function returns 0 and the data in `data`, else an error code is returned. The starting index is given by `b_start` and the end by `b_end`. *Note*: Fortran indexing order for multi-dimensional data is used when computing the starting and ending locations.

7.1.4 cgio_write_data

This function is similar to `cgio_read_data`, but writes the data from memory to the node.

7.1.5 cgio_write_all_data

This function is similar to `cgio_read_all_data`, but writes the data from memory to the node.

7.1.6 cgio_write_block_data

This function is similar to `cgio_read_block_data`, but writes the data from memory to the node.

8 Error Handling Routines

Functions	Modes
ier = cgio_error_message(*char *error_msg*);	- - -
void cgio_error_code(*int *errcode*, *int *file_type*);	- - -
void cgio_error_exit(const char *msg);	- - -
void cgio_error_abort(int abort_flag);	- - -
call cgio_error_message_f(*error_msg*, *ier*)	- - -
call cgio_error_code_f(*errcode*, *file_type*)	- - -
call cgio_error_exit_f(msg)	- - -
call cgio_error_abort_f(abort_flag)	- - -

Input/***Output***

`error_msg`	Error message from CGIO or the underlying database manager.
`errcode`	The last error code from CGIO or the underlying database manager.
`file_type`	Where the last error was encountered. `CGIO_FILE_NONE` for an error coming from CGIO, else the type of database.
`msg`	An additional message to print, which prefixes the error message before exiting. This may be `NULL` or an empty string, in which case it is not printed.
`abort_flag`	Abort on error flag.
`ier`	Error status.

8.1 Function Descriptions

8.1.1 cgio_error_message

Gets the error message for the last error encountered, and returns it in `error_msg`, Maximum length of the error message is `CGIO_MAX_ERROR_LENGTH` (80). In C, `error_msg` should be dimensioned at least 81 in the calling routine to allow for the terminating '0'. The function returns the error code corresponding to the error message.

8.1.2 cgio_error_code

Returns the last error code in `errcode` and where is was generated in `file_type`. If the error code is < 0, then the error is from the CGIO library, and `file_type` will be `CGIO_FILE_NONE`, otherwise `file_type` will be the type of database.

8.1.3 cgio_error_exit

Prints `msg` and any error message to *stderr* and exits. The exit code will be `abort_flag` if it is set, else -1. If `msg` is `NULL` or an empty string, then it is not printed.

8.1.4 `cgio_error_abort`

Sets the flag to abort (exit) when an error is encountered. If `abort_flag` is non-zero, then an error in the CGIO routines or database managers will cause `cgio_error_exit` to be called. The exceptions are `cgio_is_supported` (Section 3.1.1), `cgio_check_file` (Section 3.1.2), and `cgio_is_link` (Section 5.1.1). These routines will not cause an abort on an error.

8.2 Error Messages

Table 3: CGIO Errors

Code	Error Message
0	no error
-1	invalid cgio index
-2	malloc/realloc failed
-3	unknown file open mode
-4	invalid file type
-5	filename is NULL or empty
-6	character string is too small
-7	file was not found
-8	pathname is NULL or empty
-9	no match for pathname
-10	error opening file for reading
-11	file opened in read-only mode
-12	NULL or empty string
-13	invalid configure option
-14	rename of tempfile file failed
-15	too many open files
-16	dimensions exceed that for a 32-bit integer

Table 4: ADF/HDF5 Errors

Code	Error Message
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

Continued on next page

Table 4: ADF/HDF5 Errors (*Continued*)

Code	Error Message
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
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
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
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
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

Continued on next page

Table 4: ADF/HDF5 Errors (*Continued*)

Code	Error Message
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
62	The node ID pointer is NULL
63	The maximum size for a file exceeded
64	Dimensions exceed that for a 32-bit integer
70	H5Glink:soft link creation failed
71	Node attribute doesn't exist
72	H5Aopen:open of node attribute failed
73	H5Iget_name:failed to get node path from ID
74	H5Gmove:moving a node group failed
75	H5Gunlink:node group deletion failed
76	H5Gopen:open of a node group failed
77	H5Dget_space:couldn't get node dataspace
78	H5Dopen:open of the node data failed
79	H5Dextend:couldn't extend the node dataspace
80	H5Dcreate:node data creation failed
81	H5Screate_simple:dataspace creation failed
82	H5Acreate:node attribute creation failed
83	H5Gcreate:node group creation failed
84	H5Dwrite:write to node data failed
85	H5Dread:read of node data failed
86	H5Awrite:write to node attribute failed
87	H5Aread:read of node attribute failed
88	H5Fmount:file mount failed
89	Can't move a linked-to node
90	Can't change the data for a linked-to node

Continued on next page

Table 4: ADF/HDF5 Errors (*Continued*)

Code	Error Message
91	Parent of node is a link
92	Can't delete a linked-to node
93	File does not exist or is not a HDF5 file
94	unlink (delete) of file failed
95	couldn't get file index from node ID
96	H5Tcopy:copy of existing datatype failed
97	H5Aget_type:couldn't get attribute datatype
98	H5Tset_size:couldn't set datatype size
99	routine not implemented
100	H5L: Link target is not an HDF5 external link
101	HDF5: No external link feature available
102	HDF5: Internal problem with objinfo
103	HDF5: No value for external link
104	HDF5: Cannot unpack external link
106	HDF5: Root descriptor is NULL
107	dimensions need transposed - open in modify mode
108	invalid configuration option

9 Miscellaneous Routines

Functions	Modes
ier = cgio_flush_to_disk(int cgio_num);	- w m
ier = cgio_library_version(int cgio_num, *char *version*);	r w m
ier = cgio_file_version(int cgio_num, *char *file_version*, *char *creation_date*, *char *modified_date*);	r w m
call cgio_flush_to_disk_f(cgio_num, *ier*)	- w m
call cgio_library_version_f(cgio_num, *version*, *ier*)	r w m
call cgio_file_version_f(cgio_num, *file_version*, *creation_date*, *modified_date*, *ier*)	r w m

Input/***Output***

cgio_num	Database identifier.
version	32-byte character string containing the database library version.
file_version	32-byte character string containing the database file version.
creation_date	32-byte character string containing the database file creation date.
modified_date	32-byte character string containing the last modification date for the database file.
ier	Error status.

9.1 Function Descriptions

9.1.1 cgio_flush_to_disk

Forces any buffered data in the database manager to be written to disk. Returns 0 if successfull, else an error code.

9.1.2 cgio_library_version

Gets the current library version for the database given by cgio_num. The version is returned in version which is of maximum length CGIO_MAX_VERSION_LENGTH (32). In C, version should be dimensioned at least 33 in the calling routine to allow for the terminating '0'. The function returns 0 if successfull, else an error code.

9.1.3 cgio_file_version

Gets the version, creation and last modified dates, for the database file given by cgio_num. The version is returned in file_version, which is of maximum length CGIO_MAX_VERSION_LENGTH (32). The creation date is returned in creation_date, and the last modified date in modified_date, which are of maximum length CGIO_MAX_DATE_LENGTH (32). In C, these should be dimensioned at least 33 in the calling routine to allow for the terminating '0'. The function returns 0 if successfull, else an error code.

10 Examples

The following examples build the database file shown in the example database figure Figure 1.

10.1 Fortran Example

```
      PROGRAM TEST
C
C     SAMPLE ADF TEST PROGRAM TO BUILD FILES ILLUSTRATED
C     IN THE EXAMPLE DATABASE FIGURE
C
      INCLUDE 'cgns_io_f.h'
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*4 IC(6),ID(6)
      INTEGER IERR,ICGIO,ICGIO2
      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 CGIO_ERROR_ABORT_F(1)
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 CGIO_OPEN_FILE_F('file_two.cgio',CGIO_MODE_WRITE,
     &                      CGIO_FILE_ADF,ICGIO,IERR)
      CALL CGIO_GET_ROOT_ID_F(ICGIO,RID,IERR)
      RIDF2 = RID
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'F1',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'F2',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'F3',PID,IERR)
```

```
      CALL CGIO_SET_LABEL_F(ICGIO,PID,'LABEL ON NODE F3',IERR)
      CALL CGIO_SET_DIMENSIONS_F(ICGIO,PID,'R4',2,IDIMA,IERR)
      CALL CGIO_WRITE_ALL_DATA_F(ICGIO,PID,A,IERR)
C
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'F4',CID,IERR)
C
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'F5',CID,IERR)
C
      CALL CGIO_CLOSE_FILE_F(ICGIO,IERR)
C
C *** 1.) OPEN 2ND DATABASE
C     2.) CREATE NODES
C     3.) PUT DATA IN N13
C
      CALL CGIO_OPEN_FILE_F('file_one.cgio',CGIO_MODE_WRITE,
     &                      CGIO_FILE_ADF,ICGIO,IERR)
      CALL CGIO_GET_ROOT_ID_F(ICGIO,RID,IERR)
C
C     THREE NODES UNDER ROOT
C
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'N1',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'N2',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,RID,'N3',TMPID,IERR)
C
C     THREE NODES UNDER N1 (TWO REGULAR AND ONE LINK)
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'N1',PID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N4',TMPID,IERR)
      CALL CGIO_CREATE_LINK_F(ICGIO,PID,'L3','file_two.cgio','/F3',
     &                        TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N5',TMPID,IERR)
C
C     TWO NODES UNDER N4
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,PID,'N4',CID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,CID,'N6',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,CID,'N7',TMPID,IERR)
C
C     ONE NODE UNDER N6
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'/N1/N4/N6',PID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N8',TMPID,IERR)
C
C     THREE NODES UNDER N3
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'N3',PID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N9',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N10',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N11',TMPID,IERR)
C
C     TWO NODES UNDER N9
```

```
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,PID,'N9',CID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,CID,'N12',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,CID,'N13',TMPID,IERR)
C
C     PUT LABEL AND DATA IN N13
C
      CALL CGIO_SET_LABEL_F(ICGIO,TMPID,'LABEL ON NODE N13',IERR)
      CALL CGIO_SET_DIMENSIONS_F(ICGIO,TMPID,'I4',1,IDIMC,IERR)
      CALL CGIO_WRITE_ALL_DATA_F(ICGIO,TMPID,IC,IERR)
C
C     TWO NODES UNDER N10
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'/N3/N10',PID,IERR)
      CALL CGIO_CREATE_LINK_F(ICGIO,PID,'L1','','/N3/N9/N13',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N14',TMPID,IERR)
C
C     TWO NODES UNDER N11
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'/N3/N11',PID,IERR)
      CALL CGIO_CREATE_LINK_F(ICGIO,PID,'L2','','/N3/N9/N13',TMPID,IERR)
      CALL CGIO_CREATE_NODE_F(ICGIO,PID,'N15',TMPID,IERR)
C
C *** READ AND PRINT DATA FROM NODES
C     1.) NODE F5 THROUGH LINK L3
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'/N1/L3',PID,IERR)
      CALL CGIO_GET_LABEL_F(ICGIO,PID,TSTLBL,IERR)
      CALL CGIO_GET_DATA_TYPE_F(ICGIO,PID,DTYPE,IERR)
      CALL CGIO_GET_DIMENSIONS_F(ICGIO,PID,NUMDIM,IDIM,IERR)
      CALL CGIO_READ_ALL_DATA_F(ICGIO,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 CGIO_GET_NODE_ID_F(ICGIO,RID,'N3/N9/N13',PID,IERR)
      CALL CGIO_GET_LABEL_F(ICGIO,PID,TSTLBL,IERR)
      CALL CGIO_GET_DATA_TYPE_F(ICGIO,PID,DTYPE,IERR)
      CALL CGIO_GET_DIMENSIONS_F(ICGIO,PID,NUMDIM,IDIMD,IERR)
      CALL CGIO_READ_ALL_DATA_F(ICGIO,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 CGIO_GET_NODE_ID_F(ICGIO,RID,'N3/N10/L1',TMPID,IERR)
      CALL CGIO_GET_LABEL_F(ICGIO,TMPID,TSTLBL,IERR)
      CALL CGIO_READ_ALL_DATA_F(ICGIO,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 CGIO_GET_NODE_ID_F(ICGIO,RID,'N3/N11/L2',CID,IERR)
      CALL CGIO_GET_LABEL_F(ICGIO,CID,TSTLBL,IERR)
      CALL CGIO_READ_ALL_DATA_F(ICGIO,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(ICGIO,RID)
C
C     PRINT LIST OF CHILDREN UNDER N3
C
      CALL CGIO_GET_NODE_ID_F(ICGIO,RID,'N3',PID,IERR)
      CALL PRTCLD(ICGIO,PID)
C
C     REOPEN ADF_FILE_TWO AND GET NEW ROOT ID
C
      CALL CGIO_OPEN_FILE_F('file_two.cgio',CGIO_MODE_READ,
     &                      CGIO_FILE_ADF,ICGIO2,IERR)
      CALL CGIO_GET_ROOT_ID_F(ICGIO2,RID,IERR)
      PRINT *,' '
      PRINT *,' COMPARISON OF ROOT ID: '
      PRINT *,' file_two.cgio ORIGINAL ROOT ID = ',RIDF2
      PRINT *,' file_two.cgio NEW ROOT ID      = ',RID
C
      CALL CGIO_CLOSE_FILE_F(ICGIO,IERR)
      CALL CGIO_CLOSE_FILE_F(ICGIO2,IERR)
C
      STOP
```

```
      END
C
C ************* SUBROUTINES ****************
C
      SUBROUTINE PRTCLD(ICGIO,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 CGIO_GET_NAME_F(ICGIO,PID,NODNAM,IERR)
      CALL CGIO_NUMBER_CHILDREN_F(ICGIO,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 CGIO_CHILDREN_NAMES_F(ICGIO,PID,ISTART,MAXCLD,MAXCHR,
     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:
  file_two.cgio ORIGINAL ROOT ID =   2.
  file_two.cgio NEW ROOT ID      =   3.
```

10.2 C Example

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

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

#include "cgns_io.h"

void print_child_list(int cgio_num,double node_id);

int main ()
{
  /* --- Node header character strings */
   char label[CGIO_MAX_LABEL_LENGTH+1];
   char data_type[CGIO_MAX_DATATYPE_LENGTH+1];
```

```
/* --- Database identifier */
int cgio_num, cgio_num2;

/* --- 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}
                 };
 cgsize_t a_dimensions[2] = {4,3};

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

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

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

/* --- set database error flag to abort on error */
 cgio_error_abort(error_state);

/* -------- build file: file_two.cgio ---------- */
/* --- 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 */

 cgio_open_file("file_two.cgio",CGIO_MODE_WRITE,CGIO_FILE_NONE,&cgio_num);
 cgio_get_root_id(cgio_num,&root_id);
 root_id_file2 = root_id;
 cgio_create_node(cgio_num,root_id,"f1",&tmp_id);
 cgio_create_node(cgio_num,root_id,"f2",&tmp_id);
 cgio_create_node(cgio_num,root_id,"f3",&parent_id);
 cgio_set_label(cgio_num,parent_id,"label on node f3");

 cgio_set_dimensions(cgio_num,parent_id,"R4",2,a_dimensions);
 cgio_write_all_data(cgio_num,parent_id,a);

 cgio_create_node(cgio_num,parent_id,"f4",&child_id);
 cgio_create_node(cgio_num,parent_id,"f5",&child_id);
 cgio_close_file(cgio_num);
```

```
/* -------- build file: file_one.cgio ---------- */
/* open database and create three nodes at first level */
 cgio_open_file("file_one.cgio",CGIO_MODE_WRITE,CGIO_FILE_NONE,&cgio_num);
 cgio_get_root_id(cgio_num,&root_id);
 cgio_create_node(cgio_num,root_id,"n1",&tmp_id);
 cgio_create_node(cgio_num,root_id,"n2",&tmp_id);
 cgio_create_node(cgio_num,root_id,"n3",&tmp_id);

/* put three nodes under n1 (two regular and one link) */
 cgio_get_node_id(cgio_num,root_id,"n1",&parent_id);
 cgio_create_node(cgio_num,parent_id,"n4",&tmp_id);
 cgio_create_link(cgio_num,parent_id,"l3","file_two.cgio","/f3",&tmp_id);
 cgio_create_node(cgio_num,parent_id,"n5",&tmp_id);

/* put two nodes under n4 */
 cgio_get_node_id(cgio_num,parent_id,"n4",&child_id);
 cgio_create_node(cgio_num,child_id,"n6",&tmp_id);
 cgio_create_node(cgio_num,child_id,"n7",&tmp_id);

/* put one nodes under n6 */
 cgio_get_node_id(cgio_num,root_id,"/n1/n4/n6",&parent_id);
 cgio_create_node(cgio_num,parent_id,"n8",&tmp_id);

/* put three nodes under n3 */
 cgio_get_node_id(cgio_num,root_id,"n3",&parent_id);
 cgio_create_node(cgio_num,parent_id,"n9",&tmp_id);
 cgio_create_node(cgio_num,parent_id,"n10",&tmp_id);
 cgio_create_node(cgio_num,parent_id,"n11",&tmp_id);

/* put two nodes under n9 */
 cgio_get_node_id(cgio_num,parent_id,"n9",&child_id);
 cgio_create_node(cgio_num,child_id,"n12",&tmp_id);
 cgio_create_node(cgio_num,child_id,"n13",&tmp_id);

/* put label and data in n13 */
 cgio_set_label(cgio_num,tmp_id,"Label on Node n13");
 cgio_set_dimensions(cgio_num,tmp_id,"I4",1,&c_dimension);
 cgio_write_all_data(cgio_num,tmp_id,c);

/* put two nodes under n10 (one normal, one link) */
 cgio_get_node_id(cgio_num,root_id,"/n3/n10",&parent_id);
 cgio_create_link(cgio_num,parent_id,"l1"," ","/n3/n9/n13",&tmp_id);
 cgio_create_node(cgio_num,parent_id,"n14",&tmp_id);

/* put two nodes under n11 (one normal, one link) */
 cgio_get_node_id(cgio_num,root_id,"/n3/n11",&parent_id);
 cgio_create_link(cgio_num,parent_id,"l2"," ","/n3/n9/n13",&tmp_id);
 cgio_create_node(cgio_num,parent_id,"n15",&tmp_id);

/* ----------------- finished building file_one.cgio ------------- */
```

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

/* access data in node f3 (file_two.cgio) through link l3 */
 cgio_get_node_id(cgio_num,root_id,"/n1/l3",&tmp_id);
 cgio_get_label(cgio_num,tmp_id,label);
 cgio_get_data_type(cgio_num,tmp_id,data_type);
 cgio_get_dimensions(cgio_num,tmp_id,&num_dims,dims_b);
 cgio_read_all_data(cgio_num,tmp_id,b);
 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 */
 cgio_get_node_id(cgio_num,root_id,"/n3/n9/n13",&tmp_id);
 cgio_get_label(cgio_num,tmp_id,label);
 cgio_get_data_type(cgio_num,tmp_id,data_type);
 cgio_get_dimensions(cgio_num,tmp_id,&num_dims,&dim_d);
 cgio_read_all_data(cgio_num,tmp_id,d);
 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 */
 cgio_get_node_id(cgio_num,root_id,"/n3/n10/l1",&tmp_id);
 cgio_get_label(cgio_num,tmp_id,label);
 cgio_read_all_data(cgio_num,tmp_id,d);
 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 */
  cgio_get_node_id(cgio_num,root_id,"/n3/n11/l2",&tmp_id);
  cgio_get_label(cgio_num,tmp_id,label);
  cgio_read_all_data(cgio_num,tmp_id,d);
  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(cgio_num,root_id);

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

 /* re-open file_two and get new root id */
  cgio_open_file("file_two.cgio",CGIO_MODE_READ,CGIO_FILE_NONE,&cgio_num2);
  cgio_get_root_id(cgio_num2,&root_id);
  printf (" Comparison of root id:\n");
  printf ("   file_two.cgio original root id = %g\n",root_id_file2);
  printf ("   file_two.cgio new      root id = %g\n",root_id);

  cgio_close_file(cgio_num);
  cgio_close_file(cgio_num2);
  return 0;
}

void print_child_list(int cgio_num, double node_id)
{

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

   cgio_get_name(cgio_num,node_id,node_name);
   cgio_number_children(cgio_num,node_id,&num_children);
   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++)
```

```
    {
      cgio_children_names(cgio_num,node_id,i,1,CGIO_MAX_NAME_LENGTH+1,
          &num_ret,node_name);
      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:
  file_two.cgio original root id = 2
  file_two.cgio new      root id = 3
```