CFD General Notation System Mid-Level Library

Document Version 3.1.10

CGNS Version 3.1.3

Contents

1 Introduction

This document outlines a CGNS library designed to ease implementation of CGNS by providing developers with a collection of handy I/O functions. Since knowledge of database manager and file structure is not required to use this library, it greatly facilitates the task of interfacing with CGNS.

The CGNS Mid-Level Library is based on the SIDS File Mapping Manual. It allows reading and writing all of the information described in that manual including grid coordinates, block interfaces, flow solutions, and boundary conditions. Use of the mid-level library functions insures efficient communication between the user application and the internal representation of the CGNS data.

It is assumed that the reader is familiar with the information in the CGNS Standard Interface Data Structures (SIDS), as well as the SIDS File Mapping Manual. The reader is also strongly encouraged to read the User's Guide to CGNS, which contains coding examples using the Mid-Level Library to write and read simple files containing CGNS databases.

2 General Remarks

2.1 Acquiring the Software and Documentation

The CGNS Mid-Level Library may be downloaded 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/CGNS_docs_current/*.

2.2 Organization of This Manual

The sections that follow describe the Mid-Level Library functions in detail. The first three sections cover some basic file operations (i.e., opening and closing a CGNS file, and some configuration options) (Section 3), accessing a specific node in a CGNS database (Section 4), and error handling (Section 5). The remaining sections describe the functions used to read, write, and modify nodes and data in a CGNS database. These sections basically follow the organization used in the "Detailed CGNS Node Descriptions" section of the SIDS File Mapping manual.

At the start of each sub-section is a *Node* line, listing the the applicable CGNS node label.

Next is a table illustrating the syntax for the Mid-Level Library functions. The C functions are shown in the top half of the table, followed by the corresponding Fortran routines in the bottom half of the table. Input variables are shown in an `upright blue` font, and output variables are shown in a *`slanted red`* font. As of Version 3.1, some of the arguments to the Mid-Level Library have changed from `int` to `cgsize_t` in order to support 64-bit data. For each function, the right-hand column lists the modes (read, write, and/or modify) applicable to that function.

The input and output variables are then listed and defined.

2.3 Language

The CGNS Mid-Level Library is written in C, but each function has a Fortran counterpart. All function names start with "`cg_`". The Fortran functions have the same name as their C counterpart with the addition of the suffix "`_f`".

2.4 Character Strings

All data structure names and labels in CGNS are limited to 32 characters. When reading a file, it is advised to pre-allocate the character string variables to 32 characters in Fortran, and 33 in C (to include the string terminator). Other character strings, such as the CGNS file name or descriptor text, are unlimited in length. The space for unlimited length character strings will be created by the Mid-Level Library; it is then the responsibility of the application to release this space by a call to `cg_free`, described in Section 10.6.

2.5 Error Status

All C functions return an integer value representing the error status. All Fortran functions have an additional parameter, `ier`, which contains the value of the error status. An error status different from zero implies that an error occured. The error message can be printed using the error handling functions of the CGNS library, described in Section 5. The error codes are coded in the C and Fortran include files *cgnslib.h* and *cgnslib_f.h*.

2.6 Typedefs

Beginning with version 3.1, two new typedef variables have been introduced to support 64-bit mode. The `cglong_t` typedef is always a 64-bit integer, and `cgsize_t` will be either a 32-bit or 64-bit integer depending on how the library was built. Many of the C functions in the MLL have been changed to to use `cgsize_t` instead of `int` in the arguments. These functions include any that may exceed the `2Gb` limit of an `int`, e.g. zone dimensions, element data, boundary conditions, and connectivity. In Fortran, all integer data is taken to be `integer*4` for 32-bit and `integer*8` for 64-bit builds.

Several types of variables are defined using typedefs in the *cgnslib.h* file. These are intended to facilitate the implementation of CGNS in C. These variable types are defined as an enumeration of key words admissible for any variable of these types. The file *cgnslib.h* must be included in any C application programs which use these data types.

In Fortran, the same key words are defined as integer parameters in the include file *cgnslib_f.h*. Such variables should be declared as `integer` in Fortran applications. The file *cgnslib_f.h* must be included in any Fortran application using these key words.

Note that the first two enumerated values in these lists, *xxxNull* and *xxxUserDefined*, are only available in the C interface, and are provided in the advent that your C compiler does strict type checking. In Fortran, these values are replaced by the numerically equivalent `CG_Null` and `CG_UserDefined`. These values are also defined in the C interface, thus either form may be used. The function prototypes for the MLL use `CG_Null` and `CG_UserDefined`, rather than the more specific values.

The list of enumerated values (key words) for each of these variable types (typedefs) are:

`ZoneType_t`	`ZoneTypeNull, ZoneTypeUserDefined, Structured, Unstructured`
`ElementType_t`	`ElementTypeNull, ElementTypeUserDefined, NODE, BAR_2, BAR_3, TRI_3, TRI_6, QUAD_4, QUAD_8, QUAD_9, TETRA_4, TETRA_10, PYRA_5, PYRA_14, PENTA_6, PENTA_15, PENTA_18, HEXA_8, HEXA_20, HEXA_27, MIXED, PYRA_13, NGON_n, NFACE_n`
`DataType_t`	`DataTypeNull, DataTypeUserDefined, Integer, RealSingle, RealDouble, Character`
`DataClass_t`	`DataClassNull, DataClassUserDefined, Dimensional, NormalizedByDimensional, NormalizedByUnknownDimensional, NondimensionalParameter, DimensionlessConstant`
`MassUnits_t`	`MassUnitsNull, MassUnitsNullUserDefined, Kilogram, Gram, Slug, PoundMass`
`LengthUnits_t`	`LengthUnitsNull, LengthUnitsUserDefined, Meter, Centimeter, Millimeter, Foot, Inch`

TimeUnits_t	TimeUnitsNull, TimeUnitsUserDefined, Second
TemperatureUnits_t	TemperatureUnitsNull, TemperatureUnitsUserDefined, Kelvin, Celsius, Rankine, Fahrenheit
AngleUnits_t	AngleUnitsNull, AngleUnitsUserDefined, Degree, Radian
ElectricCurrentUnits_t	ElectricCurrentUnitsNull, ElectricCurrentUnitsUserDefined, Ampere, Abampere, Statampere, Edison, auCurrent
SubstanceAmountUnits_t	SubstanceAmountUnitsNull, SubstanceAmountUnitsUserDefined, Mole, Entities, StandardCubicFoot, StandardCubicMeter
LuminousIntensityUnits_t	LuminousIntensityUnitsNull, LuminousIntensityUnitsUserDefined, Candela, Candle, Carcel, Hefner, Violle
GoverningEquationsType_t	GoverningEquationsTypeNull, GoverningEquationsTypeUserDefined, FullPotential, Euler, NSLaminar, NSTurbulent, NSLaminarIncompressible, NSTurbulentIncompressible
ModelType_t	ModelTypeNull, ModelTypeUserDefined, Ideal, VanderWaals, Constant, PowerLaw, SutherlandLaw, ConstantPrandtl, EddyViscosity, ReynoldsStress, ReynoldsStressAlgebraic, Algebraic_BaldwinLomax, Algebraic_CebeciSmith, HalfEquation_JohnsonKing, OneEquation_BaldwinBarth, OneEquation_SpalartAllmaras, TwoEquation_JonesLaunder, TwoEquation_MenterSST, TwoEquation_Wilcox, CaloricallyPerfect, ThermallyPerfect, ConstantDensity, RedlichKwong, Frozen, ThermalEquilib, ThermalNonequilib, ChemicalEquilibCurveFit, ChemicalEquilibMinimization, ChemicalNonequilib, EMElectricField, EMMagneticField, EMConductivity, Voltage, Interpolated, Equilibrium_LinRessler, Chemistry_LinRessler
GridLocation_t	GridLocationNull, GridLocationUserDefined, Vertex, IFaceCenter, CellCenter, JFaceCenter, FaceCenter, KFaceCenter, EdgeCenter
GridConnectivityType_t	GridConnectivityTypeNull, GridConnectivityTypeUserDefined, Overset, Abutting, Abutting1to1
PointSetType_t	PointSetTypeNull, PointSetTypeUserDefined, PointList, PointRange, PointListDonor, PointRangeDonor, ElementList, ElementRange, CellListDonor
BCType_t	BCTypeNull, BCTypeUserDefined, BCAxisymmetricWedge, BCDegenerateLine, BCExtrapolate, BCDegeneratePoint,

	BCDirichlet, BCFarfield, BCNeumann, BCGeneral, BCInflow, BCOutflow, BCInflowSubsonic, BCOutflowSubsonic, BCInflowSupersonic, BCOutflowSupersonic, BCSymmetryPlane, BCTunnelInflow, BCSymmetryPolar, BCTunnelOutflow, BCWallViscous, BCWall, BCWallViscousHeatFlux, BCWallInviscid, BCWallViscousIsothermal, FamilySpecified
BCDataType_t	BCDataTypeNull, BCDataTypeUserDefined, Dirichlet, Neumann
RigidGridMotionType_t	RigidGridMotionTypeNull, RigidGridMotionTypeUserDefined, ConstantRate, VariableRate
ArbitraryGridMotionType_t	ArbitraryGridMotionTypeNull, ArbitraryGridMotionTypeUserDefined, NonDeformingGrid, DeformingGrid
SimulationType_t	SimulationTypeNull, SimulationTypeUserDefined, TimeAccurate, NonTimeAccurate
WallFunctionType_t	WallFunctionTypeNull, WallFunctionTypeUserDefined, Generic
AreaType_t	AreaTypeNull, AreaTypeUserDefined, BleedArea, CaptureArea
AverageInterfaceType_t	AverageInterfaceTypeNull, AverageInterfaceTypeUserDefined, AverageAll, AverageCircumferential, AverageRadial, AverageI, AverageJ, AverageK

2.7 Character Names For Typedefs

The CGNS library defines character arrays which map the typedefs above to character strings. These are global arrays dimensioned to the size of each list of typedefs. To retrieve a character string representation of a typedef, use the typedef value as an index to the appropiate character array. For example, to retrieve the string "`Meter`" for the `LengthUnits_t Meter` typedef, use `LengthUnitsName[Meter]`. Functions are available to retrieve these names without the need for direct global data access. These functions also do bounds checking on the input, and if out of range, will return the string "`<invalid>`". An additional benefit is that these will work from within a Windows DLL, and are thus the recommended access technique. The routines have the same name as the global data arrays, but with a "`cg_`" prepended. For the example above, use "`cg_LengthUnitsName(Meter)`".

Typedef Name Access Functions
const char *name = cg_MassUnitsName(MassUnits_t type);
const char *name = cg_LengthUnitsName(LengthUnits_t type);
const char *name = cg_TimeUnitsName(TimeUnits_t type);
const char *name = cg_TemperatureUnitsName(TemperatureUnits_t type);
const char *name = cg_ElectricCurrentUnitsName(ElectricCurrentUnits_t type);
const char *name = cg_SubstanceAmountUnitsName(SubstanceAmountUnits_t type);
const char *name = cg_LuminousIntensityUnitsName(LuminousIntensityUnits_t type);
const char *name = cg_DataClassName(DataClass_t type);
const char *name = cg_GridLocationName(GridLocation_t type);
const char *name = cg_BCDataTypeName(BCDataType_t type);
const char *name = cg_GridConnectivityTypeName(GridConnectivityType_t type);
const char *name = cg_PointSetTypeName(PointSetType_t type);
const char *name = cg_GoverningEquationsTypeName(GoverningEquationsType_t type);
const char *name = cg_ModelTypeName(ModelType_t type);
const char *name = cg_BCTypeName(BCType_t type);
const char *name = cg_DataTypeName(DataType_t type);
const char *name = cg_ElementTypeName(ElementType_t type);
const char *name = cg_ZoneTypeName(ZoneType_t type);
const char *name = cg_RigidGridMotionTypeName(RigidGridMotionType_t type);
const char *name = cg_ArbitraryGridMotionTypeName(ArbitraryGridMotionType_t type);
const char *name = cg_SimulationTypeName(SimulationType_t type);
const char *name = cg_WallFunctionTypeName(WallFunctionType_t type);
const char *name = cg_AreaTypeName(AreaType_t type);
const char *name = cg_AverageInterfaceTypeName(AverageInterfaceType_t type);

2.8 64-bit C Portability and Issues

If you use the `cgsize_t` data type in new code, it will work in both 32 and 64-bit compilation modes. In order to support CGNS versions prior to 3.1, you may also want to add something like this to your code:

```
#if CGNS_VERSION < 3100
#define cgsize_t int
#endif
```

Existing code that uses `int` will not work with a CGNS 3.1 library compiled in 64-bit mode. You may want to add something like this to your code:

```
#if CGNS_VERSION >= 3100 && CG_BUILD_64BIT
#error does not work in 64 bit mode
#endif
```

or modify your code to use `cgsize_t`.

2.9 64-bit Fortran Portability and Issues

All integer arguments in the Fortran interface - including enumerated values (enums) - are taken to be `integer*4` in 32-bit mode and `integer*8` in 64-bit mode. If you have used default or implicit integers in your Fortran code, it should port to 64-bit mode in most cases by simply turning on your compiler option that promotes implicit integers to `integer*8`. If you have explicitly defined your integers as `integer*4`, your code will not work in 64-bit mode. In that case, you will either need to change them to `integer` (recommended for portability) or `integer*8`.

A new integer parameter has been added to the *cgnslib_f.h* header, `CG_BUILD_64BIT`, which will be set to 1 in 64-bit mode and 0 otherwise. You may use this parameter to check at run time if the CGNS library has been compiled in 64-bit mode or not, as in:

```
if (CG_BUILD_64BIT .ne.  0) then
   print ,'will not work in 64-bit mode'
   stop
endif
```

If you are using a CGNS library prior to version 3.1, this parameter will not be defined and you will need to rely on your compiler initializing all undefined values to 0 (not always the case) for this test to work.

If your compiler supports automatic promotion of integers, and you use implicit integers, your code should port to 64-bit with the following exception.

If you use an `Integer` data type in any routine that takes a data type specification, and an implicit integer for the data, the code will fail when compiled in 64-bit mode with automatic integer promotion. An example of this would be:

```
integer dim
integer data(dim)
call cg_array_write_f('array',Integer,1,dim,data)
```

This is because the MLL interprets the `Integer` data type as `integer*4` regardless of the compilation mode. The compiler, however, has automatically promoted `data` to be `integer*8`. What you will need to do to prevent this problem, is to either explicitly define `data` as in:

```
integer dim
integer*4 data(dim)
call cg_array_write_f('array',Integer,1,dim,data)
```

or

```
integer dim
integer*8 data(dim)
call cg_array_write_f('array',LongInteger,1,dim,data)
```

or test on `CG_BUILD_64BIT` as in:

```
integer dim
integer data(dim)
if (CG_BUILD_64BIT .eq.  0) then
   call cg_array_write_f('array',Integer,1,dim,data)
else
   call cg_array_write_f('array',LongInteger,1,dim,data)
endif
```

The last 2 options will only work with CGNS Version 3.1, since `LongInteger` and `CG_BUILD_64BIT` are not defined in previous versions.

You may also need to be careful when using integer constants as arguments in 64-bit mode. If your compiler automatically promotes integer constants to `integer*8`, then there is no problem. This is probably the case if your compiler supports implicit integer promotion. If not, then the constants will be `integer*4`, and your code will not work in 64-bit mode. In that case you will need to do something like:

```
integer*8 one,dim
integer*4 data(dim)
one = 1
call cg_array_write_f('array',Integer,one,dim,data)
```

3 File Operations

3.1 Opening and Closing a CGNS File

Functions	Modes
`ier = cg_open(char *filename, int mode, int *fn);`	r w m
`ier = cg_version(int fn, float *version);`	r w m
`ier = cg_close(int fn);`	r w m
`ier = cg_is_cgns(const char *filename, int *file_type);`	r w m
`ier = cg_save_as(int fn, const char *filename, int file_type, int follow_links);`	r w m
`ier = cg_set_file_type(int file_type);`	r w m
`ier = cg_get_file_type(int fn, int *file_type);`	r w m
`call cg_open_f(filename, mode, fn, ier)`	r w m
`call cg_version_f(fn, version, ier)`	r w m
`call cg_close_f(fn, ier)`	r w m
`call cg_is_cgns_f(filename, file_type, ier)`	r w m
`call cg_save_as_f(fn, filename, file_type, follow_links, ier)`	r w m
`call cg_set_file_type_f(file_type, ier)`	r w m
`call cg_get_file_type_f(fn, file_type, ier)`	r w m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`filename`	Name of the CGNS file, including path name if necessary. There is no limit on the length of this character variable. (Input)
`mode`	Mode used for opening the file. The modes currently supported are `CG_MODE_READ`, `CG_MODE_WRITE`, and `CG_MODE_MODIFY`. (Input)
`fn`	CGNS file index number. (Input for `cg_version` and `cg_close`; *output* for `cg_open`)
`version`	CGNS version number. (*Output*)
`file_type`	Type of CGNS file. This will typically be either `CG_FILE_ADF` or `CG_FILE_HDF5` depending on the underlying file format. (Input for `cg_save_as` and `cg_set_file_type`; *output* for `cg_get_file_type`) However, note that when built in 32-bit, there is also an option to create a Version 2.5 CGNS file by setting the file type to `CG_FILE_ADF2`.
`follow_links`	This flag determines whether links are left intact when saving a CGNS file. If non-zero, then the links will be removed and the data associated with the linked files copied to the new file. (Input)
`ier`	Error status. (*Output*)

The function cg_open must always be the first one called. It opens a CGNS file for reading and/or writing and returns an index number fn. The index number serves to identify the CGNS file in subsequent function calls. Several CGNS files can be opened simultaneously. The current limit on the number of files opened at once depends on the platform. On an SGI workstation, this limit is set at 100 (parameter FOPEN_MAX in *stdio.h*).

The file can be opened in one of the following modes:

CG_MODE_READ	Read only mode.
CG_MODE_WRITE	Write only mode.
CG_MODE_MODIFY	Reading and/or writing is allowed.

When the file is opened, if no CGNSLibraryVersion_t node is found, a default value of 1.05 is assumed for the CGNS version number. Note that this corresponds to an old version of the CGNS standard, that doesn't include many data structures supported by the current standard.

The function cg_close must always be the last one called. It closes the CGNS file designated by the index number fn and frees the memory where the CGNS data was kept. When a file is opened for writing, cg_close writes all the CGNS data in memory onto disk prior to closing the file. Consequently, if is omitted, the CGNS file is not written properly.

In order to reduce memory usage and improve execution speed, large arrays such as grid coordinates or flow solutions are not actually stored in memory. Instead, only basic information about the node is kept, while reads and writes of the data is directly to and from the application's memory. An attempt is also made to do the same with unstructured mesh element data.

The function cg_is_cgns may be used to determine if a file is a CGNS file or not, and the type of file (CG_FILE_ADF or CG_FILE_HDF5). If the file is a CGNS file, cg_is_cgns returns CG_OK, otherwise CG_ERROR is returned and file_type is set to CG_FILE_NONE.

The CGNS file identified by fn may be saved to a different filename and type using cg_save_as. In order to save as an HDF5 file, the library must have been built with HDF5 support. ADF support is always built. The function cg_set_file_type sets the default file type for newly created CGNS files. The function cg_get_file_type returns the file type for the CGNS file identified by fn. If the CGNS library is built as 32-bit, the additional file type, CG_FILE_ADF2, is available. This allows creation of a 2.5 compatible CGNS file.

3.2 Configuring CGNS Internals

Functions	Modes
`ier = cg_configure(int option, void *value);`	r w m
`ier = cg_error_handler(void (*)(int, char *));`	r w m
`ier = cg_set_compress(int compress);`	r w m
`ier = cg_get_compress(int *compress);`	r w m
`ier = cg_set_path(const char *path);`	r w m
`ier = cg_add_path(const char *path);`	r w m
`call cg_set_compress_f(compress, ier)`	r w m
`call cg_get_compress_f(compress, ier)`	r w m
`call cg_set_path_f(path, ier)`	r w m
`call cg_add_path_f(path, ier)`	r w m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`option` The option to configure, currently one of `CG_CONFIG_ERROR`, `CG_CONFIG_COMPRESS`, `CG_CONFIG_SET_PATH`, `CG_CONFIG_ADD_PATH`, or `CG_CONFIG_HDF5_COMPRESS` as defined in *cgnslib.h*. (Input)

`value` The value to set, type cast as `void *`. (Input)

`compress` CGNS compress (rewrite) setting). (Input for `cg_set_compress`; *output* for `cg_get_compress`)

`path` Pathname to search for linked to files when opening a file with external links. (Input)

`ier` Error status. (*Output*)

The function `cg_configure` allows certain CGNS library internal options to be configured. The currently supported options and expected values are:

`CG_CONFIG_ERROR` This allows an error call-back function to be defined by the user. The value should be a pointer to a function to receive the error. The function is defined as `void err_callback(int is_error, char *errmsg)`, and will be called for errors and warnings. The first argument, `is_error`, will be 0 for warning messages, 1 for error messages, and −1 if the program is going to terminate (i.e., a call to `cg_error_exit()`). The second argument is the error or warning message. If this is defined, warning and error messages will go to the function, rather than the terminal. A `value` of `NULL` will remove the call-back function.

`CG_CONFIG_COMPRESS` When a CGNS file is closed after being opened in modify mode, the normal operation of the CGNS library is to rewrite the file if there is unused space. This happens when nodes have been rewritten or deleted. Setting `value` to 0 will prevent the library from rewriting the file, and setting it to 1 will force the rewrite. The default value is −1.

`CG_CONFIG_SET_PATH` Sets the search path for locating linked-to files. The argument `value` should be a character string containing one or more directories, formatted the same as for the `PATH` environment variable. This will replace any current settings. Setting `value` to `NULL` will remove all paths.

`CG_CONFIG_ADD_PATH` Adds a directory, or list of directories, to the linked-to file search path. This is the same as `CG_CONFIG_SET_PATH`, but adds to the path instead of replacing it.

`CG_CONFIG_HDF5_COMPRESS`
Sets the compression level for data written from HDF5. The default is no compression. Setting `value` to -1, will use the default compression level of 6. The acceptable values are 0 to 9, corresponding to gzip compression levels.

The routines `cg_error_handler`, `cg_set_compress`, `cg_set_path`, and `cg_add_path` are convenience functions built on top of `cg_configure`.

There is no Fortran counterpart to function `cg_configure` or `cg_error_handler`.

Note: The HDF5 implementation does not support search paths for linked files. The links need to be either absolute or relative pathnames. As a result, it is recommended that the search path options not be used as they may be removed in future versions.

4 Navigating a CGNS File

4.1 Accessing a Node

Functions	Modes
`ier = cg_goto(int fn, int B, ..., "end");`	r w m
`ier = cg_gorel(int fn, ..., "end");`	r w m
`ier = cg_gopath(int fn, const char *path);`	r w m
`ier = cg_golist(int fn, int B, int depth, char **label, int *index);`	r w m
`ier = cg_gowhere(int *fn, int *B, int *depth, char **label, int *index);`	r w m
`call cg_goto_f(fn, B, ier, ..., 'end')`	r w m
`call cg_gorel_f(fn, ier, ..., 'end')`	r w m
`call cg_gopath_f(fn, path, ier)`	r w m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`...` Variable argument list used to specify the path to a node. It is composed of an unlimited list of pair-arguments identifying each node in the path. Nodes may be identified by their label or name. Thus, a pair-argument may be of the form

```
"CGNS_NodeLabel", NodeIndex
```

where `CGNS_NodeLabel` is the node label and *NodeIndex* is the node index, or

```
"CGNS_NodeName", 0
```

where `CGNS_NodeName` is the node name. The 0 in the second form is required, to indicate that a node name is being specified rather than a node label. In addition, a pair-argument may be specified as

```
"..", 0
```

indicating the parent of the current node. The different pair-argument forms may be intermixed in the same function call.

There is one exception to this rule. When accessing a `BCData_t` node, the index must be set to either `Dirichlet` or `Neumann` since only these two types are allowed. (Note that `Dirichlet` and `Neumann` are defined in the include files *cgnslib.h* and *cgnslib_f.h*). Since ”`Dirichlet`” and ”`Neuman`” are also the names for these nodes, you may also use the `"Dirichlet", 0` or `"Neuman", 0` to access the node. See the example below. (Input) `end` The character string `"end"` (or `'end'` for the Fortran function) must be the last argument. It is used to indicate the end of the argument list. You may also use the empty string, `""` (`''` for Fortran), or the `NULL` string in C, to terminate the list. (Input)

`path` The pathname for the node to go to. If a position has been already set, this may be a relative path, otherwise it is an absolute path name, starting with `"/Basename"`, where `Basename` is the base under which you wish to move. (Input)

`depth` Depth of the path list. The maximum depth is defined in *cgnslib.h* by `CG_MAX_GOTO_DEPTH`, and is currently equal to 20. (Input for `cg_golist`; *output* for `cg_gowhere`)

`label` Array of node labels for the path. This argument may be passed as `NULL` to `cg_where()`, otherwise it must be dimensioned by the calling program. The maximum size required is `label[MAX_GO_TO_DEPTH][33]`. You may call `cg_where()` with both `label` and `index` set to `NULL` in order to get the current depth, then dimension to that value. (Input for `cg_golist`; *output* for `cg_gowhere`)

`index` Array of node indices for the path. This argument may be passed as `NULL` to `cg_where()`, otherwise it must be dimensioned by the calling program. The maximum size required is `index[MAX_GO_TO_DEPTH]`. You may call `cg_where()` with both `label` and `index` set to `NULL` in order to get the current depth, then dimension to that value. (Input for `cg_golist`; *output* for `cg_gowhere`)

`ier` Error status. The possible values, with the corresponding C names (or Fortran parameters) defined in *cgnslib.h* (or *cgnslib_f.h*) are listed below.

Value	*Name/Parameter*
0	`CG_OK`
1	`CG_ERROR`
2	`CG_NODE_NOT_FOUND`
3	`CG_INCORRECT_PATH`

For non-zero values, an error message may be printed using `cg_error_print()`, as described in Section 5. (*Output*)

This function allows access to any parent-type nodes in a CGNS file. A parent-type node is one that can have children. Nodes that cannot have children, like `Descriptor_t`, are not supported by this function.

Examples

To illustrate the use of the above routines, assume you have a file with CGNS index number `filenum`, a base node named `Base` with index number `basenum`, 2 zones (named `Zone1` and `Zone2`, with indices 1 and 2), and user-defined data (`User`, index 1) below each zone. To move to the user-defined data node under zone 1, you may use any of the following:

```
cg_goto(filenum, basenum, "Zone_t", 1, "UserDefinedData_t", 1, NULL);
cg_goto(filenum, basenum "Zone1", 0, "UserDefinedData_t", 1, NULL);
cg_goto(filenum, basenum, "Zone_t", 1, "User", 0, NULL);
cg_goto(filenum, basenum, "Zone1", 0, "User", 0, NULL);
cg_gopath(filenum, "/Base/Zone1/User");
```

Now, to change to the user-defined data node under zone 2, you may use the full path specification as above, or else a relative path, using one of the following:

```
cg_gorel(filenum, "..", 0, "..", 0, "Zone_t", 2, "UserDefinedData_t", 1, NULL);
cg_gorel(filenum, "..", 0, "..", 0, "Zone2", 0, "UserDefinedData_t", 1, NULL);
cg_gorel(filenum, "..", 0, "..", 0, "Zone_t", 2, "User", 0, NULL);
cg_gorel(filenum, "..", 0, "..", 0, "Zone2", 0, "User", 0, NULL);
cg_gopath(filenum, "../../Zone2/User");
```

Shown below are some additional examples of various uses of these routines, in both C and Fortran, where `fn`, `B`, `Z`, etc., are index numbers.

```
ier = cg_goto(fn, B, "Zone_t", Z, "FlowSolution_t", F, "..", 0, "MySolution",
              0, "end");

ier = cg_gorel(fn, "..", 0, "FlowSolution_t", F, NULL);

ier = cg_gopath(fn, "/MyBase/MyZone/MySolution");

ier = cg_gopath(fn, "../../MyZoneBC");

call cg_goto_f(fn, B, ier, 'Zone_t', Z, 'GasModel_t', 1, 'DataArray_t',
               A, 'end')

call cg_goto_f(fn, B, ier, 'Zone_t', Z, 'ZoneBC_t', 1, 'BC_t', BC,
               'BCDataSet_t', S, 'BCData_t', Dirichlet, 'end')

call cg_gorel_f(fn, ier, '..', 0, 'Neumann', 0, '')

call cg_gopath_f(fn, '../../MyZoneBC', ier)
```

4.2 Deleting a Node

Functions	**Modes**
`ier = cg_delete_node(char *NodeName);`	`- - m`
`call cg_delete_node_f(NodeName, ier)`	`- - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`NodeName` Name of the child to be deleted. (Input)

`ier` Error status. (*Output*)

The function `cg_delete_node` is used is conjunction with `cg_goto`. Once positioned at a parent node with `cg_goto`, a child of this node can be deleted with `cg_delete_node`. This function requires a single argument, `NodeName`, which is the name of the child to be deleted.

Since the highest level that can be pointed to with cg_goto is a base node for a CGNS database (CGNSBase_t), the highest-level nodes that can be deleted are the children of a CGNSBase_t node. In other words, nodes located directly under the ADF (or HDF) root node (CGNSBase_t and CGNSLibraryVersion_t) can not be deleted with cg_delete.

A few other nodes are not allowed to be deleted from the database because these are required nodes as defined by the SIDS, and deleting them would make the file non-CGNS compliant. These are:

- Under Zone_t: ZoneType
- Under GridConnectivity1to1_t: PointRange, PointRangeDonor, Transform
- Under OversetHoles_t: PointList and any IndexRange_t
- Under GridConnectivity_t: PointRange, PointList, CellListDonor, PointListDonor
- Under BC_t: PointList, PointRange
- Under GeometryReference_t: GeometryFile, GeometryFormat
- Under Elements_t: ElementRange, ElementConnectivity
- Under Gravity_t: GravityVector
- Under Axisymmetry_t: AxisymmetryReferencePoint, AxisymmetryAxisVector
- Under RotatingCoordinates_t: RotationCenter, RotationRateVector
- Under Periodic_t: RotationCenter, RotationAngle, Translation
- Under AverageInterface_t: AverageInterfaceType
- Under WallFunction_t: WallFunctionType
- Under Area_t: AreaType, SurfaceArea, RegionName

When a child node is deleted, both the database and the file on disk are updated to remove the node. One must be careful not to delete a node from within a loop of that node type. For example, if the number of zones below a CGNSBase_t node is nzones, a zone should never be deleted from within a zone loop! By deleting a zone, the total number of zones (nzones) changes, as well as the zone indexing. Suppose for example that nzones is 5, and that the third zone is deleted. After calling cg_delete_node, nzones is changed to 4, and the zones originally indexed 4 and 5 are now indexed 3 and 4.

5 Error Handling

Functions	**Modes**
error_message = const char *cg_get_error();	r w m
void cg_error_exit();	r w m
void cg_error_print();	r w m
call cg_get_error_f(*error_message*)	r w m
call cg_error_exit_f()	r w m
call cg_error_print_f()	r w m

If an error occurs during the execution of a CGNS library function, signified by a non-zero value of the error status variable `ier`, an error message may be retrieved using the function `cg_get_error`. The function `cg_error_exit` may then be used to print the error message and stop the execution of the program. Alternatively, `cg_error_print` may be used to print the error message and continue execution of the program.

In C, you may define a function to be called automatically in the case of a warning or error using the `cg_configure` routine. The function is of the form `void err_func(int is_error, char *errmsg)`, and will be called whenever an error or warning occurs. The first argument, `is_error`, will be 0 for warning messages, 1 for error messages, and -1 if the program is going to terminate (i.e., a call to `cg_error_exit`). The second argument is the error or warning message.

6 Structural Nodes

6.1 CGNS Base Information

Node: CGNSBase_t

Functions	Modes
`ier = cg_base_write(int fn, char *basename, int cell_dim, int phys_dim, int *B);`	- w m
`ier = cg_nbases(int fn, int *nbases);`	r - m
`ier = cg_base_read(int fn, int B, char *basename, int *cell_dim, int *phys_dim);`	r - m
`ier = cg_cell_dim(int fn, int B, int *cell_dim);`	r - m
`call cg_base_write_f(fn, basename, cell_dim, phys_dim, B, ier)`	- w m
`call cg_nbases_f(fn, nbases, ier)`	r - m
`call cg_base_read_f(fn, B, basename, cell_dim, phys_dim, ier)`	r - m
`call cg_cell_dim_f(fn, B, cell_dim, ier)`	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input for `cg_base_read`; *output* for `cg_base_write`)
`nbases`	Number of bases present in the CGNS file `fn`. (*Output*)
`basename`	Name of the base. (Input for `cg_base_write`; *output* for `cg_base_read`)
`cell_dim`	Dimension of the cells; 3 for volume cells, 2 for surface cells and 1 for line cells. (Input for `cg_base_write`; *output* for `cg_base_read` and `cg_cell_dim`)
`phys_dim`	Number of coordinates required to define a vector in the field. (Input for `cg_base_write`; *output* for `cg_base_read`)
`ier`	Error status. (*Output*)

6.2 Zone Information

Node: `Zone_t`

Functions	Modes
`ier = cg_zone_write(int fn, int B, char *zonename, cgsize_t *size, ZoneType_t zonetype, int *Z);`	`- w m`
`ier = cg_nzones(int fn, int B, int *nzones);`	`r - m`
`ier = cg_zone_read(int fn, int B, int Z, char *zonename, cgsize_t *size);`	`r - m`
`ier = cg_zone_type(int fn, int B, int Z, ZoneType_t *zonetype);`	`r - m`
`ier = cg_index_dim(int fn, int B, int Z, int *index_dim);`	`r - m`
`call cg_zone_write_f(fn, B, zonename, size, zonetype, Z, ier)`	`- w m`
`call cg_nzones_f(fn, B, nzones, ier)`	`r - m`
`call cg_zone_read_f(fn, B, Z, zonename, size, ier)`	`r - m`
`call cg_zone_type_f(fn, B, Z, zonetype, ier)`	`r - m`
`call cg_index_dim_f(fn, B, Z, index_dim, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`Z` Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input for `cg_zone_read`, `cg_zone_type`; *output* for `cg_zone_write`)

`nzones` Number of zones present in base `B`. (*Output*)

`zonename` Name of the zone. (Input for `cg_zone_write`; *output* for `cg_zone_read`)

`size` Number of vertices, cells, and boundary vertices in each (*index*)-dimension.

Note that for unstructured grids, the number of cells is the number of highest order elements. Thus, in three dimensions it's the number of 3-D cells, and in two dimensions it's the number of 2-D cells.

Also for unstructured grids, if the nodes are sorted between internal nodes and boundary nodes, the optional parameter `NBoundVertex` must be set equal to the number of boundary nodes. By default, `NBoundVertex` equals zero, meaning that the nodes are unsorted.

Note that a non-zero value for `NBoundVertex` only applies to unstructured grids. For structured grids, the `NBoundVertex` parameter always equals 0 in all directions.

Mesh Type	*Size*
3D structured	NVertexI, NVertexJ, NVertexK NCellI, NCellJ, NCellK NBoundVertexI $= 0$, NBoundVertexJ $= 0$, NBoundVertexK
2D structured	NVertexI, NVertexJ NCellI, NCellJ NBoundVertexI $= 0$, NBoundVertexJ $= 0$
3D unstructured	NVertex, NCell3D, NBoundVertex
2D unstructured	NVertex, NCell2D, NBoundVertex

(Input for cg_zone_write; *output* for cg_zone_read)

zonetype — Type of the zone. The admissible types are Structured and Unstructured. (Input for cg_zone_write; *output* for cg_zone_type)

index_dim — Index dimension for the zone. For Structured zones, this will be the base cell dimension and for Unstructured zones it will be 1. (*output*)

ier — Error status. (*Output*)

Note that the zones are sorted alphanumerically to insure that they can always be retrieved in the same order (for the same model). *Therefore, users must name their zones alphanumerically to ensure proper retrieval.*

6.3 Simulation Type

Node: SimulationType_t

Functions	Modes
ier = cg_simulation_type_write(int fn, int B, SimulationType_t SimulationType);	- w m
ier = cg_simulation_type_read(int fn, int B, *SimulationType_t SimulationType*);	r - m
call cg_simulation_type_write_f(fn, B, SimulationType, *ier*)	- w m
call cg_simulation_type_read_f(fn, B, *SimulationType*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

fn — CGNS file index number. (Input)

B — Base index number, where $1 \leq$ B $\leq$ nbases. (Input)

SimulationType — Type of simulation. Valid types are CG_Null, CG_UserDefined, TimeAccurate, and NonTimeAccurate. (Input for cg_simulation_type_write; *output* for cg_simulation_type_read)

ier — Error status. (*Output*)

7 Descriptors

7.1 Descriptive Text

Node: `Descriptor_t`

Functions	Modes
`ier = cg_descriptor_write(char *name, char *text);`	`- w m`
`ier = cg_ndescriptors(int *ndescriptors);`	`r - m`
`ier = cg_descriptor_read(int D, char *name, char **text);`	`r - m`
`call cg_descriptor_write_f(name, text, ier)`	`- w m`
`call cg_ndescriptors_f(ndescriptors, ier)`	`r - m`
`call cg_descriptor_size_f(D, size, ier)`	`r - m`
`call cg_descriptor_read_f(D, name, text, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`ndescriptors` Number of `Descriptor_t` nodes under the current node. (Output)

`D` Descriptor index number, where $1 \leq$ `D` $\leq$ `ndescriptors`. (Input)

`name` Name of the `Descriptor_t` node. (Input for `cg_descriptor_write`; *output* for `cg_descriptor_read`)

`text` Description held in the `Descriptor_t` node. (Input for `cg_descriptor_write`; *output* for `cg_descriptor_read`)

`size` Size of the descriptor data (Fortran interface only). (*Output*)

`ier` Error status. (*Output*)

Note that with `cg_descriptor_read` the memory for the descriptor character string, `text`, will be allocated by the Mid-Level Library. The application code is responsible for releasing this memory when it is no longer needed by calling `cg_free(text)`, described in Section 10.6.

7.2 Ordinal Value

Node: `Ordinal_t`

Functions	Modes
`ier = cg_ordinal_write(int Ordinal);`	`- w m`
`ier = cg_ordinal_read(int *Ordinal);`	`r - m`
`call cg_ordinal_write_f(Ordinal, ier)`	`- w m`
`call cg_ordinal_read_f(Ordinal, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`Ordinal`	Any integer value. (Input for `cg_ordinal_write`; *output* for `cg_ordinal_read`)
`ier`	Error status. (*Output*)

8 Physical Data

8.1 Data Arrays

Node: `DataArray_t`

Functions	Modes
`ier = cg_array_write(char *ArrayName, DataType_t DataType, int DataDimension, cgsize_t *DimensionVector, void *Data);`	`- w m`
`ier = cg_narrays(int *narrays);`	`r - m`
`ier = cg_array_info(int A, char *ArrayName, DataType_t *DataType, int *DataDimension, cgsize_t *DimensionVector);`	`r - m`
`ier = cg_array_read(int A, void *Data);`	`r - m`
`ier = cg_array_read_as(int A, DataType_t DataType, void *Data);`	`r - m`
`call cg_array_write_f(ArrayName, DataType, DataDimension, DimensionVector, Data, ier)`	`- w m`
`call cg_narrays_f(narrays, ier)`	`r - m`
`call cg_array_info_f(A, ArrayName, DataType, DataDimension, DimensionVector, ier)`	`r - m`
`call cg_array_read_f(A, Data, ier)`	`r - m`
`call cg_array_read_as_f(A, DataType, Data, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`narrays`	Number of `DataArray_t` nodes under the current node. (*Output*)
`A`	Data array index, where $1 \leq$ `A` $\leq$ `narrays`. (Input)
`ArrayName`	Name of the `DataArray_t` node. (Input for `cg_array_write`; *output* for `cg_array_info`)
`DataType`	Type of data held in the `DataArray_t` node. The admissible types are `Integer`, `LongInteger`, `RealSingle`, `RealDouble`, and `Character`. (Input for `cg_array_write`, `cg_array_read_as`; *output* for `cg_array_info`)
`DataDimension`	Number of dimensions. (Input for `cg_array_write`; *output* for `cg_array_info`)
`DimensionVector`	Number of data elements in each dimension. (Input for `cg_array_write`; *output* for `cg_array_info`)
`Data`	The data array. (Input for `cg_array_write`; *output* for `cg_array_read`, `cg_array_read_as`)
`ier`	Error status. (*Output*)

8.2 Data Class

Node: `DataClass_t`

Functions	Modes
`ier = cg_dataclass_write(DataClass_t dataclass);`	`- w m`
`ier = cg_dataclass_read(DataClass_t *dataclass);`	`r - m`
`call cg_dataclass_write_f(dataclass, ier)`	`- w m`
`call cg_dataclass_read_f(dataclass, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`dataclass`	Data class for the nodes at this level. See below for the data classes currently supported in CGNS. (Input for `cg_dataclass_write`; *output* for `cg_dataclass_read`)
`ier`	Error status. (*Output*)

The data classes currently supported in CGNS are:

`Dimensional`	Regular dimensional data.
`NormalizedByDimensional`	Nondimensional data that is normalized by dimensional reference quantities.
`NormalizedByUnknownDimensional`	All fields and reference data are nondimensional.
`NondimensionalParameter`	Nondimensional parameters such as Mach number and lift coefficient.
`DimensionlessConstant`	Constant such as π.

These classes are declared within typedef `DataClass_t` in *cgnslib.h*, and as parameters in *cgnslib_f.h*.

8.3 Data Conversion Factors

Node: `DataConversion_t`

Functions	Modes
`ier = cg_conversion_write(DataType_t DataType, void *ConversionFactors);`	`- w m`
`ier = cg_conversion_info(DataType_t *DataType);`	`r - m`
`ier = cg_conversion_read(void *ConversionFactors);`	`r - m`
`call cg_conversion_write_f(DataType, ConversionFactors, ier)`	`- w m`
`call cg_conversion_info_f(DataType, ier)`	`r - m`
`call cg_conversion_read_f(ConversionFactors, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`DataType`	Data type in which the conversion factors are recorded. Admissible data types for conversion factors are `RealSingle` and `RealDouble`. (Input for `cg_conversion_write`; *output* for `cg_conversion_info`)
`ConversionFactors`	Two-element array containing the scaling and offset factors. (Input for `cg_conversion_write`; *output* for `cg_conversion_read`)
`ier`	Error status. (*Output*)

The `DataConversion_t` data structure contains factors to convert the nondimensional data to "raw" dimensional data. The scaling and offset factors are contained in the two-element array `ConversionFactors`. In pseudo-Fortran, the conversion process is as follows:

```
ConversionScale  = ConversionFactors(1)
ConversionOffset = ConversionFactors(2)
Data(raw) = Data(nondimensional)*ConversionScale + ConversionOffset
```

8.4 Dimensional Units

Node: `DimensionalUnits_t`

Functions	Modes
ier = cg_units_write(MassUnits_t mass, LengthUnits_t length, TimeUnits_t time, TemperatureUnits_t temperature, AngleUnits_t angle);	- w m
ier = cg_unitsfull_write(MassUnits_t mass, LengthUnits_t length, TimeUnits_t time, TemperatureUnits_t temperature, AngleUnits_t angle, ElectricCurrentUnits_t current, SubstanceAmountUnits_t amount, LuminousIntensityUnits_t intensity);	- w m
ier = cg_nunits(*int *nunits*);	r - m
ier = cg_units_read(*MassUnits_t *mass*, *LengthUnits_t *length*, *TimeUnits_t *time*, *TemperatureUnits_t *temperature*, *AngleUnits_t *angle*);	r - m
ier = cg_unitsfull_read(*MassUnits_t *mass*, *LengthUnits_t *length*, *TimeUnits_t *time*, *TemperatureUnits_t *temperature*, *AngleUnits_t *angle*, *ElectricCurrentUnits_t *current*, *SubstanceAmountUnits_t *amount*, *LuminousIntensityUnits_t *intensity*);	r - m
call cg_units_write_f(mass, length, time, temperature, angle, *ier*)	- w m
call cg_unitsfull_write_f(mass, length, time, temperature, angle, current, amount, intensity, *ier*)	- w m
call cg_nunits_f(*int *nunits*)	r - m
call cg_units_read(*mass*, *length*, *time*, *temperature*, *angle*, *ier*)	r - m
call cg_unitsfull_read_f(*mass*, *length*, *time*, *temperature*, *angle*, *current*, *amount*, *intensity*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`mass` Mass units. Admissible values are `CG_Null`, `CG_UserDefined`, `Kilogram`, `Gram`, `Slug`, and `PoundMass`. (Input for `cg_units_write`, `cg_unitsfull_write`; *output* for `cg_units_read`, `cg_unitsfull_read`)

`length` Length units. Admissible values are `CG_Null`, `CG_UserDefined`, `Meter`, `Centimeter`, `Millimeter`, `Foot`, and `Inch`. (Input for `cg_units_write`, `cg_unitsfull_write`; *output* for `cg_units_read`, `cg_unitsfull_read`)

`time` Time units. Admissible values are `CG_Null`, `CG_UserDefined`, and `Second`. (Input for `cg_units_write`, `cg_unitsfull_write`; *output* for `cg_units_read`, `cg_unitsfull_read`)

`temperature` Temperature units. Admissible values are `CG_Null`, `CG_UserDefined`, `Kelvin`, `Celsius`, `Rankine`, and `Fahrenheit`. (Input for `cg_units_write`, `cg_unitsfull_write`; *output* for `cg_units_read`, `cg_unitsfull_read`)

`angle` Angle units. Admissible values are `CG_Null`, `CG_UserDefined`, `Degree`, and

Radian. (Input for cg_units_write, cg_unitsfull_write; *output* for cg_units_read, cg_unitsfull_read)

current
: Electric current units. Admissible values are CG_Null, CG_UserDefined, Ampere, Abampere, Statampere, Edison, and auCurrent. (Input for cg_unitsfull_write; *output* for cg_unitsfull_read)

amount
: Substance amount units. Admissible values are CG_Null, CG_UserDefined, Mole, Entities, StandardCubicFoot, and StandardCubicMeter. (Input for cg_unitsfull_write; *output* for cg_unitsfull_read)

intensity
: Luminous intensity units. Admissible values are CG_Null, CG_UserDefined, Candela, Candle, Carcel, Hefner, and Violle. (Input for cg_unitsfull_write; *output* for cg_unitsfull_read)

nunits
: Number of units used in the file (i.e., either 5 or 8). (*Output*)

ier
: Error status. (*Output*)

The supported units are declared within typedefs in *cgnslib.h* and as parameters in *cgnslib_f.h*.

When reading units data, either cg_units_read or cg_unitsfull_read may be used, regardless of the number of units used in the file. If cg_unitsfull_read is used, but only five units are used in the file, the returned values of current, amount, and intensity will be CG_Null.

8.5 Dimensional Exponents

Node: `DimensionalExponents_t`

Functions	Modes
`ier = cg_exponents_write(DataType_t DataType, void *exponents);`	`- w m`
`ier = cg_expfull_write(DataType_t DataType, void *exponents);`	`- w m`
`ier = cg_nexponents(int *nexponents);`	`r - m`
`ier = cg_exponents_info(DataType_t *DataType);`	`r - m`
`ier = cg_exponents_read(void *exponents);`	`r - m`
`ier = cg_expfull_read(void *exponents);`	`r - m`
`call cg_exponents_write_f(DataType, exponents, ier)`	`- w m`
`call cg_expfull_write_f(DataType, exponents, ier)`	`- w m`
`call cg_nexponents_f(nexponents, ier )`	`r - m`
`call cg_exponents_info_f(DataType, ier )`	`r - m`
`call cg_exponents_read_f(exponents, ier)`	`r - m`
`call cg_expfull_read_f(exponents, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`DataType` Data type in which the exponents are recorded. Admissible data types for the exponents are `RealSingle` and `RealDouble`. (Input for `cg_exponents_write`; *output* for `cg_exponents_info`)

`exponents` Exponents for the dimensional units for mass, length, time, temperature, angle, electric current, substance amount, and luminous intensity, in that order. (Input for `cg_exponents_write`, `cg_expfull_write`; *output* for `cg_exponents_read`, `cg_expfull_read`)

`nexponents` Number of exponents used in the file (i.e., either 5 or 8). (*Output*)

`ier` Error status. (*Output*)

When reading exponent data, either `cg_exponents_read` or `cg_expfull_read` may be used, regardless of the number of exponents used in the file. If `cg_exponents_read` is used, but all eight exponents are used in the file, only the first five exponents are returned. If `cg_expfull_read` is used, but only five exponents are used in the file, the returned values of the exponents for electric current, substance amount, and luminous intensity will be zero.

9 Location and Position

9.1 Grid Location

Node: GridLocation_t

Functions	Modes
ier = cg_gridlocation_write(GridLocation_t GridLocation);	- w m
ier = cg_gridlocation_read(*GridLocation_t *GridLocation*);	r - m
call cg_gridlocation_write_f(GridLocation, *ier*)	- w m
call cg_gridlocation_read_f(*GridLocation*, *ier*)	r - m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

GridLocation — Location in the grid. The admissible locations are CG_Null, CG_UserDefined, Vertex, CellCenter, FaceCenter, IFaceCenter, JFaceCenter, KFaceCenter, and EdgeCenter. (Input for cg_gridlocation_write; *output* for cg_gridlocation_read)

ier — Error status. (*Output*)

9.2 Point Sets

Node: IndexArray_t, IndexRange_t

Functions	Modes
ier = cg_ptset_write(PointSetType_t *ptset_type, cgsize_t npnts, cgsize_t *pnts);	- w m
ier = cg_ptset_info(*PointSetType_t *ptset_type*, *cgsize_t *npnts*);	r - m
ier = cg_ptset_read(*cgsize_t *pnts*);	r - m
call cg_ptset_write_f(ptset_type, npnts, pnts, *ier*)	- w m
call cg_ptset_info_f(*ptset_type*, *npnts*, *ier*)	r - m
call cg_ptset_read_f(*pnts*, *ier*)	r - m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

ptset_type — The point set type; either PointRange for a range of points or cells, or PointList for a list of discrete points or cells. (Input for cg_ptset_write; *output* for cg_ptset_info)

npnts — The number of points or cells in the point set. For a point set type of PointRange, npnts is always two. For a point set type of PointList, npnts is the number of points or cells in the list. (Input for `cg_ptset_write`; *output* for `cg_ptset_info`)

`pnts` The array of point or cell indices defining the point set. There should be `npnts` values, each of dimension `IndexDimension` (i.e., 1 for unstructured grids, and 2 or 3 for structured grids with 2-D or 3-D elements, respectively). (Input for `cg_ptset_write`; *output* for `cg_ptset_read`)

`ier` Error status. (*Output*)

These functions may be used to write and read point set data (i.e., an `IndexArray_t` node named `PointList`, or an `IndexRange_t` node named `PointRange`). They are only applicable at nodes that are descendents of a `Zone_t` node.

9.3 Rind Layers

Node: `Rind_t`

Functions	Modes
`ier = cg_rind_write(int *RindData);`	`- w m`
`ier = cg_rind_read(int *RindData);`	`r - m`
`call cg_rind_write_f(RindData, ier)`	`- w m`
`call cg_rind_read_f(RindData, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`RindData` Number of rind layers for each computational direction (structured grid) or number of rind points or elements (unstructured grid). (Input for `cg_rind_write`; *output* for `cg_rind_read`)

`ier` Error status. (*Output*)

When writing rind data for elements, `cg_section_write` must be called first (see Section 11.2), followed by `cg_goto` (Section 4) to access the `Elements_t` node, and then `cg_rind_write`.

10 Auxiliary Data

10.1 Reference State

Node: `ReferenceState_t`

Functions	Modes
`ier = cg_state_write(char *StateDescription);`	`- w m`
`ier = cg_state_read(char **StateDescription);`	`r - m`
`call cg_state_write_f(StateDescription, ier)`	`- w m`
`call cg_state_size_f(Size, ier)`	`r - m`
`call cg_state_read_f(StateDescription, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`StateDescription`	Text description of reference state. (Input for `cg_state_write`; *output* for `cg_state_read`)
`Size`	Number of characters in the `StateDescription` string (Fortran interface only). (*Output*)
`ier`	Error status. (*Output*)

The function `cg_state_write` creates the `ReferenceState_t` node and must be called even if `StateDescription` is undefined (i.e., a blank string). The descriptors, data arrays, data class, and dimensional units characterizing the `ReferenceState_t` data structure may be added to this data structure after its creation.

The function `cg_state_read` reads the `StateDescription` of the local `ReferenceState_t` node. If `StateDescription` is undefined in the CGNS database, this function returns a null string. If `StateDescription` exists, the library will allocate the space to store the description string, and return the description string to the application. It is the responsibility of the application to free this space when it is no longer needed by a call to `cg_free(StateDescription)`, described in Section 10.6.

10.2 Gravity

Node: `Gravity_t`

Functions	Modes
`ier = cg_gravity_write(int fn, int B, float *GravityVector);`	`- w m`
`ier = cg_gravity_read(int fn, int B, float *GravityVector);`	`r - m`
`call cg_gravity_write_f(fn, B, GravityVector, ier)`	`- w m`
`call cg_gravity_read_f(fn, B, GravityVector, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`fn`	CGNS file index number. (*Input*)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (*Input*)
`GravityVector`	Components of the gravity vector. The number of components must equal `PhysicalDimension`. (In Fortran, this is an array of Real*4 values.) (Input for `cg_gravity_write`; *output* for `cg_gravity_read`)
`ier`	Error status. (*Output*)

10.3 Convergence History

Node: `ConvergenceHistory_t`

Functions	Modes
`ier = cg_convergence_write(int niterations, char *NormDefinitions);`	`- w m`
`ier = cg_convergence_read(int *niterations, char **NormDefinitions);`	`r - m`
`call cg_convergence_write_f(niterations, NormDefinitions, ier)`	`- w m`
`call cg_convergence_read_f(niterations, NormDefinitions, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`niterations`	Number of iterations for which convergence information is recorded. (Input for `cg_convergence_write`; *output* for `cg_convergence_read`)
`NormDefinitions`	Description of the convergence information recorded in the data arrays. (Input for `cg_convergence_write`; *output* for `cg_convergence_read`)
`ier`	Error status. (*Output*)

The function `cg_convergence_write` creates a `ConvergenceHistory_t` node. It must be the first one called when recording convergence history data. The `NormDefinitions` may be left undefined (i.e., a blank string). After creation of this node, the descriptors, data arrays, data class, and dimensional units characterizing the `ConvergenceHistory_t` data structure may be added.

The function `cg_convergence_read` reads a `ConvergenceHistory_t` node. If `NormDefinitions` is not defined in the CGNS database, this function returns a null string. If `NormDefinitions` exists, the library will allocate the space to store the description string, and return the description string to the application. It is the responsibility of the application to free this space when it is no longer needed by a call to `cg_free(NormDefinitions)`, described in Section 10.6.

10.4 Integral Data

Node: `IntegralData_t`

Functions	Modes
`ier = cg_integral_write(char *Name);`	`- w m`
`ier = cg_nintegrals(int *nintegrals);`	`r - m`
`ier = cg_integral_read(int Index, char *Name);`	`r - m`
`call cg_integral_write_f(Name, ier)`	`- w m`
`call cg_nintegrals_f(nintegrals, ier)`	`r - m`
`call cg_integral_read_f(Index, Name, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`Name` Name of the `IntegralData_t` data structure. (Input for `cg_integral_write`; *output* for `cg_integral_read`)

`nintegrals` Number of `IntegralData_t` nodes under current node. (*Output*)

`Index` Integral data index number, where $1 \leq$ `Index` $\leq$ `nintegrals`. (Input)

`ier` Error status. (*Output*)

10.5 User-Defined Data

Node: `UserDefinedData_t`

Functions	Modes
`ier = cg_user_data_write(char *Name);`	`- w m`
`ier = cg_nuser_data(int *nuserdata);`	`r - m`
`ier = cg_user_data_read(int Index, char *Name);`	`r - m`
`call cg_user_data_write_f(Name, ier)`	`- w m`
`call cg_nuser_data_f(nuserdata, ier)`	`r - m`
`call cg_user_data_read_f(Index, Name, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`nuserdata` Number of `UserDefinedData_t` nodes under current node. (*Output*)

`Name` Name of the `UserDefinedData_t` node. (Input for `cg_user_data_write`; *output* for `cg_user_data_read`)

`Index` User-defined data index number, where $1 \leq$ `Index` $\leq$ `nuserdata`. (Input)

`ier` Error status. (*Output*)

After accessing a particular `UserDefinedData_t` node using `cg_goto`, the Point Set functions described in Section 9.2 may be used to read or write point set information for the node. The function `cg_gridlocation_write` may also be used to specify the location of the data with respect to the grid (e.g., `Vertex` or `FaceCenter`).

Multiple levels of `UserDefinedData_t` nodes may be written and retrieved by positioning via `cg_goto`. E.g.,

```
ier = cg_goto(fn, B, "Zone_t", Z, "UserDefinedData_t", ud1,
              "UserDefinedData_t", ud2, "UserDefinedData_t", ud3, "end");
```

10.6 Freeing Memory

Functions	Modes
`ier = cg_free(void *data);`	`r w m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`data` Data allocated by the Mid-Level Library. (*Input*)

`ier` Error status. (*Output*)

This function does not affect the structure of a CGNS file; it is provided as a convenience to free memory allocated by the Mid-Level Library when using C. This isn't necessary in Fortran, and thus an equivalent Fortran function is not provided.

The functions that are used to allocate memory for return values are `cg_descriptor_read`, `cg_convergence_read`, `cg_geo_read`, `cg_link_read`, and `cg_state_read`. Each of these may allocate space to contain the data returned to the application. It is the responsibility of the application to free this data when it is no longer needed. Calling `cg_free` is identical to calling the standard C function `free`, however it is probably safer in that the memory is freed in the same module in which it is created, particularly when the Mid-Level Library is a shared library or DLL. The routine checks for NULL data and will return `CG_ERROR` in this case, otherwise it returns `CG_OK`.

11 Grid Specification

11.1 Zone Grid Coordinates

Node: `GridCoordinates_t`

`GridCoordinates_t` nodes are used to describe grids associated with a particular zone. The original grid must be described by a `GridCoordinates_t` node named `GridCoordinates`. Additional `GridCoordinates_t` nodes may be used, with user-defined names, to store grids at multiple time steps or iterations. In addition to the discussion of the `GridCoordinates_t` node in the SIDS and File Mapping manuals, see the discussion of the `ZoneIterativeData_t` and `ArbitraryGridMotion_t` nodes in the SIDS manual.

Functions	Modes
`ier = cg_grid_write(int fn, int B, int Z, char *GridCoordName, int *G);`	`- w m`
`ier = cg_ngrids(int fn, int B, int Z, int *ngrids);`	`- w m`
`ier = cg_grid_read(int fn, int B, int Z, int G, char *GridCoordName);`	`r - m`
`call cg_grid_write_f(fn, B, Z, GridCoordName, G, ier)`	`- w m`
`call cg_ngrids_f(fn, B, Z, ngrids, ier)`	`- w m`
`call cg_grid_read_f(fn, B, Z, G, GridCoordName, ier)`	`r - m`

Input/Output

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`G`	Grid index number, where $1 \leq$ `G` $\leq$ `ngrids`. (Input for `cg_grid_read`; *output* for `cg_grid_write`)
`ngrids`	Number of `GridCoordinates_t` nodes for zone `Z`. (*Output*)
`GridCoordinateName`	Name of the `GridCoordinates_t` node. Note that the name “`GridCoordinates`” is reserved for the original grid and must be the first `GridCoordinates_t` node to be defined. (Input for `cg_grid_write`; *output* for `cg_grid_read`)
`ier`	Error status. (*Output*)

The above functions are applicable to any `GridCoordinates_t` node.

Functions	Modes
`ier = cg_coord_write(int fn, int B, int Z, DataType_t datatype, char *coordname, void *coord_array, int *C);`	- w m
`ier = cg_coord_partial_write(int fn, int B, int Z, DataType_t datatype, char *coordname, cgsize_t *range_min, cgsize_t *range_max, void *coord_array, int *C);`	- w m
`ier = cg_ncoords(int fn, int B, int Z, int *ncoords);`	r - m
`ier = cg_coord_info(int fn, int B, int Z, int C, DataType_t *datatype, char *coordname);`	r - m
`ier = cg_coord_read(int fn, int B, int Z, char *coordname, DataType_t datatype, cgsize_t *range_min, cgsize_t *range_max, void *coord_array);`	r - m
`call cg_coord_write_f(fn, B, Z, datatype, coordname, coord_array, C, ier)`	- w m
`call cg_coord_partial_write_f(fn, B, Z, datatype, coordname, range_min, range_max, coord_array, C, ier)`	- w m
`call cg_ncoords_f(fn, B, Z, ncoords, ier)`	r - m
`call cg_coord_info_f(fn, B, Z, C, datatype, coordname, ier)`	r - m
`call cg_coord_read_f(fn, B, Z, coordname, datatype, range_min, range_max, coord_array, ier)`	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where 1 ≤ `B` ≤ `nbases`. (Input)
`Z`	Zone index number, where 1 ≤ `Z` ≤ `nzones`. (Input)
`C`	Coordinate array index number, where 1 ≤ `C` ≤ `ncoords`. (Input for `cg_coord_info`; *output* for `cg_coord_write`)
`ncoords`	Number of coordinate arrays for zone `Z`. (*Output*)
`datatype`	Data type in which the coordinate array is written. Admissible data types for a coordinate array are `RealSingle` and `RealDouble`. (Input for `cg_coord_write`, `cg_coord_partial_write`, `cg_coord_read`; *output* for `cg_coord_info`)
`coordname`	Name of the coordinate array. It is strongly advised to use the SIDS nomenclature conventions when naming the coordinate arrays to insure file compatibility. (Input for `cg_coord_write`, `cg_coord_partial_write`, `cg_coord_read`; *output* for `cg_coord_info`)
`range_min`	Lower range index (eg., `imin, jmin, kmin`). (Input)
`range_max`	Upper range index (eg., `imax, jmax, kmax`). (Input)

`coord_array`	Array of coordinate values for the range prescribed. (Input for `cg_coord_write`; `cg_coord_partial_write`, *output* for `cg_coord_read`)
`ier`	Error status. (*Output*)

The above functions are applicable *only* to the `GridCoordinates_t` node named `GridCoordinates`, used for the original grid in a zone. Coordinates for additional `GridCoordinates_t` nodes in a zone must be read and written using the `cg_array_xxx` functions described in Section 8.1.

When writing, the function `cg_coord_write` will automatically write the full range of coordinates (i.e., the entire `coord_array`). The function `cg_coord_partial_write` may be used to write only a subset of `coord_array`. When using the partial write, any existing data as defined by `range_min` and `range_max` will be overwritten by the new values. All other values will not be affected.

The function `cg_coord_read` returns the coordinate array `coord_array`, for the range prescribed by `range_min` and `range_max`. The array is returned to the application in the data type requested in `datatype`. This data type does not need to be the same as the one in which the coordinates are stored in the file. A coordinate array stored as double precision in the CGNS file can be returned to the application as single precision, or vice versa.

In Fortran, when using `cg_coord_read_f` to read 2D or 3D coordinates, the extent of each dimension of `coord_array` must be consistent with the requested range. When reading a 1D solution, the declared size can be larger than the requested range. For example, for a 2D zone with 100 $\times$ 50 vertices, if `range_min` and `range_max` are set to (11,11) and (20,20) to read a subset of the coordinates, then `coord_array` must be dimensioned (10,10). If `coord_array` is declared larger (e.g., (100,50)) the indices for the returned coordinates will be wrong.

11.2 Element Connectivity

Node: `Elements_t`

Functions	Modes
ier = cg_section_write(int fn, int B, int Z, char *ElementSectionName, ElementType_t type, cgsize_t start, cgsize_t end, int nbndry, cgsize_t *Elements, *int *S*);	- w m
ier = cg_section_partial_write(int fn, int B, int Z, char *ElementSectionName, ElementType_t type, cgsize_t start, cgsize_t end, int nbndry, *int *S*);	- w m
ier = cg_elements_partial_write(int fn, int B, int Z, int S, cgsize_t start, cgsize_t end, cgsize_t *Elements);	- w m
ier = cg_parent_data_write(int fn, int B, int Z, int S, cgsize_t *ParentData);	- w m
ier = cg_parent_data_partial_write(int fn, int B, int Z, int S, cgsize_t start, cgsize_t end, cgsize_t *ParentData);	- w m
ier = cg_nsections(int fn, int B, int Z, *int *nsections*);	r - m
ier = cg_section_read(int fn, int B, int Z, int S, *char *ElementSectionName*, *ElementType_t *type*, *cgsize_t *start*, *cgsize_t *end*, *int *nbndry*, *int *parent_flag*);	r - m
ier = cg_ElementDataSize(int fn, int B, int Z, int S, *cgsize_t *ElementDataSize*);	r - m
ier = cg_ElementPartialSize(int fn, int B, int Z, int S, cgsize_t start, cgsize_t end, *cgsize_t *ElementDataSize*);	r - m
ier = cg_elements_read(int fn, int B, int Z, int S, *cgsize_t *Elements*, *cgsize_t *ParentData*);	r - m
ier = cg_elements_partial_read(int fn, int B, int Z, int S, cgsize_t start, cgsize_t end, *cgsize_t *Elements*, *cgsize_t *ParentData*);	r - m
ier = cg_npe(ElementType_t type, *int *npe*);	r w m

Functions	Modes
`call cg_section_write_f(fn, B, Z, ElementSectionName, type, start, end, nbndry, Elements, S, ier)`	- w m
`call cg_section_partial_write_f(fn, B, Z, ElementSectionName, type, start, end, nbndry, S, ier)`	- w m
`call cg_elements_partial_write_f(fn, B, Z, S, start, end, Elements, ier);`	- w m
`call cg_parent_data_write_f(fn, B, Z, S, ParentData, ier)`	- w m
`call cg_parent_data_partial_write_f(fn, B, Z, S, start, end, ParentData, ier)`	- w m
`call cg_nsections_f(fn, B, Z, nsections, ier)`	r - m
`call cg_section_read_f(fn, B, Z, S, ElementSectionName, type, start, end, nbndry, parent_flag, ier)`	r - m
`call cg_ElementDataSize_f(fn, B, Z, S, ElementDataSize, ier)`	r - m
`call cg_ElementPartialSize_f(fn, B, Z, S, start, end, ElementDataSize, ier)`	r - m
`call cg_elements_read_f(fn, B, Z, S, Elements, ParentData, ier)`	r - m
`call cg_elements_partial_read_f(fn, B, Z, S, start, end, Elements, ParentData, ier)`	r - m
`call cg_npe_f(type, npe, ier)`	r w m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`ElementSectionName`	Name of the `Elements_t` node. (Input for `cg_section_write`; *output* for `cg_section_read`)
`type`	Type of element. See the eligible types for `ElementType_t` in Section 2.6. (Input for `cg_section_write`, `cg_npe`; *output* for `cg_section_read`)
`start`	Index of first element in the section. (Input for `cg_section_write`, `cg_section_partial_write`, `cg_parent_data_partial_write`, `cg_ElementPartialSize`, `cg_elements_partial_write`; *output* for `cg_section_read`)
`end`	Index of last element in the section. (Input for `cg_section_write`, `cg_section_partial_write`, `cg_parent_data_partial_write`, `cg_ElementPartialSize`, `cg_elements_partial_write`; *output* for `cg_section_read`)

`nbndry`	Index of last boundary element in the section. Set to zero if the elements are unsorted. (Input for `cg_section_write`; *output* for `cg_section_read`)
`nsections`	Lower range index (eg., `imin, jmin, kmin`). (*Output*)
`S`	Element section index, where $1 \leq$ `S` $\leq$ `nsections`. (Input for `cg_parent_data_write`, `cg_section_read`, `cg_ElementDataSize`, `cg_elements_read`; *output* for `cg_section_write`)
`parent_flag`	Flag indicating if the parent data are defined. If the parent data exist, `parent_flag` is set to 1; otherwise it is set to 0. (*Output*)
`ElementDataSize`	Number of element connectivity data values. (*Output*)
`Elements`	Element connectivity data. (Input for `cg_section_write`; *output* for `cg_elements_read`)
`ParentData`	For boundary or interface elements, this array contains information on the cell(s) and cell face(s) sharing the element. If you do not need to read the `ParentData` when reading the `ElementData`, you may set the value to `NULL`. (*Output*)
`npe`	Number of nodes for an element of type `type`. (*Output*)
`ier`	Error status. (*Output*)

It is important to note that each element under a given `Zone_t` – including all cells, faces, edges, boundary elements, etc. – must have a unique element index number. The numbering should be consecutive (i.e., no gaps). This global numbering system insures that each and every element within a zone is uniquely identified by its number.

If the specified `Elements_t` node doesn't yet exist, it may be created using either `cg_section_write` or `cg_section_partial_write`. The function `cg_section_write` writes the full range as indicated by `start` and `end` and supplied by the element connectivity array `Elements`. The `cg_section_partial_write` function will create the element section data for the range start to end with the element data intialized to 0. To add elements to the section, use `cg_elements_partial_write` and parent data (it it exists) using `cg_parent_data_partial_write`. Both of these functions will replace the data for the range as indicated by `start` and `end` with the new values. In most cases, the data is not duplicated in the mid-level library, but written directly from the user data to disk. The exception to this is in the case of `MIXED`, `NGON_n`, and `NFACE_n` element sets. Since the size of the element connectivity array is not known directly, the MLL will keep a copy of the data in memory for the partial writes.

The function `cg_elements_read` returns all of the element connectivity and parent data. Specified subsets of the element connectivity and parent data may be read using `cg_elements_partial_read`.

11.3 Axisymmetry

Node: `Axisymmetry_t`

Functions	Modes
ier = cg_axisym_write(int fn, int B, float *ReferencePoint, float *AxisVector);	- w m
ier = cg_axisym_read(int fn, int B, *float *ReferencePoint*, *float *AxisVector*);	r - m
call cg_axisym_write_f(fn, B, ReferencePoint, AxisVector, *ier*)	- w m
call cg_axisym_read_f(fn, B, *ReferencePoint*, *AxisVector*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (*Input*)
`B`	Base index number, where 1 ≤ `B` ≤ `nbases`. (*Input*)
`ReferencePoint`	Origin used for defining the axis of rotation. (In Fortran, this is an array of Real*4 values.) (Input for `cg_axisym_write`; *output* for `cg_axisym_read`)
`AxisVector`	Direction cosines of the axis of rotation, through the reference point. (In Fortran, this is an array of Real*4 values.) (Input for `cg_axisym_write`; *output* for `cg_axisym_read`)
`ier`	Error status. (*Output*)

This node can only be used for a bi-dimensional model, i.e., `PhysicalDimension` must equal two.

11.4 Rotating Coordinates

Node: `RotatingCoordinates_t`

Functions	Modes
ier = cg_rotating_write(float *RotationRateVector, float *RotationCenter);	- w m
ier = cg_rotating_read(*float *RotationRateVector*, *float *RotationCenter*);	r - m
call cg_rotating_write_f(RotationRateVector, RotationCenter, *ier*)	- w m
call cg_rotating_read_f(*RotationRateVector*, *RotationCenter*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`RotationRateVector`	Components of the angular velocity of the grid about the center of rotation. (In Fortran, this is an array of Real*4 values.) (Input for `cg_rotating_write`; *output* for `cg_rotating_read`)

`RotationCenter`	Coordinates of the center of rotation. (In Fortran, this is an array of Real*4 values.) (Input for `cg_rotating_write`; *output* for `cg_rotating_read`)
`ier`	Error status. (*Output*)

12 Solution Data

12.1 Flow Solution

Node: `FlowSolution_t`

Functions	Modes
`ier = cg_sol_write(int fn, int B, int Z, char *solname,` `GridLocation_t location, int *S);`	`- w m`
`ier = cg_nsols(int fn, int B, int Z, int *nsols);`	`r - m`
`ier = cg_sol_info(int fn, int B, int Z, int S, char *solname,` `GridLocation_t *location);`	`r - m`

`ier = cg_sol_ptset_write(int fn, int B, int Z, char *solname,`
`GridLocation_t location, PointSetType_t ptset_type,`
`cgsize_t npnts, cgsize_t *pnts, int *S);`
	`- w m`

`ier = cg_sol_ptset_info(int fn, int B, int Z, int S,` `PointSetType_t *ptset_type, cgsize_t *npnts);`	`r - m`
`ier = cg_sol_ptset_read(int fn, int B, int Z, int S, cgsize_t *pnts);`	`r - m`
`ier = cg_sol_size(int fn, int B, int Z, int S, int *data_dim,` `cgsize_t *dim_vals);`	`r - m`
`call cg_sol_write_f(fn, B, Z, solname, location, S, ier)`	`- w m`
`call cg_nsols_f(fn, B, Z, nsols, ier)`	`r - m`
`call cg_sol_info_f(fn, B, Z, S, solname, location, ier)`	`r - m`
`call cg_sol_ptset_write_f(fn, B, Z, solname, location, ptset_type,` `npnts, pnts, S, ier)`	`- w m`
`call cg_sol_ptset_info_f(fn, B, Z, S, ptset_type, npnts, ier)`	`r - m`
`call cg_sol_ptset_read_f(fn, B, Z, S, pnts, ier)`	`r - m`
`call cg_sol_size_f(fn, B, Z, S, data_dim, dim_vals, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`S`	Flow solution index number, where $1 \leq$ `S` $\leq$ `nsols`. (Input for `cg_sol_info`; *output* for `cg_sol_write`)
`nsols`	Number of flow solutions for zone `Z`. (*Output*)
`solname`	Name of the flow solution. (Input for `cg_sol_write`; *output* for `cg_sol_info`)
`location`	Grid location where the solution is recorded. The current admissible locations are `Vertex`, `CellCenter`, `IFaceCenter`, `JFaceCenter`, and `KFaceCenter`. (Input for `cg_sol_write` and `cg_sol_ptset_write`; *output* for `cg_sol_info`)

`ptset_type`	Type of point set defining the interface in the current solution; either `PointRange` or `PointList`. (Input for `cg_sol_ptset_write`; *output* for `cg_sol_ptset_info`)
`npnts`	Number of points defining the interface in the current solution. For a `ptset_type` of `PointRange`, npnts is always two. For a `ptset_type` of `PointList`, npnts is the number of points in the list. (Input for `cg_sol_ptset_write`; *output* for `cg_sol_ptset_info`)
`pnts`	Array of points defining the interface in the current solution. (Input for `cg_sol_ptset_write`; *output* for `cg_sol_ptset_read`)
`data_dim`	Number of dimensions defining the solution data. If a point set has been defined, this will be 1, otherwise this will be the current zone index dimension. (*Output*)
`dim_vals`	The array of `data_dim` dimensions for the solution data. (*Output*)
`ier`	Error status. (*Output*)

The above functions are used to create, and get information about, `FlowSolution_t` nodes.

Solution data may be specified over the entire zone, as in previous versions of the library, using `cg_sol_write` or over a patch region of the zone using `cg_sol_ptset_write`. The two are mutually exclusive. In the first case, the size of the solution data is determined by the size of the zone, the grid location, and rind data (if any) as in previous versions. In the second case the solution data size is entirely determined by the `PointList/PointRange` patch. In order to determine which of the two forms in which the solution data was written, use `cg_sol_ptset_info`. If the solution is over the entire zone, `ptset_type` will be returned as `CG_Null` and `npnts` as 0. Otherwise, `ptset_type` will be either `PointList` or `PointRange`, and the number of points will be returned in `npnts`.

To assist in determining the size of the solution data, the function `cg_sol_size` has been added. For a solution defined over the full zone, `data_dim` returns the index dimension for the zone, and `dim_vals` specifies the dimensions of the data, corrected for the grid location type and rind data. If a point set patch has been specified, `data_dim` will be 1 and `dim_vals` will contain the total size of the patch.

Acceptable values of `GridLocation_t` are `Vertex` and `CellCenter`. If the base cell dimension is 2 or greater (surface or volume), then `EdgeCenter` is also allowed. For 3 dimensional bases, `FaceCenter`, and for structured zones, `IFaceCenter`, `JFaceCenter` and `KFaceCenter`, are also acceptable.

12.2 Flow Solution Data

Functions	Modes
`ier = cg_field_write(int fn, int B, int Z, int S, DataType_t datatype, char *fieldname, void *solution_array, int *F);`	- w m
`ier = cg_field_partial_write(int fn, int B, int Z, int S, DataType_t datatype, char *fieldname, cgsize_t *range_min, cgsize_t *range_max, void *solution_array, int *F);`	- w m
`ier = cg_nfields(int fn, int B, int Z, int S, int *nfields);`	r - m
`ier = cg_field_info(int fn, int B, int Z, int S, int F, DataType_t *datatype, char *fieldname);`	r - m
`ier = cg_field_read(int fn, int B, int Z, int S, char *fieldname, DataType_t datatype, cgsize_t *range_min, cgsize_t *range_max, void *solution_array);`	r - m
`call cg_field_write_f(fn, B, Z, S, datatype, fieldname, solution_array, F, ier)`	- w m
`call cg_field_partial_write_f(fn, B, Z, S, datatype, fieldname, range_min, range_max, solution_array, F, ier)`	- w m
`call cg_nfields_f(fn, B, Z, S, nfields, ier)`	r - m
`call cg_field_info_f(fn, B, Z, S, F, datatype, fieldname, ier)`	r - m
`call cg_field_read_f(fn, B, Z, S, fieldname, datatype, range_min, range_max, solution_array, ier)`	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`Z` Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`S` Flow solution index number, where $1 \leq$ `S` $\leq$ `nsols`. (Input)

`F` Solution array index number, where $1 \leq$ `F` $\leq$ `nfields`. (Input for `cg_field_info`; *output* for `cg_field_write`)

`nfields` Number of data arrays in flow solution `S`. (*Output*)

`datatype` Data type in which the solution array is written. Admissible data types for a solution array are `Integer`, `LongInteger`, `RealSingle`, and `RealDouble`. (Input for `cg_field_write`, `cg_field_read`; *output* for `cg_field_info`)

`fieldname` Name of the solution array. It is strongly advised to use the SIDS nomenclature conventions when naming the solution arrays to insure file compatibility. (Input for `cg_field_write`, `cg_field_read`; *output* for `cg_field_info`)

`range_min`	Lower range index (eg., `imin, jmin, kmin`). (Input)
`range_max`	Upper range index (eg., `imax, jmax, kmax`). (Input)
`solution_array`	Array of solution values for the range prescribed. (Input for `cg_field_write`; *output* for `cg_field_read`)
`ier`	Error status. (*Output*)

The above functions are used to read and write solution arrays stored below a `FlowSolution_t` node.

When writing, the function `cg_field_write` will automatically write the full range of the solution (i.e., the entire `solution_array`). The function `cg_field_partial_write` may be used to write only a subset of `solution_array`. When using the partial write, any existing data from `range_min` to `range_max` will be overwritten by the new values. All other values will not be affected.

The function `cg_field_read` returns the solution array `fieldname`, for the range prescribed by `range_min` and `range_max`. The array is returned to the application in the data type requested in `datatype`. This data type does not need to be the same as the one in which the data is stored in the file. A solution array stored as double precision in the CGNS file can be returned to the application as single precision, or vice versa.

In Fortran, when using `cg_field_read_f` to read a 2D or 3D solution, the extent of each dimension of `solution_array` must be consistent with the requested range. When reading a 1D solution, the declared size can be larger than the requested range. For example, for a 2D zone with 100×50 vertices, if `range_min` and `range_max` are set to (11,11) and (20,20) to read a subset of the solution, then `solution_array` must be dimensioned (10,10). If `solution_array` is declared larger (e.g., (100,50)) the indices for the returned array values will be wrong.

12.3 Discrete Data

Node: DiscreteData_t

Functions	Modes
`ier = cg_discrete_write(int fn, int B, int Z, char *DiscreteName, int *D);`	`- w m`
`ier = cg_ndiscrete(int fn, int B, int Z, int *ndiscrete);`	`r - m`
`ier = cg_discrete_read(int fn, int B, int Z, int D, char *DiscreteName);`	`r - m`
`ier = cg_discrete_ptset_write(int fn, int B, int Z, char *DiscreteName, GridLocation_t location, PointSetType_t ptset_type, cgsize_t npnts, cgsize_t *pnts, int *D);`	`- w m`
`ier = cg_discrete_ptset_info(int fn, int B, int Z, int D, PointSetType_t *ptset_type, cgsize_t *npnts);`	`r - m`
`ier = cg_discrete_ptset_read(int fn, int B, int Z, int D, cgsize_t *pnts);`	`r - m`
`ier = cg_discrete_size(int fn, int B, int Z, int D, int *data_dim, cgsize_t *dim_vals);`	`r - m`
`call cg_discrete_write_f(fn, B, Z, DiscreteName, D, ier)`	`- w m`
`call cg_ndiscrete_f(fn, B, Z, ndiscrete, ier)`	`r - m`
`call cg_discrete_read_f(fn, B, Z, D, DiscreteName, ier)`	`r - m`
`call cg_discrete_ptset_write_f(fn, B, Z, DiscreteName, location, ptset_type, npnts, pnts, D, ier)`	`- w m`
`call cg_discrete_ptset_info_f(fn, B, Z, D, ptset_type, npnts, ier)`	`r - m`
`call cg_discrete_ptset_read_f(fn, B, Z, D, pnts, ier)`	`r - m`
`call cg_discrete_size_f(fn, B, Z, D, data_dim, dim_vals, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`D`	Discrete data index number, where $1 \leq$ `D` $\leq$ `ndiscrete`. (Input for `cg_discrete_read`; *output* for `cg_discrete_write`)
`ndiscrete`	Number of `DiscreteData_t` data structures under zone `Z`. (*Output*)
`DiscreteName`	Name of `DiscreteData_t` data structure. (Input for `cg_discrete_write`; *output* for `cg_discrete_read`)
`location`	Grid location where the discrete data is recorded. The current admissible locations are `Vertex`, `CellCenter`, `IFaceCenter`, `JFaceCenter`, and

	KFaceCenter. (Input for cg_discrete_ptset_write; *output* for cg_sol_info)
ptset_type	Type of point set defining the interface for the discrete data; either PointRange or PointList. (Input for cg_sol_ptset_write; *output* for cg_sol_ptset_info)
npnts	Number of points defining the interface for the discrete data. For a ptset_type of PointRange, npnts is always two. For a ptset_type of PointList, npnts is the number of points in the list. (Input for cg_sol_ptset_write; *output* for cg_sol_ptset_info)
pnts	Array of points defining the interface for the discrete data. (Input for cg_sol_ptset_write; *output* for cg_sol_ptset_read)
data_dim	Number of dimensions defining the discrete data. If a point set has been defined, this will be 1, otherwise this will be the current zone index dimension. (*Output*)
dim_vals	The array of data_dim dimensions for the discrete data. (*Output*)
ier	Error status. (*Output*)

DiscreteData_t nodes are intended for the storage of fields of data not usually identified as part of the flow solution, such as fluxes or equation residuals.

The description for these functions is similar to the FlowSolution_t node (Section 12.1). To read and write the discrete data, use cg_goto (Section 4) to access the DiscreteData_t node, then cg_array_read and cg_array_write (Section 8.1).

12.4 Zone Subregions

Node: ZoneSubRegion_t

Functions	Modes
`ier = cg_nsubregs(int fn, int B, int Z, int *nsubregs);`	r - m
`ier = cg_subreg_info(int fn, int B, int Z, int S, char *regname, int *dimension, GridLocation_t *location, PointSetType_t *ptset_type, cgsize_t *npnts, int *bcname_len, int *bcname_len);`	r - m
`ier = cg_subreg_ptset_read(int fn, int B, int Z, int S, cgsize_t *pnts);`	r - m
`ier = cg_subreg_bcname_read(int fn, int B, int Z, int S, char *bcname);`	r - m
`ier = cg_subreg_gcname_read(int fn, int B, int Z, int S, char *gcname);`	r - m
`ier = cg_subreg_ptset_write(int fn, int B, int Z, const char *regname, int dimension, GridLocation_t location, PointSetType_t ptset_type, cgsize_t npnts, cgsize_t *pnts, int *S);`	- w m
`ier = cg_subreg_bcname_write(int fn, int B, int Z, const char *regname, int dimension, const char *bcname, int *S);`	- w m
`ier = cg_subreg_gcname_write(int fn, int B, int Z, const char *regname, int dimension, const char *gcname, int *S);`	- w m
`call cg_nsubregs_f(fn, B, Z, nsubregs, ier)`	r - m
`call cg_subreg_info_f(fn, B, Z, S, regname, dimension, location, ptset_type, npnts, bcname_len, bcname_len, ier)`	r - m
`call cg_subreg_ptset_read_f(fn, B, Z, S, pnts, ier)`	r - m
`call cg_subreg_bcname_read_f(fn, B, Z, S, bcname, ier)`	r - m
`call cg_subreg_gcname_read_f(fn, B, Z, S, gcname, ier)`	r - m
`call cg_subreg_ptset_write_f(fn, B, Z, regname, dimension, location, ptset_type, npnts, pnts, S, ier)`	- w m
`call cg_subreg_bcname_write_f(fn, B, Z, regname, dimension, bcname, S, ier)`	- w m
`call cg_subreg_gcname_write_f(fn, B, Z, regname, dimension, gcname, S, ier)`	- w m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

S	ZoneSubRegion index number, where $1 \leq$ S $\leq$ nsubregs. (Input for cg_subreg_info, cg_subreg_ptset_read, cg_subreg_bcname_read and cg_subreg_gcname_read; *output* for cg_subreg_ptset_write, cg_subreg_bcname_write and cg_subreg_gcname_write)
nsubregs	Number of ZoneSubRegion_t nodes under Zone Z. (*Output*)
regname	Name of the ZoneSubRegion_t node. (Input for cg_subreg_ptset_write, cg_subreg_bcname_write, and cg_subreg_gcname_write; *output* for cg_subreg_info)
dimension	Dimensionality of the subregion. 1 for lines, 2 for faces, 3 for volumes. (Input for cg_subreg_ptset_write, cg_subreg_bcname_write, and cg_subreg_gcname_write; *output* for cg_subreg_info)
location	Grid location used in the definition of the point set. The current admissible locations are Vertex and CellCenter. (Input for cg_subreg_ptset_write; *output* for cg_subreg_info)
ptset_type	Type of point set defining the interface for the subregion data; either PointRange or PointList. (Input for cg_subreg_ptset_write; *output* for cg_subreg_info)
npnts	Number of points defining the interface for the subregion data. For a ptset_type of PointRange, npnts is always two. For a ptset_type of PointList, npnts is the number of points in the list. (Input for cg_subreg_ptset_write; *output* for cg_subreg_info)
pnts	Array of points defining the interface for the subregion data. (Input for cg_subreg_ptset_write; *output* for cg_subreg_ptset_read)
bcname	The name of a BC_t node which defines the subregion. (Input for cg_subreg_bcname_write; *output* for cg_subreg_bcname_read)
gcname	The name of a GridConnectivity_t or GridConnectivity1to1_t node which defines the subregion. (Input for cg_subreg_gcname_write; *output* for cg_subreg_gcname_read)
bcname_len	String length of bcname. (*Output*)
gcname_len	String length of gcname. (*Output*)
ier	Error status. (*Output*)

These functions allow for the specification of Zone subregions. The subregion may be specified as either the name of an existing BC_t node (cg_subreg_bcname_write), an existing GridConnectivity_t or GridConnectivity1to1_t node (cg_subreg_gcname_write), or as a PointSet/PointRange (cg_subreg_ptset_write). These specifications are mutually exclusive. To determine the type of the subregion, use cg_subreg_info. If the subregion is a point set, then ptset_type will indicate the point set type (either PointList or PointRange) and npts will be set to the number of points to define the region. Otherwise, ptset_type will be set to CG_Null and npnts will be 0. In this case, one of bcname_len or gcname_len will be non-zero, indicating whether the subregion references a BC_t node (bcname_len non-zero) or GridConnectivity_t node (gcname_len non-zero).

13 Grid Connectivity

13.1 One-to-One Connectivity

Node: `GridConnectivity1to1_t`

Functions	Modes
`ier = cg_n1to1_global(int fn, int B, int *n1to1_global);`	`r - m`
`ier = cg_1to1_read_global(int fn, int B, char **connectname, char **zonename, char **donorname, cgsize_t **range, cgsize_t **donor_range, int **transform);`	`r - m`
`call cg_n1to1_global_f(fn, B, n1to1_global, ier)`	`r - m`
`call cg_1to1_read_global_f(fn, B, connectname, zonename, donorname, range, donor_range, transform, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`n1to1_global`	Total number of one-to-one interfaces in base `B`, stored under `GridConnectivity1to1_t` nodes. (I.e., this does not include one-to-one interfaces that may be stored under `GridConnectivity_t` nodes, used for generalized zone interfaces.) Note that the function `cg_n1to1` (described below) may be used to get the number of one-to-one interfaces in a specific zone. (*Output*)
`connectname`	Name of the interface. (*Output*)
`zonename`	Name of the first zone, for all one-to-one interfaces in base `B`. (*Output*)
`donorname`	Name of the second zone, for all one-to-one interfaces in base `B`. (*Output*)
`range`	Range of points for the first zone, for all one-to-one interfaces in base `B`. (*Output*)
`donor_range`	Range of points for the current zone, for all one-to-one interfaces in base `B`. (*Output*)
`transform`	Short hand notation for the transformation matrix defining the relative orientation of the two zones. This transformation is given for all one-to-one interfaces in base `B`. See the description of `GridConnectivity1to1_t` in the SIDS manual for details. (*Output*)
`ier`	Error status. (*Output*)

The above functions may be used to get information about all the one-to-one zone interfaces in a CGNS database.

Functions	Modes
ier = cg_1to1_write(int fn, int B, int Z, char *connectname, char *donorname, cgsize_t *range, cgsize_t *donor_range, int *transform, int *I);	- w m
ier = cg_n1to1(int fn, int B, int Z, int *n1to1);	r - m
ier = cg_1to1_read(int fn, int B, int Z, int I, char *connectname, char *donorname, cgsize_t *range, cgsize_t *donor_range, int *transform);	r - m
call cg_1to1_write_f(fn, B, Z, connectname, donorname, range, donor_range, transform, I, ier)	- w m
call cg_n1to1_f(fn, B, Z, n1to1, ier)	r - m
call cg_1to1_read_f(fn, B, Z, I, connectname, donorname, range, donor_range, transform, ier)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`Z` Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`I` Interface index number, where $1 \leq$ `I` $\leq$ `n1to1`. (Input for `cg_1to1_read`; *output* for `cg_1to1_write`)

`n1to1` Number of one-to-one interfaces in zone `Z`, stored under `GridConnectivity1to1_t` nodes. (I.e., this does not include one-to-one interfaces that may be stored under `GridConnectivity_t` nodes, used for generalized zone interfaces.) (*Output*)

`connectname` Name of the interface. (Input for `cg_1to1_write`; *output* for `cg_1to1_read`)

`donorname` Name of the zone interfacing with the current zone. (Input for `cg_1to1_write`; *output* for `cg_1to1_read`)

`range` Range of points for the current zone. (Input for `cg_1to1_write`; *output* for `cg_1to1_read`)

`donor_range` Range of points for the donor zone. (Input for `cg_1to1_write`; *output* for `cg_1to1_read`)

`transform` Short hand notation for the transformation matrix defining the relative orientation of the two zones. See the description of `GridConnectivity1to1_t` in the SIDS manual for details. (Input for `cg_1to1_write`; *output* for `cg_1to1_read`)

`ier` Error status. (*Output*)

The above functions are used to read and write one-to-one connectivity data for a specific zone.

13.2 Generalized Connectivity

Node: `GridConnectivity_t`

Functions	Modes
`ier = cg_conn_write(int fn, int B, int Z, char *connectname, GridLocation_t location, GridConnectivityType_t connect_type, PointSetType_t ptset_type, cgsize_t npnts, cgsize_t *pnts, char *donorname, ZoneType_t donor_zonetype, PointSetType_t donor_ptset_type, DataType_t donor_datatype, cgsize_t ndata_donor, cgsize_t *donor_data, int *I);`	`- w m`
`ier = cg_conn_write_short(int fn, int B, int Z, char *connectname, GridLocation_t location, GridConnectivityType_t connect_type, PointSetType_t ptset_type, cgsize_t npnts, cgsize_t *pnts, char *donorname, int *I);`	`- w m`
`ier = cg_nconns(int fn, int B, int Z, int *nconns);`	`r - m`
`ier = cg_conn_info(int fn, int B, int Z, int I, char *connectname, GridLocation_t *location, GridConnectivityType_t *connect_type, PointSetType_t *ptset_type, cgsize_t *npnts, char *donorname, ZoneType_t *donor_zonetype, PointSetType_t *donor_ptset_type, DataType_t *donor_datatype, cgsize_t *ndata_donor);`	`r - m`
`ier = cg_conn_read(int fn, int B, int Z, int I, cgsize_t *pnts, DataType_t donor_datatype, cgsize_t *donor_data);`	`r - m`
`ier = cg_conn_read_short(int fn, int B, int Z, int I, cgsize_t *pnts);`	`r - m`
`call cg_conn_write_f(fn, B, Z, connectname, location, connect_type, ptset_type, npnts, pnts, donorname, donor_zonetype, donor_ptset_type, donor_datatype, ndata_donor, donor_data, I, ier)`	`- w m`
`call cg_conn_write_short_f(fn, B, Z, connectname, location, connect_type, ptset_type, npnts, pnts, donorname, I, ier)`	`- w m`
`call cg_nconns_f(fn, B, Z, nconns, ier)`	`r - m`
`call cg_conn_info_f(fn, B, Z, I, connectname, location, connect_type, ptset_type, npnts, donorname, donor_zonetype, donor_ptset_type, donor_datatype, ndata_donor, ier)`	`r - m`
`call cg_conn_read_f(fn, B, Z, I, pnts, donor_datatype, donor_data, ier)`	`r - m`
`call cg_conn_read_short_f(fn, B, Z, I, pnts, ier)`	

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

Z	Zone index number, where $1 \le$ Z $\le$ nzones. (Input)
I	Discrete data index number, where $1 \le$ I $\le$ nconns. (Input for cg_conn_info, cg_conn_read; *output* for cg_conn_write, cg_conn_write_short)
nconns	Number of interfaces for zone Z. (*Output*)
connectname	Name of the interface. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_info)
location	Grid location used in the definition of the point set. The currently admissible locations are Vertex and CellCenter. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_info)
connect_type	Type of interface being defined. The admissible types are Overset, Abutting, and Abutting1to1. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_info)
ptset_type	Type of point set defining the interface in the current zone; either PointRange or PointList. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_info)
donor_ptset_type	Type of point set defining the interface in the donor zone; either PointListDonor or CellListDonor. (Input for cg_conn_write; *output* for cg_conn_info)
npnts	Number of points defining the interface in the current zone. For a ptset_type of PointRange, npnts is always two. For a ptset_type of PointList, npnts is the number of points in the PointList. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_info)
ndata_donor	Number of points or cells in the current zone. These are paired with points, cells, or fractions thereof in the donor zone. (Input for cg_conn_write; *output* for cg_conn_info)
donorname	Name of the zone interfacing with the current zone. (Input for cg_conn_write; *output* for cg_conn_info)
donor_datatype	Data type in which the donor points are stored in the file. As of Version 3.0, this value is ignored when writing, and on reading it will return either Integer or LongInteger depending on whether the file was written using 32 or 64-bit. The donor_datatype argument was left in these functions only for backward compatibility. The donot data is always read as cgsize_t. (Input for cg_conn_write, cg_conn_read; *output* for cg_conn_info)
pnts	Array of points defining the interface in the current zone. (Input for cg_conn_write, cg_conn_write_short; *output* for cg_conn_read)
donor_data	Array of donor points or cells corresponding to ndata_donor. Note that it is possible that the same donor point or cell may be used multiple times. (Input for cg_conn_write; *output* for cg_conn_read)
donor_zonetype	Type of the donor zone. The admissible types are Structured and Unstructured. (Input for cg_conn_write; *output* for cg_conn_info)

`ier` Error status. (*Output*)

Note that the interpolation factors stored in the `InterpolantsDonor` data array are accessed using the `cg_goto` and `cg_array_`*`xxx`* functions, described in Section 4 and Section 8.1, respectively.

13.3 Special Grid Connectivity Properties

Node: `GridConnectivityProperty_t`

Functions	Modes
ier = cg_conn_periodic_write(int fn, int B, int Z, int I, float *RotationCenter, float *RotationAngle, float *Translation);	- w m
ier = cg_conn_average_write(int fn, int B, int Z, int I, AverageInterfaceType_t AverageInterfaceType);	- w m
ier = cg_1to1_periodic_write(int fn, int B, int Z, int I, float *RotationCenter, float *RotationAngle, float *Translation);	- w m
ier = cg_1to1_average_write(int fn, int B, int Z, int I, AverageInterfaceType_t AverageInterfaceType);	- w m
ier = cg_conn_periodic_read(int fn, int B, int Z, int I, *float *RotationCenter*, *float *RotationAngle*, *float *Translation*);	r - m
ier = cg_conn_average_read(int fn, int B, int Z, int I, *AverageInterfaceType_t *AverageInterfaceType*);	r - m
ier = cg_1to1_periodic_read(int fn, int B, int Z, int I, *float *RotationCenter*, *float *RotationAngle*, *float *Translation*);	r - m
ier = cg_1to1_average_read(int fn, int B, int Z, int I, *AverageInterfaceType_t *AverageInterfaceType*);	r - m
call cg_conn_periodic_write_f(fn, B, Z, I, RotationCenter, RotationAngle, Translation, *ier*)	- w m
call cg_conn_average_write_f(fn, B, Z, I, AverageInterfaceType, *ier*)	- w m
call cg_1to1_periodic_write_f(fn, B, Z, I, RotationCenter, RotationAngle, Translation, *ier*)	- w m
call cg_1to1_average_write_f(fn, B, Z, I, AverageInterfaceType, *ier*)	- w m
call cg_conn_periodic_read_f(fn, B, Z, I, *RotationCenter*, *RotationAngle*, *Translation*, *ier*)	r - m
call cg_conn_average_read_f(fn, B, Z, I, *AverageInterfaceType*, *ier*)	r - m
call cg_1to1_periodic_read_f(fn, B, Z, I, *RotationCenter*, *RotationAngle*, *Translation*, *ier*)	r - m
call cg_1to1_average_read_f(fn, B, Z, I, *AverageInterfaceType*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`I`	Grid connectivity index number, where $1 \leq$ `I` $\leq$ `nconns` for the "`cg_conn`" functions, and $1 \leq$ `I` $\leq$ `n1to1` for the "`cg_1to1`" functions. (Input)
`RotationCenter`	An array of size `phys_dim` defining the coordinates of the origin for defining the rotation angle between the periodic interfaces. (`phys_dim` is the number of coordinates required to define a vector in the field.) (In Fortran, this is an array of Real*4 values.) (Input for `cg_conn_periodic_write`, `cg_1to1_periodic_write`; *output* for `cg_conn_periodic_read`, `cg_1to1_periodic_read`)
`RotationAngle`	An array of size `phys_dim` defining the rotation angle from the current interface to the connecting interface. If rotating about more than one axis, the rotation is performed first about the x-axis, then the y-axis, then the z-axis. (In Fortran, this is an array of Real*4 values.) (Input for `cg_conn_periodic_write`, `cg_1to1_periodic_write`; *output* for `cg_conn_periodic_read`, `cg_1to1_periodic_read`)
`Translation`	An array of size `phys_dim` defining the translation from the current interface to the connecting interface. (In Fortran, this is an array of Real*4 values.) (Input for `cg_conn_periodic_write`, `cg_1to1_periodic_write`; *output* for `cg_conn_periodic_read`, `cg_1to1_periodic_read`)
`AverageInterfaceType`	The type of averaging to be done. Valid types are `CG_Null`, `CG_UserDefined`, `AverageAll`, `AverageCircumferential`, `AverageRadial`, `AverageI`, `AverageJ`, and `AverageK`. (Input for `cg_conn_average_write`, `cg_1to1_average_write`; *output* for `cg_conn_average_read`, `cg_1to1_average_read`)
`ier`	Error status. (*Output*)

These functions may be used to store special grid connectivity properties. The "`cg_conn`" functions apply to generalized grid connectivity nodes (i.e., `GridConnectivity_t`), and the "`cg_1to1`" functions apply to 1-to-1 grid connectivity nodes (i.e., `GridConnectivity1to1_t`).

The "`write`" functions will create the `GridConnectivityProperty_t` node if it doesn't already exist, then add the appropriate connectivity property. Multiple connectivity properties may be recorded under the same `GridConnectivityProperty_t` node.

The "`read`" functions will return with `ier` = 2 = `CG_NODE_NOT_FOUND` if the requested connectivity property, or the `GridConnectivityProperty_t` node itself, doesn't exist.

13.4 Overset Holes

Node: `OversetHoles_t`

Functions	Modes

`ier = cg_hole_write(int fn, int B, int Z, char *holename,`
`GridLocation_t location, PointSetType_t ptset_type, int nptsets,`
`cgsize_t npnts, cgsize_t *pnts, int *I);`
	`- w m`

`ier = cg_nholes(int fn, int B, int Z, int *nholes);`	`r - m`

`ier = cg_hole_info(int fn, int B, int Z, int I, char *holename,`
`GridLocation_t *location, PointSetType_t *ptset_type,`
`int *nptsets, cgsize_t *npnts);`
	`r - m`

`ier = cg_hole_read(int fn, int B, int Z, int I, cgsize_t *pnts);`	`r - m`
`call cg_hole_write_f(fn, B, Z, holename, location, ptset_type,` `nptsets, npnts, pnts, I, ier)`	`- w m`
`call cg_nholes_f(fn, B, Z, nholes, ier)`	`r - m`
`call cg_hole_info_f(fn, B, Z, I, holename, location, ptset_type,` `nptsets, npnts, ier)`	`r - m`
`call cg_hole_read_f(fn, B, Z, I, pnts, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`Z` Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`I` Overset hole index number, where $1 \leq$ `I` $\leq$ `nholes`. (Input for `cg_hole_info`, `cg_hole_read`; *output* for `cg_hole_write`)

`nholes` Number of overset holes in zone `Z`. (*Output*)

`holename` Name of the overset hole. (Input for `cg_hole_write`; *output* for `cg_hole_info`)

`location` Grid location used in the definition of the point set. The currently admissible locations are `Vertex` and `CellCenter`. (Input for `cg_hole_write`; *output* for `cg_hole_info`)

`ptset_type` The extent of the overset hole may be defined using a range of points or cells, or using a discrete list of all points or cells in the overset hole. If a range of points or cells is used, `ptset_type` is set to `PointRange`. When a discrete list of points or cells is used, `ptset_type` equals `PointList`. (Input for `cg_hole_write`; *output* for `cg_hole_info`)

`nptsets` Number of point sets used to define the hole. If `ptset_type` is `PointRange`, several point sets may be used. If `ptset_type` is `PointList`, only one point set is allowed. (Input for `cg_hole_write`; *output* for `cg_hole_info`) npnts — Number of points (or cells) in the point set. For a ptset_type of PointRange, npnts is always two. For a ptset_type of PointList, npnts is the number of points or cells in the PointList. (Input for cg_hole_write; *output* for cg_hole_info)

pnts — Array of points or cells in the point set. (Input for cg_hole_write; *output* for cg_hole_read)

ier — Error status. (*Output*)

14 Boundary Conditions

14.1 Boundary Condition Type and Location

Node: BC_t

Functions	Modes
`ier = cg_boco_write(int fn, int B, int Z, char *boconame, BCType_t bocotype, PointSetType_t ptset_type, cgsize_t npnts, cgsize_t *pnts, int *BC);`	- w m
`ier = cg_boco_normal_write(int fn, int B, int Z, int BC, int *NormalIndex, int NormalListFlag, DataType_t NormalDataType, void *NormalList);`	- w m
`ier = cg_boco_gridlocation_write(int fn, int B, int Z, int BC, GridLocation_t location);`	- w m
`ier = cg_nbocos(int fn, int B, int Z, int *nbocos);`	r - m
`ier = cg_boco_info(int fn, int B, int Z, int BC, char *boconame, BCType_t *bocotype, PointSetType_t *ptset_type, cgsize_t *npnts, int *NormalIndex, cgsize_t *NormalListFlag, DataType_t *NormalDataType, int *ndataset);`	r - m
`ier = cg_boco_read(int fn, int B, int Z, int BC, cgsize_t *pnts, void *NormalList);`	r - m
`ier = cg_boco_gridlocation_read(int fn, int B, int Z, int BC, GridLocation_t *location);`	r - m
`call cg_boco_write_f(fn, B, Z, boconame, bocotype, ptset_type, npnts, pnts, BC, ier)`	- w m
`call cg_boco_normal_write_f(fn, B, Z, BC, NormalIndex, NormalListFlag, NormalDataType, NormalList, ier)`	- w m
`call cg_boco_gridlocation_write_f(fn, B, Z, BC, location, ier);`	- w m
`call cg_nbocos_f(fn, B, Z, nbocos, ier)`	r - m
`call cg_boco_info_f(fn, B, Z, BC, boconame, bocotype, ptset_type, npnts, NormalIndex, NormalListFlag, NormalDataType, ndataset, ier)`	r - m
`call cg_boco_read_f(fn, B, Z, BC, pnts, NormalList, ier)`	r - m
`call cg_boco_gridlocation_read_f(fn, B, Z, BC, location, ier);`	r - m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

BC	Boundary condition index number, where $1 \leq$ BC $\leq$ nbocos. (Input for cg_boco_normal_write, cg_boco_info, cg_boco_read; *output* for cg_boco_write)
nbocos	Number of boundary conditions in zone Z. (*Output*)
boconame	Name of the boundary condition. (Input for cg_boco_write; *output* for cg_boco_info)
bocotype	Type of boundary condition defined. See the eligible types for BCType_t in Section 2.6. Note that if bocotype is FamilySpecified the boundary condition type is being specified for the family to which the boundary belongs. The boundary condition type for the family may be read and written using cg_fambc_read and cg_fambc_write, as described in Section 16.3. (Input for cg_boco_write; *output* for cg_boco_info)
ptset_type	The extent of the boundary condition may be defined using a range of points or elements using PointRange, or using a discrete list of all points or elements at which the boundary condition is applied using PointList. When the boundary condition is to be applied anywhere other than points, then GridLocation_t under the BC_t node must be used to indicate this. The value of GridLocation_t may be read or written by cg_boco_gridlocation_read and cg_boco_gridlocation_write. As in previous versions of the library, this may also be done by first using cg_goto (Section 4) to access the BC_t node, then using cg_gridlocation_read or cg_gridlocation_write (Section 9.1). (Input for cg_boco_write; *output* for cg_boco_info)
npnts	Number of points or elements defining the boundary condition region. For a ptset_type of PointRange, npnts is always two. For a ptset_type of PointList, npnts is the number of points or elements in the list. (Input for cg_boco_write; *output* for cg_boco_info)
pnts	Array of point or element indices defining the boundary condition region. There should be npnts values, each of dimension IndexDimension (i.e., 1 for unstructured grids, and 2 or 3 for structured grids with 2-D or 3-D elements, respectively). (Input for cg_boco_write; *output* for cg_boco_read)
NormalIndex	Index vector indicating the computational coordinate direction of the boundary condition patch normal. (Input for cg_boco_normal_write; *output* for cg_boco_info)
NormalListFlag	For cg_boco_normal_write, NormalListFlag is a flag indicating if the normals are defined in NormalList; 1 if they are defined, 0 if they're not. For cg_boco_info, if the normals are defined in NormalList, NormalListFlag is the number of points in the patch times phys_dim, the number of coordinates required to define a vector in the field. If the normals are not defined in NormalList, NormalListFlag is 0. (Input for cg_boco_normal_write; *output* for cg_boco_info)
NormalDataType	Data type used in the definition of the normals. Admissible data types for the normals are RealSingle and RealDouble. (Input for cg_boco_normal_write; *output* for cg_boco_info)

`NormalList`	List of vectors normal to the boundary condition patch pointing into the interior of the zone. (Input for `cg_boco_normal_write`; *output* for `cg_boco_read`)
`ndataset`	Number of boundary condition datasets for the current boundary condition. (*Output*)
`location`	Grid location used in the definition of the point set. The currently admissible locations are `Vertex` (the default if not given). For 2-D grids, `EdgeCenter` is also allowed, and for 3-D grids, the additional values of `FaceCenter`, `IFaceCenter`, `JFaceCenter`, and `KFaceCenter` may be used. (Input for `cg_boco_gridlocation_write`; *output* for `cg_boco_gridlocation_read`)
`ier`	Error status. (*Output*)

Notes: (see CPEX 0031)

- The use of `ElementList` and `ElementRange` for `ptset_type` is deprecated and should not be used in new code. These are still currently accepted, but will be internally replaced with the appropriate values of `PointList/PointRange` and `GridLocation_t`.
- `CellCenter` for `GridLocation_t` is also deprecated. If used, the value will be replaced by `EdgeCenter` for 2-D grids or `FaceCenter` for 3-D grids.

14.2 Boundary Condition Datasets

Node: `BCDataSet_t`

Functions	Modes
`ier = cg_dataset_write(int fn, int B, int Z, int BC, char *DatasetName, BCType_t BCType, int *Dset);`	`- w m`
`ier = cg_dataset_read(int fn, int B, int Z, int BC, int Dset, char *DatasetName, BCType_t *BCType, int *DirichletFlag, int *NeumannFlag);`	`r - m`
`call cg_dataset_write_f(fn, B, Z, BC, DatasetName, BCType, Dset, ier)`	`- w m`
`call cg_dataset_read_f(fn, B, Z, BC, Dset, DatasetName, BCType, DirichletFlag, NeumannFlag, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Zone index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`BC`	Boundary condition index number, where $1 \leq$ `BC` $\leq$ `nbocos`. (Input)

`Dset`	Dataset index number, where $1 \leq$ `Dset` $\leq$ `ndataset`. (Input for `cg_dataset_read`; *output* for `cg_dataset_write`)
`DatasetName`	Name of dataset. (Input for `cg_dataset_write`; *output* for `cg_dataset_read`)
`BCType`	Simple boundary condition type for the dataset. The supported types are listed in the table of "Simple Boundary Condition Types" in the SIDS manual, but note that `FamilySpecified` does not apply here. (Input for `cg_dataset_write`; *output* for `cg_dataset_read`)
`DirichletFlag`	Flag indicating if the dataset contains Dirichlet data. (*Output*)
`NeumannFlag`	Flag indicating if the dataset contains Neumann data. (*Output*)
`ier`	Error status. (*Output*)

The above functions are applicable to `BCDataSet_t` nodes that are children of `BC_t` nodes.

For `BCDataSet_t` nodes that are children of a `BC_t` node, after accessing a particular `BCDataSet_t` node using `cg_goto`, the Point Set functions described in Section 9.2 may be used to read or write the locations at which the boundary conditions are to be applied. This is only applicable when the boundary conditions are to be applied at locations different from those used with `cg_boco_write` to define the boundary condition region (e.g., when the region is being defined by specification of vertices, but the boundary conditions are to be applied at face centers).

When writing point set data to a `BCDataSet_t` node, in addition to the specification of the indices using `cg_ptset_write`, the function `cg_gridlocation_write` must also be used to specify the location of the data with respect to the grid (e.g., `Vertex` or `FaceCenter`).

Functions	Modes
`ier = cg_bcdataset_write(char *DatasetName, BCType_t BCType, BCDataType_t BCDataType);`	`- w m`
`ier = cg_bcdataset_info(int *ndataset);`	
`ier = cg_bcdataset_read(int Dset, char *DatasetName, BCType_t *BCType, int *DirichletFlag, int *NeumannFlag);`	`r - m`
`call cg_bcdataset_write_f(DatasetName, BCType, BCDataType_t BCDataType, ier)`	`- w m`
`call cg_bcdataset_info_f(int *ndataset, ier)`	
`call cg_bcdataset_read_f(Dset, DatasetName, BCType, DirichletFlag, NeumannFlag, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`Dset`	Dataset index number, where $1 \leq$ `Dset` $\leq$ `ndataset`. (Input)
`DatasetName`	Name of dataset. (Input for `cg_bcdataset_write`; *output* for `cg_bcdataset_read`)

`BCType`	Simple boundary condition type for the dataset. The supported types are listed in the table of "Simple Boundary Condition Types" in the SIDS manual, but note that `FamilySpecified` does not apply here. (Input for `cg_bcdataset_write`; *output* for `cg_bcdataset_read`)
`BCDataType`	Type of boundary condition in the dataset (i.e., for a `BCData_t` child node). Admissible types are `Dirichlet` and `Neumann`. (Input)
`ndataset`	Number of `BCDataSet` nodes under the current `FamilyBC_t` node. (*Output*)
`DirichletFlag`	Flag indicating if the dataset contains Dirichlet data. (*Output*)
`NeumannFlag`	Flag indicating if the dataset contains Neumann data. (*Output*)
`ier`	Error status. (*Output*)

The above functions are applicable to `BCDataSet_t` nodes that are used to define boundary conditions for a CFD family, and thus are children of a `FamilyBC_t` node. The `FamilyBC_t` node must first be accessed using `cg_goto`.

The first time `cg_bcdataset_write` is called with a particular `DatasetName`, `BCType`, and `BCDataType`, a new `BCDataSet_t` node is created, with a child `BCData_t` node. Subsequent calls with the same `DatasetName` and `BCType` may be made to add additional `BCData_t` nodes, of type `BCDataType`, to the existing `BCDataSet_t` node.

14.3 Boundary Condition Data

Node: `BCData_t`

Functions	Modes
`ier = cg_bcdata_write(int fn, int B, int Z, int BC, int Dset, BCDataType_t BCDataType);`	`- w m`
`call cg_bcdata_write_f(fn, B, Z, BC, Dset, BCDataType, ier)`	`- w m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \le$ `B` $\le$ `nbases`. (Input)
`Z`	Zone index number, where $1 \le$ `Z` $\le$ `nzones`. (Input)
`BC`	Boundary condition index number, where $1 \le$ `BC` $\le$ `nbocos`. (Input)
`Dset`	Dataset index number, where $1 \le$ `Dset` $\le$ `ndataset`. (Input)
`BCDataType`	Type of boundary condition in the dataset. Admissible boundary condition types are `Dirichlet` and `Neumann`. (Input)
`ier`	Error status. (*Output*)

To write the boundary condition data itself, after creating the `BCData_t` node using the function `cg_bcdata_write`, use `cg_goto` to access the node, then `cg_array_write` to write the data. Note that when using `cg_goto` to access a `BCData_t` node, the node index should be specified as either `Dirichlet` or `Neumann`, depending on the type of boundary condition. See the description of `cg_goto` in Section 4 for details.

14.4 Special Boundary Condition Properties

Node: BCProperty_t

Functions	Modes
ier = cg_bc_wallfunction_write(int fn, int B, int Z, int BC, WallFunctionType_t WallFunctionType);	- w m
ier = cg_bc_area_write(int fn, int B, int Z, int BC, AreaType_t AreaType, float SurfaceArea, *char *RegionName*);	- w m
ier = cg_bc_wallfunction_read(int fn, int B, int Z, int BC, *WallFunctionType_t *WallFunctionType*);	r - m
ier = cg_bc_area_read(int fn, int B, int Z, int BC, *AreaType_t *AreaType*, *float *SurfaceArea*, *char *RegionName*);	r - m
call cg_bc_wallfunction_write_f(fn, B, Z, BC, WallFunctionType, *ier*)	- w m
call cg_bc_area_write_f(fn, B, Z, BC, AreaType, SurfaceArea, RegionName, *ier*)	- w m
call cg_bc_wallfunction_read_f(fn, B, Z, BC, *WallFunctionType*, *ier*)	r - m
call cg_bc_area_read_f(fn, B, Z, BC, *AreaType*, *SurfaceArea*, *RegionName*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

fn	CGNS file index number. (Input)
B	Base index number, where $1 \leq$ B $\leq$ nbases. (Input)
Z	Zone index number, where $1 \leq$ Z $\leq$ nzones. (Input)
BC	Boundary condition index number, where $1 \leq$ BC $\leq$ nbocos. (Input)
WallFunctionType	The wall function type. Valid types are CG_Null, CG_UserDefined, and Generic. (Input for cg_bc_wallfunction_write; *output* for cg_bc_wallfunction_read)
AreaType	The type of area. Valid types are CG_Null, CG_UserDefined, BleedArea, and CaptureArea. (Input for cg_bc_area_write; *output* for cg_bc_area_read)
SurfaceArea	The size of the area. (In Fortran, this is a Real*4 value.) (Input for cg_bc_area_write; *output* for cg_bc_area_read)
RegionName	The name of the region, 32 characters max. (Input for cg_bc_area_write; *output* for cg_bc_area_read)
ier	Error status. (*Output*)

The "write" functions will create the BCProperty_t node if it doesn't already exist, then add the appropriate boundary condition property. Multiple boundary condition properties may be recorded under the same BCProperty_t node.

The "read" functions will return with ier = 2 = CG_NODE_NOT_FOUND if the requested boundary condition property, or the BCProperty_t node itself, doesn't exist.

15 Equation Specification

15.1 Flow Equation Set

Node: FlowEquationSet_t

Functions	Modes
`ier = cg_equationset_write(int EquationDimension);`	`- w m`
`ier = cg_equationset_read(int *EquationDimension, int *GoverningEquationsFlag, int *GasModelFlag, int *ViscosityModelFlag, int *ThermalConductModelFlag, int *TurbulenceClosureFlag, int *TurbulenceModelFlag);`	`r - m`
`ier = cg_equationset_chemistry_read(int *ThermalRelaxationFlag, int *ChemicalKineticsFlag);`	`r - m`
`ier = cg_equationset_elecmagn_read(int *ElecFldModelFlag, int *MagnFldModelFlag, ConductivityModelFlag);`	`r - m`
`call cg_equationset_write_f(EquationDimension, ier)`	`- w m`
`call cg_equationset_read_f(EquationDimension, GoverningEquationsFlag, GasModelFlag, ViscosityModelFlag, ThermalConductModelFlag, TurbulenceClosureFlag, TurbulenceModelFlag, ier)`	`r - m`
`call cg_equationset_chemistry_read_f(ThermalRelaxationFlag, ChemicalKineticsFlag, ier)`	`r - m`
`call cg_equationset_elecmagn_read_f(ElecFldModelFlag, MagnFldModelFlag, ConductivityModelFlag, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`EquationDimension`	Dimensionality of the governing equations; it is the number of spatial variables describing the flow. (Input for `cg_equationset_write`; *output* for `cg_equationset_info`)
`GoverningEquationsFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of the governing equations; 0 if it doesn't, 1 if it does. (*Output*)
`GasModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a gas model; 0 if it doesn't, 1 if it does. (*Output*)
`ViscosityModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a viscosity model; 0 if it doesn't, 1 if it does. (*Output*)
`ThermalConductModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a thermal conductivity model; 0 if it doesn't, 1 if it does. (*Output*)

`TurbulenceClosureFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of the turbulence closure; 0 if it doesn't, 1 if it does. (*Output*)
`TurbulenceModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a turbulence model; 0 if it doesn't, 1 if it does. (*Output*)
`ThermalRelaxationFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of thae thermal relaxation model; 0 if it doesn't, 1 if it does. (*Output*)
`ChemicalKineticsFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a chemical kinetics model; 0 if it doesn't, 1 if it does. (*Output*)
`ElecFldModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of an electric field model for electromagnetic flows; 0 if it doesn't, 1 if it does. (*Output*)
`MagnFldModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a magnetic field model for electromagnetic flows; 0 if it doesn't, 1 if it does. (*Output*)
`ConductivityModelFlag`	Flag indicating whether or not this `FlowEquationSet_t` node includes the definition of a conductivity model for electromagnetic flows; 0 if it doesn't, 1 if it does. (*Output*)
`ier`	Error status. (*Output*)

15.2 Governing Equations

Node: `GoverningEquations_t`

Functions	Modes
`ier = cg_governing_write(GoverningEquationsType_t Equationstype);`	`- w m`
`ier = cg_governing_read(GoverningEquationsType_t *EquationsType);`	`r - m`
`ier = cg_diffusion_write(int *diffusion_model);`	`- w m`
`ier = cg_diffusion_read(int *diffusion_model);`	`r - m`
`call cg_governing_write_f(EquationsType, ier)`	`- w m`
`call cg_governing_read_f(EquationsType, ier)`	`r - m`
`call cg_diffusion_write_f(diffusion_model, ier)`	`- w m`
`call cg_diffusion_read_f(diffusion_model, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

`EquationsType`	Type of governing equations. The admissible types are `CG_Null`, `CG_UserDefined`, `FullPotential`, `Euler`, `NSLaminar`, `NSTurbulent`,

	NSLaminarIncompressible, and NSTurbulentIncompressible. (Input for cg_governing_write; *output* for cg_governing_read)
diffusion_model	Flags defining which diffusion terms are included in the governing equations. This is only applicable to the Navier-Stokes equations with structured grids. See the discussion of GoverningEquations_t in the SIDS manual for details. (Input for cg_diffusion_write; *output* for cg_diffusion_read)
ier	Error status. (*Output*)

15.3 Auxiliary Models

Nodes: GasModel_t, ViscosityModel_t, ThermalConductivityModel_t, TurbulenceClosure_t, TurbulenceModel_t, ThermalRelaxationModel_t, ChemicalKineticsModel_t, EMElectricFieldModel_t, EMMagneticFieldModel_t, EMConductivityModel_t

Functions	Modes
ier = cg_model_write(char *ModelLabel, ModelType_t ModelType);	- w m
ier = cg_model_read(char *ModelLabel, *ModelType_t *ModelType*);	r - m
call cg_model_write_f(ModelLabel, ModelType, *ier*)	- w m
call cg_model_read_f(ModelLabel, *ModelType*, *ier*)	r - m

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See “64-bit Fortran Portability and Issues” section.)

ModelLabel — The CGNS label for the model being defined. The models supported by CGNS are:

- GasModel_t
- ViscosityModel_t
- ThermalConductivityModel_t
- TurbulenceClosure_t
- TurbulenceModel_t
- ThermalRelaxationModel_t
- ChemicalKineticsModel_t
- EMElectricFieldModel_t
- EMMagneticFieldModel_t
- EMConductivityModel_t

(Input)

ModelType — One of the model types (listed below) allowed for the ModelLabel selected. (Input for cg_model_write; *output* for cg_model_read)

ier — Error status. (*Output*)

The types allowed for the various models are:

GasModel_t	CG_Null, CG_UserDefined, Ideal, VanderWaals, CaloricallyPerfect, ThermallyPerfect, ConstantDensity, RedlichKwong
ViscosityModel_t	CG_Null, CG_UserDefined, Constant, PowerLaw, SutherlandLaw
ThermalConductivityModel_t	CG_Null, CG_UserDefined, PowerLaw, SutherlandLaw, ConstantPrandtl
TurbulenceModel_t	CG_Null, CG_UserDefined, Algebraic_BaldwinLomax, Algebraic_CebeciSmith, HalfEquation_JohnsonKing, OneEquation_BaldwinBarth, OneEquation_SpalartAllmaras, TwoEquation_JonesLaunder, TwoEquation_MenterSST, TwoEquation_Wilcox
TurbulenceClosure_t	CG_Null, CG_UserDefined, EddyViscosity, ReynoldsStress, ReynoldsStressAlgebraic
ThermalRelaxationModel_t	CG_Null, CG_UserDefined, Frozen, ThermalEquilib, ThermalNonequilib
ChemicalKineticsModel_t	CG_Null, CG_UserDefined, Frozen, ChemicalEquilibCurveFit, ChemicalEquilibMinimization, ChemicalNonequilib
EMElectricFieldModel_t	CG_Null, CG_UserDefined, Constant, Frozen, Interpolated, Voltage
EMMagneticFieldModel_t	CG_Null, CG_UserDefined, Constant, Frozen, Interpolated
EMConductivityModel_t	CG_Null, CG_UserDefined, Constant, Frozen, Equilibrium_LinRessler, Chemistry_LinRessler

16 Families

16.1 Family Definition

Node: `Family_t`

Functions	Modes
`ier = cg_family_write(int fn, int B, char *FamilyName, int *Fam);`	`- w m`
`ier = cg_nfamilies(int fn, int B, int *nfamilies);`	`r - m`
`ier = cg_family_read(int fn, int B, int Fam, char *FamilyName, int *nFamBC, int *nGeo);`	`r - m`
`call cg_family_write_f(fn, B, FamilyName, Fam, ier)`	`- w m`
`call cg_nfamilies_f(fn, B, nfamilies, ier)`	`r - m`
`call cg_family_read_f(fn, B, Fam, FamilyName, nFamBC, nGeo, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`nfamilies`	Number of families in base `B`. (*Output*)
`Fam`	Family index number, where $1 \leq$ `Fam` $\leq$ `nfamilies`. (Input for `cg_family_read`; *output* for `cg_family_write`)
`FamilyName`	Name of the family. (Input for `cg_family_write`; *output* for `cg_family_read`)
`nFamBC`	Number of boundary conditions for this family. This should be either 0 or 1. (*Output*)
`nGeo`	Number of geometry references for this family. (*Output*)
`ier`	Error status. (*Output*)

16.2 Geometry Reference

Node: `GeometryReference_t`

Functions	Modes
`ier = cg_geo_write(int fn, int B, int Fam, char *GeoName, char *FileName, char *CADSystem, int *G);`	`- w m`
`ier = cg_geo_read(int fn, int B, int Fam, int G, char *GeoName, char **FileName, char *CADSystem, int *nparts);`	`r - m`
`ier = cg_part_write(int fn, int B, int Fam, int G, char *PartName, int *P);`	`- w m`
`ier = cg_part_read(int fn, int B, int Fam, int G, int P, char *PartName);`	`r - m`
`call cg_geo_write_f(fn, B, Fam, GeoName, FileName, CADSystem, G, ier)`	`- w m`
`call cg_geo_read_f(fn, B, Fam, G, GeoName, FileName, CADSystem, nparts, ier)`	`r - m`
`call cg_part_write_f(fn, B, Fam, G, PartName, P, ier)`	`- w m`
`call cg_part_read_f(fn, B, Fam, G, P, PartName, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Fam`	Family index number, where $1 \leq$ `Fam` $\leq$ `nfamilies`. (Input)
`G`	Geometry reference index number, where $1 \leq$ `G` $\leq$ `nGeo`. (Input for `cg_geo_read`, `cg_part_write`, `cg_part_read`; *output* for `cg_geo_write`)
`P`	Geometry entity index number, where $1 \leq$ `P` $\leq$ `nparts`. (Input for `cg_part_read`; *output* for `cg_part_write`)
`GeoName`	Name of `GeometryReference_t` node. (Input for `cg_geo_write`; *output* for `cg_geo_read`)
`FileName`	Name of geometry file. (Input for `cg_geo_write`; *output* for `cg_geo_read`)
`CADSystem`	Geometry format. (Input for `cg_geo_write`; *output* for `cg_geo_read`)
`nparts`	Number of geometry entities. (*Output*)
`PartName`	Name of a geometry entity in the file `FileName`. (Input for `cg_part_write`; *output* for `cg_part_read`)
`ier`	Error status. (*Output*)

Note that with `cg_geo_read` the memory for the filename character string, `FileName`, will be allocated by the Mid-Level Library. The application code is responsible for releasing this memory when it is no longer needed by calling `cg_free(FileName)`, described in Section 10.6.

16.3 Family Boundary Condition

Node: FamilyBC_t

Functions	Modes
`ier = cg_fambc_write(int fn, int B, int Fam, char *FamBCName, BCType_t BCType, int *BC);`	- w m
`ier = cg_fambc_read(int fn, int B, int Fam, int BC, char *FamBCName, BCType_t *BCType);`	r - m
`call cg_fambc_write_f(fn, B, Fam, FamBCName, BCType, BC, ier)`	- w m
`call cg_fambc_read_f(fn, B, Fam, BC, FamBCName, BCType, ier)`	r - m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where 1 ≤ `B` ≤ `nbases`. (Input)

`Fam` Family index number, where 1 ≤ `Fam` ≤ `nfamilies`. (Input)

`BC` Family boundary condition index number. This must be equal to 1. (Input for `cg_fambc_read`; *output* for `cg_fambc_write`)

`FamBCName` Name of the `FamilyBC_t` node. (Input for `cg_fambc_write`; *output* for `cg_fambc_read`)

`BCType` Boundary condition type for the family. See the eligible types for `BCType_t` in Section 2.6. (Input for `cg_fambc_write`; *output* for `cg_fambc_read`)

`ier` Error status. (*Output*)

16.4 Family Name

Node: FamilyName_t

Functions	Modes
`ier = cg_famname_write(char *FamilyName);`	- w m
`ier = cg_famname_read(char *FamilyName);`	r - m
`call cg_famname_write_f(FamilyName, ier)`	- w m
`call cg_famname_read_f(FamilyName, ier)`	r - m

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`FamilyName` Family name. (Input for `cg_famname_write`; *output* for `cg_famname_read`)

`ier` Error status. (*Output*)

17 Time-Dependent Data

17.1 Base Iterative Data

Node: BaseIterativeData_t

Functions	Modes
`ier = cg_biter_write(int fn, int B, char *BaseIterName, int Nsteps);`	`- w m`
`ier = cg_biter_read(int fn, int B, char *BaseIterName, int *Nsteps);`	`r - m`
`call cg_biter_write_f(fn, B, BaseIterName, Nsteps, ier)`	`- w m`
`call cg_biter_read_f(fn, B, BaseIterName, Nsteps, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`BaseIterName`	Name of the `BaseIterativeData_t` node. (Input for `cg_biter_write`; *output* for `cg_biter_read`)
`Nsteps`	Number of time steps or iterations. (Input for `cg_biter_write`; *output* for `cg_biter_read`)
`ier`	Error status. (*Output*)

17.2 Zone Iterative Data

Node: ZoneIterativeData_t

Functions	Modes
`ier = cg_ziter_write(int fn, int B, int Z, char *ZoneIterName);`	`- w m`
`ier = cg_ziter_read(int fn, int B, int Z, char *ZoneIterName);`	`r - m`
`call cg_ziter_write_f(fn, B, Z, ZoneIterName, ier)`	`- w m`
`call cg_ziter_read_f(fn, B, Z, ZoneIterName, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Family index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`ZoneIterName`	Name of the `ZoneIterativeData_t` node. (Input for `cg_ziter_write`; *output* for `cg_ziter_read`)
`ier`	Error status. (*Output*)

17.3 Rigid Grid Motion

Node: `RigidGridMotion_t`

Functions	Modes
`ier = cg_rigid_motion_write(int fn, int B, int Z, char *RigidGridMotionName, RigidGridMotionType_t RigidGridMotionType, int *R);`	`- w m`
`ier = cg_n_rigid_motions(int fn, int B, int Z, int *n_rigid_motions);`	`r - m`
`ier = cg_rigid_motion_read(int fn, int B, int Z, int R, char *RigidGridMotionName, RigidGridMotionType_t RigidGridMotionType);`	`r - m`
`call cg_rigid_motion_write_f(fn, B, Z, RigidGridMotionName, RigidGridMotionType, R, ier)`	`- w m`
`call cg_n_rigid_motions_f(fn, B, Z, n_rigid_motions, ier)`	`r - m`
`call cg_rigid_motion_read_f(fn, B, Z, R, RigidGridMotionName, RigidGridMotionType, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Family index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`RigidGridMotionName`	Name of the `RigidGridMotion_t` node. (Input for `cg_rigid_motion_write`; *output* for `cg_rigid_motion_read`)
`RigidGridMotionType`	Type of rigid grid motion. The admissible types are `CG_Null`, `CG_UserDefined`, `ConstantRate`, and `VariableRate`. (Input for `cg_rigid_motion_write`; *output* for `cg_rigid_motion_read`)
`n_rigid_motions`	Number of `RigidGridMotion_t` nodes under zone `Z`. (*Output*)
`R`	Rigid rotation index number, where $1 \leq$ `R` $\leq$ `n_rigid_motions`. (Input for `cg_rigid_motion_read`; *output* for `cg_rigid_motion_write`)
`ier`	Error status. (*Output*)

17.4 Arbitrary Grid Motion

Node: `ArbitraryGridMotion_t`

Functions	Modes
`ier = cg_arbitrary_motion_write(int fn, int B, int Z, char *ArbitraryGridMotionName, ArbitraryGridMotionType_t ArbitraryGridMotionType, int *A);`	`- w m`
`ier = cg_n_arbitrary_motions(int fn, int B, int Z, int *n_arbitrary_motions);`	`r - m`
`ier = cg_arbitrary_motion_read(int fn, int B, int Z, int A, char *ArbitraryGridMotionName, ArbitraryGridMotionType_t ArbitraryGridMotionType);`	`r - m`
`call cg_arbitrary_motion_write_f(fn, B, Z, ArbitraryGridMotionName, ArbitraryGridMotionType, A, ier)`	`- w m`
`call cg_n_arbitrary_motions_f(fn, B, Z, n_arbitrary_motions, ier)`	`r - m`
`call cg_arbitrary_motion_read_f(fn, B, Z, A, ArbitraryGridMotionName, ArbitraryGridMotionType, ier)`	`r - m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn`	CGNS file index number. (Input)
`B`	Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)
`Z`	Family index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)
`ArbitraryGridMotionName`	Name of the `ArbitraryGridMotion_t` node. (Input for `cg_arbitrary_motion_write`; *output* for `cg_arbitrary_motion_read`)
`ArbitraryGridMotionType`	Type of arbitrary grid motion. The admissible types are `CG_Null`, `CG_UserDefined`, `NonDeformingGrid`, and `DeformingGrid`. (Input for `cg_arbitrary_motion_write`; *output* for `cg_arbitrary_motion_read`)
`n_arbitrary_motions`	Number of `ArbitraryGridMotion_t` nodes under zone `Z`. (*Output*)
`A`	Arbitrary grid motion index number, where $1 \leq$ `A` $\leq$ `n_arbitrary_motions`. (Input for `cg_arbitrary_motion_read`; *output* for `cg_arbitrary_motion_write`)
`ier`	Error status. (*Output*)

17.5 Zone Grid Connectivity

Node: `ZoneGridConnectivity_t`

Functions	Modes
`ier = cg_nzconns(int fn, int B, int Z, int *nzconns);`	`r - m`
`ier = cg_zconn_read(int fn, int B, int Z, int ZC, char *zcname);`	`r - m`
`ier = cg_zconn_write(int fn, int B, int Z, const char *zcname, int *ZC);`	`- w m`
`ier = cg_zconn_set(int fn, int B, int Z, int ZC);`	`r w m`
`ier = cg_zconn_get(int fn, int B, int Z, int *ZC);`	`r w m`
`call cg_nzconns_f(fn, B, Z, nzconns, ier)`	`r - m`
`call cg_zconn_read_f(fn, B, Z, ZC, zcname, ier)`	`r - m`
`call cg_zconn_write_f(fn, B, Z, zcname, ZC, ier)`	`- w m`
`call cg_zconn_set_f(fn, B, Z, ZC, ier)`	`r w m`
`call cg_zconn_get_f(fn, B, Z, ZC, ier)`	`r w m`

Input/*Output*

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`fn` CGNS file index number. (Input)

`B` Base index number, where $1 \leq$ `B` $\leq$ `nbases`. (Input)

`Z` Family index number, where $1 \leq$ `Z` $\leq$ `nzones`. (Input)

`ZC` Zone grid connectivity index number, where $1 \leq$ `ZC` $\leq$ `nzconns`. (Input for `cg_zconn_read` and `cg_zconn_set`; *output* for `cg_zconn_write` and `cg_zconn_get`)

`nzconns` Number of `ZoneGridConnectivity_t` nodes under Zone `Z`. (Output)

`zcname` Name of the `ZoneGridConnectivity_t` node. (Input for `cg_zconn_write`; *output* for `cg_zconn_read`)

`ier` Error status. (*Output*)

This allows for the specification of multiple `ZoneGridConnectivity_t` nodes. If these functions are not used, or `cg_zconn_write` is called once with a `zcname` of `ZoneGridConnectivity`, then there will be no differences from previous versions of the CGNS library. However, with multiple `ZoneGridConnectivity_t` nodes, there is an implicit *current* `ZoneGridConnectivity_t` node on which subsequent grid connectivity functions will operate, i.e. `cg_conn_read/write` (Section 13.2) or `cg_1to1_read/write` (Section 13.1).

The functions `cg_zconn_read` and `cg_zconn_write` will implicity set the *current* `ZoneGrid-Connectivity_t` node, while `cg_zconn_set` explicitly sets it. The functions `cg_nzconns` and `cg_zconn_get` do not change it.

The time-dependent changes to the connectivities may then be recorded in the `ZoneInterative-Data_t` (Section 17.2) node as an array of `ZoneGridConnectivityPointers`.

18 Links

Functions	Modes
`ier = cg_link_write(char *nodename, char *filename,`	`- w m`
`      char *name_in_file);`	
`ier = cg_is_link(int *path_length);`	`r - m`
`ier = cg_link_read(char **filename, char **link_path);`	`r - m`
`call cg_link_write_f(nodename, filename, name_in_file, ier)`	`- w m`
`call cg_is_link_f(path_length, ier)`	`r - m`
`call cg_link_read_f(filename, link_path, ier)`	`r - m`

Input/***Output***

(Note that for Fortran calls, all integer arguments are integer*4 in 32-bit mode and integer*8 in 64-bit mode. See "64-bit Fortran Portability and Issues" section.)

`nodename` — Name of the link node to create, e.g., `GridCoordinates`. (Input)

`filename` — Name of the linked file, or empty string if the link is within the same file. (Input for `cg_link_write`; *output* for `cg_link_read`)

`name_in_file` — Path name of the node which the link points to. This can be a simple or a compound name, e.g., `Base/Zone 1/GridCoordinates`. (Input)

`path_length` — Length of the path name of the linked node. The value 0 is returned if the node is not a link. (*Output*)

`link_path` — Path name of the node which the link points to. (*Output*)

`ier` — Error status. (*Output*)

Use `cg_goto(_f)`, described in Section 4, to position to a location in the file prior to calling these routines.

When using `cg_link_write`, the node being linked to does not have to exist when the link is created. However, when the link is used, an error will occur if the linked-to node does not exist.

Only nodes that support child nodes will support links.

It is assumed that the CGNS version for the file containing the link, as determined by the `CGNSLibraryVersion_t` node, is also applicable to `filename`, the file containing the linked node.

Memory is allocated by the library for the return values of the C function `cg_link_read`. This memory should be freed by the user when no longer needed by calling `cg_free(filename)` and `cg_free(link_path)`, described in Section 10.6.