Silo User Manual

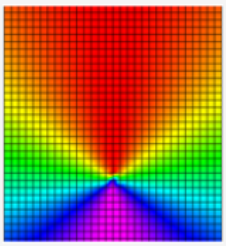
Release 4.11.1

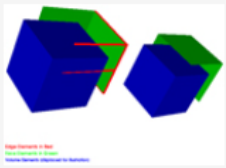
LLNL

May 04, 2024

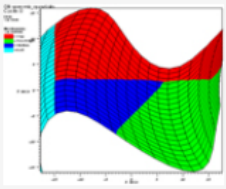
CONTENTS

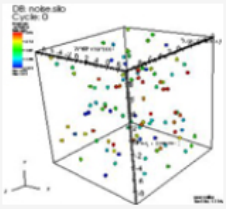
Silo is a library for reading and writing a wide variety of scientific data to binary, files. The files Silo produces and the data within them can be easily shared and exchanged between wholly independently developed applications running on disparate computing platforms. Consequently, Silo facilitates the development of general purpose tools for processing scientific data. One of the more popular tools that process Silo data files is the VisIt visualization tool.

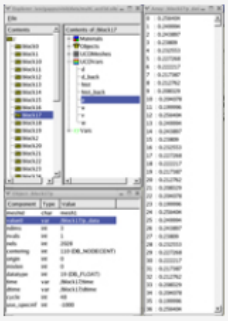

Structured Rectilinear Mesh

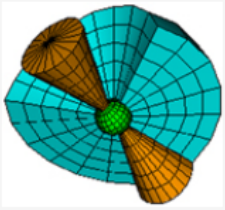

Arbitrary Subsets

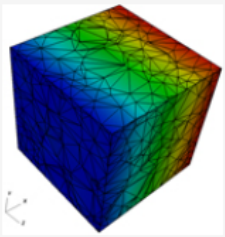

Constructive Solid Geometry (CSG) Mesh

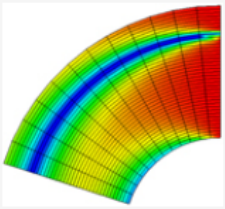

Mixing Materials

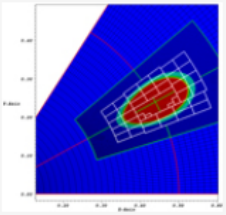

Gridless Point Mesh

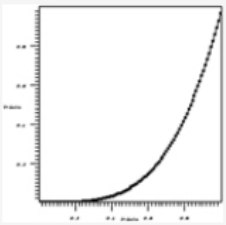

Silex browser for Silo files


Unstructured Zoo (UCD) Mesh


Arbitrary Polyhedral Mesh


Structured (Curvilinear) Mesh


Adaptive Mesh Refinement (AMR) Mesh


XY Curve

Silo supports gridless (point) meshes, structured meshes, unstructured-zoo and unstructured-arbitrary-polyhedral meshes, block structured AMR meshes, constructive solid geometry (CSG) meshes, piecewise-constant (e.g., zone-centered) and piecewise-linear (e.g. node-centered) variables defined on the node, edge, face or volume elements of meshes as well as the decomposition of meshes into arbitrary subset hierarchies including materials and mixing materials. In addition, Silo supports a wide variety of other useful objects and metadata to address various scientific computing application needs. Although the Silo library is a serial library, key features enable it to be applied effectively in scalable, parallel applications using the Multiple Independent File (MIF) parallel I/O paradigm.

Architecturally, the library is divided into two main pieces; an upper-level application programming interface (API) and a lower-level I/O implementation called a *driver*. Silo supports multiple I/O drivers. The two most common are the HDF5 (Hierarchical Data Format 5) and PDB (Portable DataBase) drivers.

CHAPTER

ONE

MAJOR SECTIONS OF THE USER'S MANUAL

1.1 Introduction to Silo

1.1.1 Overview

Silo is a library which implements an application programing interface (API) designed for reading and writing a wide variety of scientific data to binary, files. The files Silo produces and the data within them can be easily shared and exchanged between wholly independently developed applications running on disparate computing platforms.

Consequently, the Silo API facilitates the development of general purpose tools for processing scientific data. One of the more popular tools that process Silo data files is the VisIt visualization tool.

Silo supports gridless (point) meshes, structured meshes, unstructured-zoo and unstructured-arbitrary-polyhedral meshes, block structured AMR meshes, constructive solid geometry (CSG) meshes as well as piecewise-constant (e.g. zone-centered) and piecewise-linear (e.g. node-centered) variables defined on the node, edge, face or volume elements of meshes as well as the decomposition of meshes into arbitrary subset hierarchies including materials and mixing materials. In addition, Silo supports a wide variety of other useful objects and metadata to address various scientific computing application needs. Although the Silo library is a serial library, key features enable it to be applied effectively in scalable, parallel applications using the Multiple Independent File (MIF) parallel I/O paradigm.

Architecturally, the library is divided into two main pieces; an upper-level application programming interface (API) and a lower-level I/O implementation called a *driver*. Silo supports multiple I/O drivers, the two most common of which are the HDF5 (Hierarchical Data Format 5) and PDB (Portable DataBase, an API and binary database file format developed at LLNL by Stewart Brown and not to be confused with Protien Database also abbreviated as PDB) drivers. However, the reader should take care not to infer from this that Silo can read *any* HDF5 or PDB file. It cannot. For the most part, Silo is able to read only files that it has also written.

1.1.2 Where to Find Example Code

In the `tests` directory within the Silo source tree, there are numerous example C codes that demonstrate the use of Silo for writing various types of data. There are not as many examples of reading the data there.

If you are interested in point meshes, for example, you would search (e.g. `grep -i pointmesh`) for `DBPutPointMesh`. Or, if you are interested in how to use some option like `DBOPT_CONSERVED`, search for it within the C files in the `tests` directory.

1.1.3 Brief History and Background

Development of the Silo library began in the early 1990's at Lawrence Livermore National Laboratory to address a range of issues related to the storage and exchange of data among a wide variety of scientific computing applications and platforms.

In the early days of scientific computing, roughly 1950 - 1980, simulation software development at many labs, like Livermore, invariably took the form of a number of software *stovepipes*. Each big code effort included sub-efforts to develop supporting tools for data browsing and differencing, visualization, and management.

Developers working in a particular stovepipe designed every piece of software they wrote, simulation code and tools alike, to conform to a common representation for the data. In a sense, all software in a particular stovepipe was really just one big, monolithic application, held together by a common, binary or ASCII file format.

Data exchanges across stovepipes were laborious and often achieved only by employing one or more computer scientists whose sole task in life was to write a conversion tool called a *linker*. Worse, each linker needed to be kept it up to date as changes were made to one or the other codes that it linked. In short, there was nothing but brute force data sharing and exchange. Furthermore, there was duplication of effort in the development of data analysis and management tools for each code.

Between 1980 and 2000, an important innovation emerged, the general purpose I/O library. In fact, two variants emerged each working at a different level of abstraction. One focused on lower level abstractions…the *objects* of computer science. That is data structures such as arrays, structs and linked lists. The other focused on higher level abstractions…the *objects* of computational science. That is meshes and fields defined thereon.

Examples of the former are netCDF, HDF (HDF4 and HDF5) and PDB. Examples of the latter are ExodusII, Mili and Silo. At the same time, the higher level libraries are often implemented on top the lower level libraries. For example, Silo is implemented on top of either HDF5 or PDB and ExodusII is implemented on top of netCDF.

1.1.4 Silo Architecture

Silo has several drivers. Some are read-only and some are read-write. These are illustrated in the figure below…

Silo supports both read and write on the PDB (Portable DataBase) and HDF5 drivers. In addition, Silo supports two different *flavors* of PDB drivers. One known within Silo as *PDBLite* and is just called *PDB* which is a very old version of PDB that was frozen into the Silo library in 1999. That is the default driver. The other flavor of PDB is known within Silo as *PDB Proper* and can use a more recent release of the PDB library.

Although Silo can write and read PDB and HDF5 files, it cannot read just any PDB or HDF5 file. It can read only PDB or HDF5 files that were also written with Silo. Silo supports only read on the Taurus and netCDF drivers. The particular driver used to write data is chosen by an application when a Silo file is created. It can be automatically determined by the Silo library when a Silo file is opened.

Reading Silo Files

The Silo library has application-level routines to be used for reading mesh and mesh-related data. These functions return compound C data structures which represent data in a general way.

Writing Silo files

The Silo library contains application-level routines to be used for writing mesh and mesh-related data into Silo files.

In the C interface, the application provides a compound C data structure representing the data. In the Fortran interface, the data is passed via individual arguments.

Terminology

Here is a short summary of some of the terms used throughout the Silo interface and documentation. These terms are common to most computer simulation environments.

Mesh
: A discretization of a computational *domain* over which variables (fields) are defined. A mesh is a collection of *points* (aka *nodes*) optionally knitted together to form a network of higher dimensional primitive shapes (aka *elements*) via enumeration (explicit or implicit) of nodal *connectivities*.

  Practically speaking, a mesh consists of a list of nodes with coordinates and, optionally, one or more list(s) of elements, where each element is defined either directly in terms of the nodes or indirectly in terms of other elements which are ultimately defined in terms of the nodes.

  A mesh supports two notions of *dimension*. One is its *geometric* (or spatial) dimension and the other is its *parametric* (or topological) dimension. For example, the path of a rocket going into orbit around the Earth has three geometric dimensions (e.g. latitude, longitude and elevation). However, in *parametric* terms, that path is really just a one dimensional mesh (e.g. a curve).

Node
: A mathematical point. The fundamental building-block of the elements of a mesh. Other names for node are *point* or *vertex*. The topological dimension of a *node* is always zero. A point has no extent in any dimension or coordinate space.

Zone
: An element of a mesh. An element defines a region of support over which a variable (aka *field*, see below) may be interpolated. Zones are typically polygons or polyhedra with nodes as their vertices. Other names for zone are *cell* or *element*.

Variable (or *Field*)
: A field defined on a computational mesh or portion thereof. Variables usually represent some physical quantity (e.g., pressure or velocity) but that is not a requirement. Variables typically account for the overwhelming majority of data stored in a Silo database.

  The set of *numbers* stored for a variable are in general not the field's *values* but instead the field's *degrees of freedom* used in a given numerical scheme to *interpolate* the field over the *elements* of a mesh. However, for piecewise-constant and piecewise-linear interpolation schemes over the standard zoo of element shapes and types, the numbers stored are indeed also the field's values. This is due to the fact that the associated interpolation schemes are indeed *interpolating* as opposed to *approximating*.

  The terms *zone-centered* (or *cell-centered* or *element-centered*) and *node-centered* (or *vertex-centered*) are synonyms for piecewise-constant and piecewise-linear interpolation schemes, respectively

Coordinate Field
: The coordinates of a mesh are a field like any other field. The coordinates are a *special* field and must obey certain mathematical properties to serve as *coordinates*. However, it is important to understand that the coordinates of a mesh are also just a field.

  In Silo, coordinate fields are written as part of the `DBPutXxxmesh()` methods whereas other fields on the mesh are written with `DBPutXxxvar()` methods. The coordinates are always node-centered (e.g. piecewise-linear interpolating). It may be possible to support higher order interpolation schemes in the coordinate fields by adopting certain use conventions which downstream post-processing tools will also need to be made aware of.

Material
A decomposition of a mesh into distinct regions having different properties of some kind. Typically, different groups of elements of the mesh represent different physical materials such as brass or steel.

In the case of *mixing* materials, a single element may include contributions from more than one constituent material. In this case, the *fractions* of each material contained in the element is also part of the material description.

Material Species
A decomposition of a material into different concentrations of pure, atomic table elements. For example, *common yellow brass* is, nominally, a mixture of Copper (Cu) and Zinc (Zn) while *tool steel* is composed primarily of Iron (Fe) but mixed with some Carbon (C) and a variety of other elements. In certain computational science scenarios, detailed knowledge of the concentration of the constutient atomic elements comprising each material is needed.

Block
A *block* defines one coherent, contiguous piece (or fragment) of a larger mesh that has been decomposed into pieces typically for parallel processing but also potentially for other purposes such as streaming analysis, etc. The coordinates, connectivities and associated mesh variables of a block are all enumerated *relative* to the block and independently from any other block. In some sense, block's represent the fundamental storage *quanta* of a mesh that is too large to process as a single, monolithic whole.

A mesh that is decomposed into blocks is called a *multi-block* mesh. To go along with multi-block meshes, there are multi-block variables, multi-block materials and multi-block species. Different blocks of a larger mesh may be stored in different Silo files.

Frequently, blocks are also called *domains*.

1.1.5 Computational Meshes Supported by Silo

Silo supports several classes, or types, of meshes. These are quadrilateral, unstructured-zoo, unstructured-arbitrary, point, constructive solid geometry (CSG), and adaptive refinement meshes.

Quadrilateral-Based Meshes and Related Data

A quadrilateral mesh is one which contains four nodes per zone in 2-D and eight nodes per zone (four nodes per zone face) in 3-D. Quad meshes can be either regular, rectilinear, or curvilinear, but they must be logically rectangular

UCD-Based Meshes and Related Data

An unstructured cell data (UCD) mesh is a very general mesh representation; it is composed of an arbitrary list of zones of arbitrary sizes and shapes. Most meshes, including quadrilateral ones, can be represented as an unstructured mesh (Fig. 1-4). Because of their generality, however, unstructured meshes require more storage space and more complex algorithms.

In UCD meshes, the basic concept of zones (cells) still applies, but there is no longer an implied connectivity between a zone and its neighbor, as with the quadrilateral mesh. In other words, given a 2-D quadrilateral mesh zone accessed by (i, j), one knows that this zone's neighbors are (i-1,j), (i+1,j), (i, j-1), and so on. This is not the case with a UCD mesh.

In a UCD mesh, a structure called a zonelist is used to define the nodes which make up each zone. A UCD mesh need not be composed of zones of just one shape (Fig. 1-5). Part of the zonelist structure describes the shapes of the zones in the mesh and a count of how many of each zone shape occurs in the mesh. The facelist structure is analogous to the zonelist structure, but defines the nodes which make up each zone face.

Point Meshes and Related Data

A point mesh consists of a set of locations, or points, in space. This type of mesh is well suited for representing random scalar data, such as tracer particles.

Constructive Solid Geometry (CSG) Meshes and Related Data

A constructive Solid Geometry mesh is constructed by boolean combinations of solid model primitives such as spheres, cones, planes and quadric surfaces. In a CSG mesh, a *zone* is a region defined by such a boolean combination. CSG meshes support only zone-centered variables.

Block Structured, Adaptive Refinement Meshes (AMR) and Related Data

Block structured AMR meshes are composed of a large number of Quad meshes representing refinements of other quad meshes. The hierarchy of refinement is characterized using a Mesh Region Grouping (MRG) tree.

Summary of Silo's Computational Modeling Objects

Objects are a grouping mechanism for maintaining related variables, dimensions, and other data. The Silo library understands and operates on specific types of objects including the previously described computational meshes and related data. The user is also able to define arbitrary objects for storage of data if the standard Silo objects are not sufficient.

The objects are generalized representations for data commonly found in physics simulations. These objects include:

Quadmesh
: A mesh where the elements are implicitly defined by a *logical* cross product of the parametric dimensions. In one dimension, the elements are edges. In two dimensions, they are quadrilaterals In three dimensions, cubes. The geometric dimensions may also be implicitly defined by a cross product (e.g. rectilinear mesh) or they may be explicit (e.g. curvilinear mesh).

Quadvar
: A variable associated with a quad mesh. This includes the variable's data, centering information (node-centered vs. zone centered), and the name of the quad mesh with which this variable is associated. Additional information, such as time, cycle, units, label, and index ranges can also be included.

Ucdmesh
: An unstructured cell data (UCD) mesh. This is a mesh where the elements are only ever explicitly defined via enumeration of nodal connectivities. This includes the dimension, connectivity, and coordinate data, but typically also includes the mesh's coordinate system, labelling and unit information, minimum and maximum extents, and a list of face indices.

  Any quad mesh can be respresented as a UCD mesh. However, the reverse is not true.

  A quad mesh offers certain storage efficiencies (for the coordinate data) over UCD meshes. When the number of variables associated with a quad mesh is small, those efficiencies can be significant. However, as the number of variables grows, they are quickly washed out.

  When considering all the data stored in a Silo *file*, the storage efficiency is often not significant. However, when considering all the data needed in memory at any one time to perform a specific data analysis task (e.g. produce a Pseudocolor plot), the storage efficiency is indeed significant.

Ucdvar
A variable associated with a UCD mesh. This includes the variable's data, centering information (node-centered vs. zone-centered), and the name of the UCD mesh with which this variable is associated. Additional information, such as time, cycle, units, and label can also be included.

Pointmesh
A mesh consisting entirely of points as the mesh elements. A pointmesh has a parametric (topological) dimension of zero. However, a pointmesh can have a geometric dimension of 1, 2 or 3 (or more).

Csgmesh
A constructive solid geometry (CSG) mesh. This is a mesh where the elements are defined by set expressions (e.g. unions, intersections and differences) involving a handful of primitive shapes (e.g. sphers, cylinders, cones, etc.)

Csgvar
A variable defined on a CSG mesh (always piecewise-constant or zone centered).

Defvar
Defined variable representing an arithmetic expression involving other variables. The arithmetic expression may involve the names of functions (e.g. `log()`, `cos()`, etc.). The named functions may be specific to a given post-processing tool (e.g. the function `revolved_volume()` is known only to VisIt).

Material
An object defining all the materials present in a given mesh. This includes the number of materials present, a list of valid material identifiers, a zonal-length array which contains the material identifiers for each zone and, optionally, the material fractions associated with materials *mixing* in one or more zones.

Silo's mixed material data structure can trace its roots to Fortran codes developed at Livermore Labs in the early 1960s. It is a complicated data structure for software developers to deal with.

Zonelist
An object enumerating the nodal connectivities of elements comprising a mesh.

PHZonelist
An extension of a zonelist to support arbitrary polyhedra. In a PHZonelist, elements are enumerated in terms of their faces and the faces are enumerated in terms of their nodes.

Facelist
Face-oriented connectivity information for a UCD mesh. This object contains a sequential list of nodes which identifies the *external* faces of a mesh, and arrays which describe the shape(s) of the faces in the mesh. It may optionally include arrays which provide type information for each face.

Material species
Extra material information. A material species is a type of a material. They are used when a given material (i.e. air) may be made up of other materials (i.e. oxygen, nitrogen) in differing amounts.

Mesh Region Grouping (MRG) tree
Generalized mechanism used to define arbitray subsets of a mesh. MRG trees define how zones in the mesh may be grouped into parts, materials, boundary conditions, nodesets or facesets, etc.

Groupel Map
A *grouping element* map. Used in concert with an MRG tree to hold problem-sized data defining subsetted regions of meshes.

Multiblock
A way of specifying how a mesh is decomposed into pieces for I/O and computation

Multimesh
A set of mesh pieces (usually parallel decomposition) comprising a larger aggregate mesh object. This object contains the names of and types of the meshes in the set.

Multivar
A set of variable pieces comprising a larger aggregate variable object. Mesh variable data associated with a multimesh.

Multimat
A set of material pieces. This object contains the names of the materials in the set.

Multimatspecies
A set of material species. This object contains the names of the material species in the set.

Curve
X versus Y data. This object must contain at least the domain and range values, along with the number of points in the curve. In addition, a title, variable names, labels, and units may be provided.

Other Silo Objects

In addition to the objects listed in the previous section which are tailored to the job of representing computational data from scientific computing applications. Silo supports a number of other objects useful to scientific comput-ing applications. Some of the more useful ones are briefly summarized here.

Compound Array
A struct-like object which contains a list of similarly typed but differently named and sized (usually small) items that one often treats as a group (particularly for I/O purposes).

Directory
A silo file can be organized into directories (or folders) in much the same way as a Unix file system.

Optlist
An options list object used to pass additional options to various Silo API functions.

Simple Variable
A simple variable is just a named, multi-dimensional array of arbitrary data. This object contains, in addition to the data, the dimensions and data type of the array. This object is not required to be associated with any mesh.

User Defined Object
A generic, user-defined object of arbitrary composition.

Extended Silo Object
A Silo object which includes any number of user-defined additional data members.

1.1.6 Silo's Fortran Interface

The Silo library is implemented in C. Nonetheless, a set of Fortran callable wrappers have been written to make a majority of Silo's functionality available to Fortran applications. These wrappers simply take the data that is passed through a Fortran function interface, re-package it and call the equivalent C function. However, there are a few limitations of the Fortran interface.

Limitations of Fortran Interface

First, the Fortran interface is primarily a write-only interface. This means Fortran applications can use the interface to write Silo files so that other tools, like VisIt, can read them. However, for all but a few of Silo's objects, only the functions necessary to write the objects to a Silo file have been implemented in the Fortran interface. This means Fortran applications cannot really use Silo for restart file purposes.

Conceptually, the Fortran interface is identical to the C interface. To avoid duplication of documentation, the Fortran interface is documented right along with the C interface. However, because of differences in C and Fortran argument passing conventions, there are key differences in the interfaces. Here, we use an example to outline the key differences in the interfaces as well as the rules to be used to construct the Fortran interface from the C.

Conventions used to construct the Fortran interface from C

In this section, we show an example of a C function in Silo and its equivalent Fortran. We use this example to demonstrate many of the conventions used to construct the Fortran interface from the C.

We describe these rules so that Fortran user's can be assured of having up to date documentation (which tends to always first come for the C interface) but still be aware of key differences between the two.

A C function specification…

```
int DBAddRegionArray(DBmrgtree *tree, int nregn, const char **regn_names,
    int info_bits, const char *maps_name, int nsegs, int *seg_ids, int *seg_lens,
    int *seg_types, DBoptlist *opts)
```

The equivalent Fortran function…

```
integer function dbaddregiona(tree_id, nregn, regn_names, lregn_names,
    type_info_bits, maps_name, lmaps_name, nsegs, seg_ids, seg_lens, seg_types,
    optlist_id, status)

    integer tree_id, nregn, lregn_names, type_info_bits, lmaps_name
    integer nsegs, optlist_id, status
    integer lregn_names(), seg_ids(), seg_lens(), seg_types()
    character* maps_name
    character*N regn_names
```

About Fortran's `l<strname>` arguments

Wherever the C interface accepts a `char*`, the fortran interface accepts two arguments; the `character*` argument followed by an integer argument indicating the string's length. In the function specifications, it will always be identified with an ell (`l`) in front of the name of the `character*` argument that comes before it. In the example above, this rule is evident in the `maps_name` and `lmaps_name` arguments.

About Fortran's `l<strname>s` arguments

Wherever the C interface accepts an array of `char*` (e.g. `char**`), the Fortran interface accepts a `character*N` followed by an array of lengths of the strings. In the above example, this rule is evident by the `regn_names` and `lregn_names` arguments. By default, N=32, but the value for N can be changed, as needed by the `dbset2dstrlen()` method.

About Fortran's `<object>_id` arguments

Wherever the C interface accepts a pointer to an abstract Silo object, like the Silo database file handle (`DBfile *`) or, as in the example above, a `DBmrgtree*`, the Fortran interface accepts an equivalent *pointer id*. A *pointer id* really an integer index into an internally maintained table of pointers to Silo's objects. In the above example, this rule is evident in the `tree_id` aand `optlist_id` arguments.

About Fortran's `data_ids` arguments

Wherever the C interface accepts an array of `void*` (e.g. a `void**` argument), the Fortran interface accepts an array of integer *pointer ids*. The Fortran application may use the `dbmkptr()` function to a create the pointer ids to populate this array. The above example does not demonstrate this rule.

ABout Fortran's `status` arguments

Wherever the C interface returns integer error information in the return value of the function, the Fortran interface accepts an extra integer argument named `status` as the last argument in the list. The above example demonstrates this rule.

Finally, there are a few function in Silo's API that are unique to the Fortran interface. Those functions are described in the section of the API manual having to do with Fortran.

1.1.7 Using Silo in Parallel

Silo is a serial library. Nevertheless, it (as well as the tools that use it like VisIt) has several features that enable its effective use in parallel with excellent scaling behavior. However, using Silo effectively in parallel does require an application to store its data to multiple Silo files typically depending on the number of concurrent I/O channels the application has available at the time of Silo file creation.

The two features that enable Silo to be used effectively in parallel are its ability to create separate namespaces (directories) within a single file and the fact that a multi-block object can span multiple Silo files. With these features, aparallel application can easily divide its processors into N groups and write a separate Silo file for each group.

Within a group, each processor in the group writes to its own directory within the Silo file. One and only one processor has write access to the group's Silo file at any one time. So, I/O is serial within a group. However, because each group has a separate Silo file to write to, each group has one processor writing concurrently with other processors from other groups. So, I/O is parallel across groups.

After all processors have created all their individual objects in various directories within the each group's Silo file, one processor is designated to write multi-block objects. The multi-block objects serve as an assembly of the names of all the individual objects written from various processors.

When N, the number of processor groups, is equal to one, I/O is effectively serial. All the processors write their data to a single Silo file. When N is equal to the number of processors, each processor writes its data to its own, unique

Silo file. Both of these extremes are bad for effective and scalable parallel I/O. A good choice for N is the number of concurrent I/O channels available to the application when it is actually running.

This technique for using a serial I/O library effectively in parallel while being able to tune the number of files concurrently being written to is *Multiple Independent File (MIF)* parallel I/O.

There is a separate header file, `pmpio.h`, with a set of convenience CPP macros and methods to facilitate PMPIO-based parallel I/O with Silo.

1.2 Error Handling and Global Library Behavior

The functions described in this section of the Silo manual, are those that effect behavior of the library, globally, for any file(s) that are or will be created or opened. These include such things as error handling, requiring Silo to do extra work to warn of and avoid overwrites, to compute and warn of checksum errors and to compress data before writing it to disk.

1.2.1 Global and File-level variants

Some behaviors have both *file* level and *global* variants. For example, see *DBSetAllowOverwrites()* and *DBSetAllowOverwritesFile()*.

When a file is *opened or *created*, it *inherents* whatever the library's *global* settings are. However, a file's settings can be adjusted independently from the library's global settings by calling the equivalent `DBSetXxxFile()` methods.

1.2.2 DBErrfuncname()

- **Summary:** Get name of error-generating function
- **C Signature:**

```
char const *DBErrfuncname (void)
```

- **Fortran Signature:**

```
None
```

- **Returned value:**

  `DBErrfuncname` returns a `char const *` containing the name of the function that generated the last error. It cannot fail.

- **Description:**

  The `DBErrfuncname` function is used to find the name of the function that generated the last Silo error. It is implemented as a macro. The returned pointer points into Silo private space and must not be modified or freed.

1.2.3 DBErrno()

- **Summary:** Get internal error number.
- **C Signature:**

```
int DBErrno (void)
```

- **Fortran Signature:**

```
integer function dberrno()
```

- **Returned value:**

  DBErrno returns the internal error number of the last error. It cannot fail.

- **Description:**

  The DBErrno function is used to obtain the number of the last Silo error message. It returns the value of db_errno, a publicly available Silo library global variable.

1.2.4 DBErrString()

- **Summary:** Get error message.
- **C Signature:**

```
char const *DBErrString (void)
```

- **Fortran Signature:**

```
None
```

- **Returned value:**

  DBErrString returns a char const * containing the last error message. It cannot fail.

- **Description:**

  The DBErrString function is used to find the last Silo error message. It is implemented as a macro. The returned pointer points into Silo private space and must not be modified or freed.

1.2.5 DBShowErrors()

- **Summary:** Set the error reporting mode.
- **C Signature:**

```
void DBShowErrors (int level, void (*func)(char*))
```

- **Fortran Signature:**

```
integer function dbshowerrors(level)
```

- **Arguments:**

Arg name	Description
level	Error reporting level. One of DB_ALL, DB_ABORT, DB_TOP, or DB_NONE.
func	Function pointer to an error-handling function.

- **Returned value:**

  DBShowErrors returns nothing (void). It cannot fail.

- **Description:**

  The DBShowErrors function sets the level of error reporting done by Silo when it encounters an error. The following table describes the action taken upon error for different values of level.

  Ordinarily, error reporting from the driver (e.g. HDF5 or PDB) library is disabled. However, DBShowErrors can also control the behavior of error reporting from the driver library.

Error level value	Error action
DB_ALL	Show all errors, beginning with the (possibly internal) routine that first detected the error and continuing up the call stack to the application.
DB_ALL_AND_DRVR	Same as DB_ALL but also show error messages generated by the underlying driver library (PDB or HDF5).
DB_ABORT	Same as DB_ALL except abort is called after the error message is printed.
DB_TOP	(Default) Only the top-level Silo functions issue error messages.
DB_NONE	The library does not handle error messages. The application is responsible for checking the return values of the Silo functions and handling the error.

  For the HDF5 driver, when level is set to DB_ALL_AND_DRVR, this has the effect of calling H5Eset_auto((H5E_auto1_t) H5Eprint1, stderr) in the HDF5 library.

1.2.6 DBErrlvl()

- **Summary:** Return current error level setting of the library
- **C Signature:**

```
int DBErrlvl(void)
```

- **Fortran Signature:**

```
int dberrlvl()
```

- **Returned value:**

  Returns current error level of the library. Cannot fail.

1.2.7 DBErrfunc()

- **Summary:** Get current error function set by DBShowErrors()
- **C Signature:**

```
void (*func)(char*) DBErrfunc(void);
```

- **Fortran Signature:**

```
None
```

- **Description:**

  Returns the function pointer of the current error function set in the most recent previous call to `DBShowErrors()`. Testing whether we can type into this page and save it.

1.2.8 DBVariableNameValid()

- **Summary:** check if character string represents a valid Silo variable name
- **C Signature:**

```
int DBValidVariableName(char const *s)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
s	The character string to check

- **Returned value:**

  non-zero if the given character string represents a valid Silo variable name; zero otherwise

- **Description:**

  This is a convenience function for Silo applications to check whether a given variable name they wish to use will be considered valid by Silo.

  The only valid characters that can appear in a Silo variable name are all alphanumerics (e.g. `[a-zA-Z0-9]`) and the underscore (e.g. `'_'`). If a candidate variable name contains any characters other than these, that variable name is considered invalid. If that variable name is ever used in a call to create an object in a Silo file, the call will fail with error `E_INVALIDNAME`.

1.2.9 DBVersion()

- **Summary:** Get the version of the Silo library.
- **C Signature:**

```
char const *DBVersion (void)
```

- **Fortran Signature:**

```
None
```

- **Returned value:**

  DBVersion returns the version as a character string.

- **Description:**

  The DBVersion function determines what version of the Silo library is being used in the calling executable and returns that version in string form. The returned string should **not** be free'd by the caller.

1.2.10 DBVersionDigits()

- **Summary:** Return the integer version digits of the library
- **C Signature:**

```
int DBVersionDigits(int *Maj, int *Min, int *Pat, int *Pre);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
Maj	Pointer to returned major version digit
Min	Pointer to returned minor version digit
Pat	Pointer to returned patch version digit
Pre	Pointer to returned pre-release version digit (if any)

- **Returned value:**

  Returns 0 on success, -1 on failure..

- **Description:**

  The DBVersionDigits function determines what version of the Silo library is being used in the calling executable and returns the version digits.

1.2.11 DBVersionGE()

- **Summary:** Greater than or equal comparison for version of the Silo library
- **C Signature:**

```
int DBVersionGE(int Maj, int Min, int Pat)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
Maj	Integer, major version number
Min	Integer, minor version number
Pat	Integer, patch version number

- **Returned value:**

  One (1) if the version number of the library the calling executable is using is greater than or equal to the version number specified by Maj, Min, Pat arguments, zero (0) otherwise.

- **Description:**

  This function is the run-time equivalent of the *DB_VERSION_GE* macro.

1.2.12 DBSetAllowOverwrites()

1.2.13 DBSetAllowOverwritesFile()

- **Summary:** Allow library to over-write existing objects in Silo files
- **C Signature:**

```
int DBSetAllowOverwrites(int allow)
int DBSetAllowOverwritesFile(DBfile *dbfile, int allow)
```

- **Fortran Signature:**

```
integer function dbsetovrwrt(allow)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which the over-write property is desired.
allow	Integer value controlling the Silo library's overwrite behavior. A non-zero value sets the Silo library to permit overwrites of existing objects. A zero value disables overwrites. By default, Silo does **not** permit overwrites.

- **Returned value:**

  Returns the previous setting of the value.

- **Description:**

  By default, Silo permits a caller to over-write existing objects in a Silo file. However, overwrites are supported in Silo only when the new object involves individual member data that is no bigger than the original member data. If this condition is violated (which is sometimes difficult to predict), then behavior is undefined and will likely lead to corrupted data.

  If you suspect you have a case where unintended overwrites are occurring, it can be useful to turn off overwrites. Then, you can watch for any errors reported by the Silo library and determine if and where overwrites might be occurring.

  Note that there is currently a bug in the HDF5 driver where overwrites wind up orphaning existing data in the file rather than in fact overwriting that data. If the same object is overwritten many times, this could lead to unexpectedly large Silo files due to all the orphaned data. An HDF5 tool named `h5repack` maybe useful in reclaiming that wasted space.

1.2.14 DBGetAllowOverwrites()

1.2.15 DBGetAllowOverwritesFile()

- **Summary:** Get current setting for the allow overwrites flag
- **C Signature:**

```
int DBGetAllowOverwrites(void)
int DBGetAllowOverwritesFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetovrwrt()
```

- **Returned value:**

  Returns the current setting for the allow overwrites flag of the library or the specified file

- **Description:**

  See *DBSetAllowOverwrites* for a description of the meaning of this flag

1.2.16 DBSetAllowEmptyObjects()

1.2.17 DBSetAllowEmptyObjectsFile()

- **Summary:** Permit the creation of empty silo objects
- **C Signature:**

```
int DBSetAllowEmptyObjects(int allow)
int DBSetAllowEmptyObjectsFile(DBfile *file, int allow)
```

- **Fortran Signature:**

```
integer function dbsetemptyok(allow)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which the empty object setting is desired.
allow	Integer value indicating whether or not empty objects should be allowed to be created in Silo files. A zero value prevents callers from creating empty objects in Silo files. A non-zero value permits it. By default, the Silo library does **not** permit callers to create empty objects.

- **Returned value:**

  The previous setting of this value is returned.

- **Description:**

  For much of Silo's early history, the "EMPTY" keyword convention (see *DBPutMultimesh()*) was sufficient for dealing with cases where callers needed to create multiple, related multi-block objects with missing blocks. In fact, in many cases this convention was sufficient for combining variables which by and large existed on different collections of blocks on a common multi-block mesh.

  More recently, the need has arisen for the Silo library to permit callers to instantiate within Silo files actual empty objects; that is Silo objects with no problem-sized data associated with them. For example, a point mesh with no points or a ucd variable with no variable arrays. This requirement has been driven by the need to scale to larger problems and the use of nameschemes (see *DBMakeNamescheme()*) in combination with meshes and variables with missing blocks.

  Historically, such an operation has been considered an error by the Silo library and creating such objects was prevented. But, that has been largely an overly cautious restriction in Silo to avert anticipated and not necessarily any real problems. `DBSetAllowEmptyObjects()` with a non-zero argument enables the Silo library to by-pass these checks.

1.2.18 DBGetAllowEmptyObjects()

1.2.19 DBGetAllowEmptyObjectsFile()

- **Summary:** Get current setting for the allow empty objects flag
- **C Signature:**

```
int DBGetAllowEmptyObjects(void)
int DBGetAllowEmptyObjectsFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetemptyok()
```

- **Arguments:**

  None

- **Description:**

  Get the current library setting for the allow empty objects flag.

1.2.20 DBForceSingle()

- **Summary:** Convert all datatype'd data read in read methods to type float
- **C Signature:**

```
int DBForceSingle(int force)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`force`	Flag to indicate if forcing should be set or not. Pass non-zero to `force` single precision. Pass zero to **not** `force` single precision.

- **Returned value:**

  Zero on success. -1 on failure.

- **Description:**

  This setting is global to the whole library and effects subsequent read operations.

  If `force` is non-zero, then any datatype'd arrays are converted on read from whatever their native datatype is to float. A datatype'd array is an array that is part of some Silo object struct containing a `datatype` member which indicates the type of data in the array. For example, a `DBucdvar` struct has a `datatype` member to indicate the type of data in its var and mixvar arrays. Such arrays will be converted on read if `force` here is non-zero. However, a `DBmaterial` object struct is *always* integer data. There is no datatype member for such an object and so its data will *never* be converted to float on read regardless of the `force` single status set here.

  This function's original intention may have been to convert *only* double precision arrays to single precision on read. However, the PDB driver was apparently never designed that way and the PDB driver's behavior sort of established the defacto meaning of `DBForceSingle`. Consequently, as of Silo version 4.8 the HDF5 driver obeys these same semantics as well. Though, in fact the HDF5 driver was written to support the original intention of `DBForceSingle` and it worked in this (buggy) fashion for many years before real problems with it were encountered.

  This method is typically used by downstream, post-processing tools to reduce memory requirements. By default, Silo does not have single precision forcing enabled. When it is enabled, only the methods that result in reading of floating point data from a Silo file are effected. Finally, note that write methods are **not** effected.

  It might be nice for data producers to also use this method during write. That would be a potentially useful enhancement request because it alleviates software developers from having to write the conversion code before handing to Silo.

1.2.21 DBGetDatatypeString()

- **Summary:** Return a string name for a given Silo datatype
- **C Signature:**

```
char *DBGetDatatypeString(int datatype)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
datatype	One of the Silo datatypes (e.g.DB_INT,DB_FLOAT,DB_DOUBLE, etc.)

- **Returned value:**

  A pointer to a newly allocated string representing the data type name. The caller must free the returned string.

- **Description:**

  Obtain the string name of a given Silo datatype. Caller is responsible for freeing the returned string.

1.2.22 DBSetDataReadMask2()

1.2.23 DBSetDataReadMask2File()

- **Summary:** Set the data read mask
- **C Signature:**

```
unsigned long long DBSetDataReadMask2(unsigned long long mask)
unsigned long long DBSetDataReadMask2File(DBfile *dbfile, unsigned long long mask)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	the file whose read mask is to be set.
mask	The mask to use to read data. This is a bit vector of values that define whether each data portion of the various Silo objects should be read.

- **Returned value:**

  DBSetDataReadMask2 returns the previous data read mask.

- **Description:**

  DBSetDataReadMask2 replaces the now obsolete DBSetDataReadMask.

The DBSetDataReadMask2 allows the user to set the mask that's used to read various large data components within Silo objects.

Most Silo objects involve a combination of (small) metadata portion and a (potentially very large) data portion. The data portion is that part of the object that consists of pointers to potentially large arrays of data. These arrays are typically problem-sized but in any event require additional I/O to read. By default, the read mask is set to DBAll.

Setting the data read mask allows for a DBGetXxx() call to return only part of the associated object's data. With the data read mask set to DBAll, subsequent calls to DBGetXxx() functions return all of the information. With the data read mask set to DBNone, they return only the metadata. The mask is a bit vector specifying which part of the potentially problem-sized data should be read.

A special case is found in the DBCalc flag. Sometimes data is not stored in the file, but is instead calculated from other information. The DBCalc flag controls this behavior. If it is turned off, the data is not calculated. If it is turned on, the data is calculated.

The values that DBSetDataReadMask takes as the mask parameter are binary-or'ed combinations of the values shown in the following table:

Mask bit	Meaning
DBAll	All data values are read. This value is identical to specifying all of the other mask bits or'ed togeth
DBNone	No data values are read. This value sets all of the bit values to 0.
DBCalc	If data is calculable, calculate it. Otherwise, return NULL for that part.
DBMatMatnos	Material numbers (matnos) read by DBGetMaterial.
DBMatMatnames	Material names (matnames) read by DBGetMaterial.
DBMatMatlist	Zone-by-zone material list read by DBGetMaterial.
DBMatMixList	Mixed material information read by DBGetMaterial.
DBCurveArrays	Data values of curves read by DBGetCurve.
DBPMCoords	Coordinate arrays read by DBGetPointmesh.
DBPVData	Var data arrays read by DBGetPointvar.
DBQMCoords	Coordinate arrays read by DBGetQuadmesh.
DBQVData	Var data arrays read by DBGetQuadvar.
DBUMCoords	Coordinate arrays read by DBGetUcdmesh.
DBUMFacelist	Facelists of UCD meshes read by DBGetUcdmesh.
DBUMZonelist	Zonelists of UCD meshes read by DBGetUcdmesh.
DBUVData	Var data arrays read by DBGetUcdvar.
DBFacelistInfo	Nodelists and shape info read by DBGetFacelist.
DBZonelistInfo	Nodelist and shape info read by DBGetZonelist.
DBUMGlobNodeNo	Global node numbers read by DBGetUcdmesh
DBZonelistGlobZoneNo	Global zone numbers read by DBGetUcdmesh
DBMatMatcolors	Material colors read by DBGetMaterial and DBGetMultimat
DBMMADJNodelists	Adjacency nodelists read by DBGetMultimeshadj
DBMMADJZonelists	Adjacency zonelists read by DBGetMultimeshadj
DBCSGMBoundaryInfo	Boundary list read by DBGetCsgmesh
DBCSGMZonelist	Zonelist read by DBGetCsgmesh
DBCSGMBoundaryNames	Boundary names read by DBGetCsgmesh
DBCSGVData	Var data arrays read by DBGetCsgvar
DBCSGZonelistZoneNames	Zone names read by DBGetCSGZonelist
DBCSGZonelistRegNames	Region names read by DBGetCSGZonelist
DBPMGlobNodeNo	Global node numbers read by DBGetPointmesh
DBPMGhostNodeLabels	Ghost node labels read by DBGetPointmesh
DBQMGhostNodeLabels	Ghost node labels read by DBGetQuadmesh

Table 1 – continued from previous page

Mask bit	Meaning
DBQMGhostZoneLabels	Ghost zone labels read by DBGetQuadmesh
DBUMGhostNodeLabels	Ghost node labels read by DBGetUcdmesh
DBZonelistGhostZoneLabels	Ghost zone labels read by DBGetUcdmesh and/or DBGetZonelist

Use the DBGetDataReadMask2 call to retrieve the current data read mask without setting one.

By default, the data read mask is set to DBAll. The data read mask effects only the read portion of the Silo API.

1.2.24 DBGetDataReadMask2()

1.2.25 DBGetDataReadMask2File()

- **Summary:** Get the current data read mask
- **C Signature:**

```
unsigned long long DBGetDataReadMask2(void)
unsigned long long DBGetDataReadMask2File(DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Returned value:**

  DBGetDataReadMask2 returns the current data read mask.

- **Description:**

  Note that DBGetDataReadMask2 replaces the now obsolete DBGetDataReadMask.

  The DBGetDataReadMask2 allows the user to find out what mask is currently being used to read the data within Silo objects.

  See *DBSetDataReadMask2* for a complete description.

1.2.26 DBSetEnableChecksums()

1.2.27 DBSetEnableChecksumsFile()

- **Summary:** Set flag controlling checksum checks
- **C Signature:**

```
int DBSetEnableChecksums(int enable)
int DBSetEnableChecksumsFile(DBfile *dbfile, int enable)
```

- **Fortran Signature:**

```
integer function dbsetcksums(enable)
```

- **Arguments:**

Arg name	Description
`dbfile`	The file for which the checksum property should be set
`enable`	Integer value controlling checksum behavior of the Silo library. See description for a complete explanation.

- **Returned value:**

  Returns the previous setting for checksum behavior.

- **Description:**

  If checksums are enabled, whenever Silo writes data, it will compute checksums on the data in memory and store these checksums with the data in the file. Note that during a write call, in no circumstance will Silo re-read data written to confirm it was written correctly (e.g. it gets back what it wrote). In other words, Silo will not detect checksum errors on writes. It will detect checksum errors only on reads, only if checksums were actually computed and stored with the data when it was written and only when checksums are indeed enabled by the reader.

  If checksums are enabled, whenever Silo reads data *and* the data it is reading has checksums stored in the file, it will compute and compare checksums. If the checksums computed on read do not agree with the checksums stored in the file, the Silo call resulting in the data read will fail. The error, `E_CHECKSUM`, will be set (See *DBShowErrors*). Note that because checksums are not checked on write, there is no foolproof way to detect whether a read has failed because the data was corrupted when it was originally written or because the read itself has failed.

  Checksum checks are supported *only* on the HDF5 driver. The PDB driver does not support checksum checks. Calling `DBCreate()` with checksumming enabled will fail if `DB_PDB` is specified as the driver. If checksumming is enabled while any PDB file is opened, the request for checksumming will be silently ignored by all attempts to write or read data from a PDB file.

  In the HDF5 driver, only the data that winds up in HDF5 datasets in the file is checksummed. In most applications, this represents more than 99% of all the data the client writes. However, it is important to note that when checksumming is enabled, not all data written by Silo is checksummed. Various bits of metadata is not checksummed.

  Finally, empirical results show that the resulting files are 1-5% larger and take about 1-5% longer to write when checksumming is enabled. This is due primarily to the fact that a different class of HDF5 dataset, called a chunked dataset, is required in order to enable checksumming.

1.2.28 DBGetEnableChecksums()

1.2.29 DBGetEnableChecksumsFile()

- **Summary:** Get current state of flag controlling checksumming
- **C Signature:**

```
int DBGetEnableChecksums(void)
int DBGetEnableChecksumsFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetcksums()
```

- **Returned value:**

  Zero if checksumming is not currently enabled. Non-zero if checksumming is currently enabled.

- **Description:**

  This function returns the current setting for the library-global flag controlling checksumming behavior.

1.2.30 DBSetCompression()

1.2.31 DBSetCompressionFile()

- **Summary:** Set compression options for succeeding writes of Silo data
- **C Signature:**

```
int DBSetCompression(char const *options)
int DBSetCompressionFile(DBfile *dbfile, char const *options)
```

- **Fortran Signature:**

```
integer function dbsetcompress(options, loptions)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which compression settings should be set.
options	Character string containing the name of the compression method and various parameters. The method set using the keyword, "METHOD=". Any remaining parameters are dependent on the compression method and are described below.

- **Returned value:**

  Returns the previous value set for compression behavior.

- **Description:**

  Compression is currently supported only on the HDF5 driver.

  Note that the responsibility for enabling compression falls only on the data producer. Any Silo clients attempting to read compressed data may do so without concern for whether the data in the file is compressed or not. If the data is compressed, decompression will occur automatically during read. This is true as long as the Silo library to which the client reading the data was compiled and linked has the necessary decompression code. Because writer and reader need not be compiled and linked to the same exact Silo library installation, each could be compiled with differing compression capabilities making it impossible to read data in some situations.

  To the extent possible, the public installations of Silo on LLNL systems have all been enabled with compatible compression features. However, because many application developers have taken to creating their own installations of Silo, it is important to consider the effect of disabling (or enabling) various compression features.

  Compression features are controlled by an arbitrary string, whose contents are described in more detail below. By default, the Silo library does not have compression enabled. A number of different compression techniques are available. Some operate in a mesh and variable and data-type agnostic way while others depend on the type of data and sometimes even the type of mesh.

Some compression settings are global to all compression methods. There are two global parameters that control behavior of compression algorithms. These must appear in the compression `options` string before any compression-specific parameters.

The first is the error mode (`"ERRMODE=<word>"` which controls how the Silo library responds when it encounters an error during compression and/or is unable to compress the data. The two options for `<word>` are `FALLBACK` or `FAIL`. Including `"ERRMODE=FALLBACK"` in the compression options string tells Silo that whenever compression fails, it should simply fallback to writing uncompressed data. Including `"ERRMODE=FAIL"` in the compression `options` string tells Silo to fail the write and return `E_COMPRESSION` error for the operation.

The second is the minimum compression ratio to be achieved by compressing the data. It is specified as `"MINRATIO=<float>"`. For example, including `"MINRATIO=2.5"` in the compression options string tells Silo that all data must be compressed by at least a factor of 2.5:1. If it is unable the compress by at least this amount, Silo will either fallback or fail the write depending on the `ERRMODE` setting.

The remaining paragraphs describe compression algorithm specific options.

GZIP compression
is enabled using `"METHOD=GZIP"` in the options string.

GZIP recognizes the `LEVEL=<int>`, compression parameter. The compression level is an integer from 0 to 9, where 0 results in the fastest compression performance but at the expense of lower compression ratios. Likewise, a level of 9 results in the slowest compression performance but with possibly better compression ratios. If the `"LEVEL=<int>"` keyword does not appear in the options string or specifies invalid values, the default is level one (1). The GZIP method of compression is applied independently to float and integer data for all types of meshes and variables. It is also guaranteed to be available to all Silo clients.

When GZIP compression is applied to floating point data, a secondary HDF5 shuffle filter is also applied to improve compressibilty of the data.

SZIP compression:
is enabled using `"METHOD=SZIP"` in the options string. The SZIP compression algorithm is designed specifically for scientific data. SZIP recognizes the `BLOCK=<int>`, and `MASK={EC|NN}` parameters. The `BLOCK=<int>`, takes an integer value from 0 to 32, which is a blocking size and must be even and not greater than 32, with typical values being 8, 10, 16, or 32.

This parameter affects the compression ratio; the more values vary, the smaller this number should be to achieve better performance. The `MASK=EC` setting, selects entropy coding method, this is best suited for data that has been processed, working best for small numbers. The `MASK=NN` setting, selects the nearest neighbor coding method, preprocesses the data then applies the EC method as above. The default parameters for SZIP compression are `"METHOD=SZIP BLOCK=4 MASK=NN"`. If in a subsequent write operation the value for `BLOCK` is bigger than the total number of elements in a dataset, the write will fail. This means that you should take care not to have compression turned on when doing small writes. To achieve optimal performance for SZIP compression, it is recommended that one select a value for `BLOCK` that is an integral divisor of the dataset's fastest-changing dimension. Note that the SZIP compression encoder *may* be licensed for non-commercial use only while the decoder (e.g. decompression) is unlimited. Note that SZIP decompression is **not** guaranteed to be available to all Silo clients; only those for which the Silo library was configured with SZIP compression capability enabled. Like GZIP, SZIP compression is applied to float and integer data independently of the types of meshes and variables.

FPZIP compression
is enabled using `"METHOD=FPZIP"` in the options string. The FPZIP compression algorithm was developed by Peter Lindstrom at LLNL and is also designed for high speed compression of regular arrays of data. FPZIP recognizes the `"LOSS=0|1|2|3"` parameter which specifies the amount of loss that is tolerable in the result in terms of quarters of full precision. For example, `"LOSS=3"` indicates that a loss of 3/4 of full precision is tolerable (resulting in 8 bit floats or 16 bit doubles). Note that for data being written from a double precision writer for down stream visualization purposes, visualization tools such as VisIt often enforce single precision data. Therefore, specifying a loss of 32 bits here for double precision data

could have a dramatic impact on compression and I/O performance with negligible effect in down stream visualization. If the `LOSS` parameter is not specified, the default is `LOSS=0`. It is possible to build the Silo library without FPZIP compression support. So, it is not always guaranteed to exist.

HZIP compression
is enabled using `"METHOD=HZIP"` in the options string. The HZIP compression algorithm was developed by Peter Lindstrom at LLNL and is designed for high-speed compression of unstructured meshes of quad or hex elements and node-centered variables (it does not yet support zone-centered variables) defined on a mesh. Before applying this compression method to any given Silo mesh or variable object, the Silo library checks for compatibility with the constraints of the compression algorithm. If the mesh or variable object is compatible, the object will be written with compression enabled. Otherwise, compression will be silently ignored. It is possible to build the Silo library without HZIP compression support. So, it is not always guaranteed to exist.

Note that FPZIP and HZIP compression features are **not** available in a BSD Licensed release of Silo library. They are available only in a Legacy licensed release of the Silo library.

ZFP compression
is enabled using `"METHOD=ZFP"` in the options string. ZFP compression in the Silo library uses a built-in version of the H5Z-ZFP compression filter. Data compressed with Silo using ZFP is fully compatible with any reader using the H5Z-ZFP compression filter. Use `"RATE=<float>"` to set compression mode to use ZFP's rate-based compression. Use `"ACCURACY=<float>"` to set compression mode to use ZFP's accuracy-based compression. Use `"PRECISION=<int>"` to set compression mode to use ZFP's precision-based compression. Use `"EXPERT=<minbits,maxbits,maxprec,minexp>"` to set compression mode to use ZFP's expert compression mode. Use `"REVERSIBLE"` to set compression mode to use ZFP's reversible compression. Note that all other ZFP related parameters having to do with data type and array dimensions are handled by Silo automatically during each `DBPutXxx()` call.

1.2.32 DBGetCompression()

1.2.33 DBGetCompressionFile()

- **Summary:** Get current compression parameters
- **C Signature:**

```
char const *DBGetCompression()
char const *DBGetCompressionFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetcompress(options, loptions)
```

- **Arguments:**

  None

- **Returned value:**

  `NULL` if no compress parameters have been set. A string of compression parameters if compression has been set

- **Description:**

  Obtain the current compression parameters. Caller should **not** free the returned string.

1.2.34 DBSetFriendlyHDF5Names()

1.2.35 DBSetFriendlyHDF5NamesFile()

- **Summary:** Set flag to indicate Silo should create friendly names for HDF5 datasets
- **C Signature:**

```
int DBSetFriendlyHDF5Names(int enable)
int DBSetFriendlyHDF5NamesFile(DBfile *dbfile, int enable)
```

- **Fortran Signature:**

```
integer function dbsethdfnms(enable)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which HDF5 friendly names property should be set
enable	Flag to indicate if friendly names should be turned on (non-zero value) or off (zero).

- **Returned value:**

  Old setting for this flag

- **Description:**

  In versions of Silo prior to 4.8, the default behavior of the HDF5 driver was that it used HDF5 in a way that made the data somewhat UNnatural to the user when viewed with HDF5 tools such as `h5ls`, `h5dump` and `hdfview` as well as other tools that interact with the data via the HDF5 API. This was not a problem for Silo but was a problem for these and other HDF5 tools.

  `DBSetFriendlyHDF5Names()` was introduced as a way to address this issue so that the data in an HDF5 file written by Silo look more *natural* to HDF5. Calling `DBSetFriendlyHDF5Names()` with a value of one (1) will result in additional HDF5 metadata being added to the file (in the form of soft links) with better names (and locations) for Silo objects' datasets. Note that creation of links does increase the file size ever so slightly. This affect is less significant for larger files. It is also likely to have some negative but as yet to be investigated effect on I/O performance

  Calling `DBSetFriendlyHDF5Names()` with a value of two (2) will forgo the creation of soft links and instead write the actual dataset data where those links would have been created (e.g. the current working directory of the Silo file). This may be important for files consisting of a large number of objects as it eliminates the creation of the `/.silo` group and subsequent very large number of dataset objects in that one group.

  In versions of Silo 4.8 and newer, the default behavior of the Silo library is to use mode 2, that is to create the datasets themselves there the links would have otherwise been created.

  If it was not obvious from the name, this method effects only the HDF5 driver.

1.2.36 DBGetFriendlyHDF5Names()

1.2.37 DBGetFriendlyHDF5NamesFile()

- **Summary:** Get setting for friendly HDF5 names flag
- **C Signature:**

```
int DBGetFriendlyHDF5Names()
int DBGetFriendlyHDF5NamesFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgethdfnms()
```

- **Arguments:**

  None

- **Returned value:**

  The current setting for the HDF5 friendly names flag.

- **Description:**

  See *DBSetFriendlyHDF5Names()*.

1.2.38 DBSetDeprecateWarnings()

1.2.39 DBSetDeprecateWarningsFile()

- **Summary:** Set maximum number of deprecate warnings Silo will issue for any one function, option or convention
- **C Signature:**

```
int DBSetDeprecateWarnings(int max_count)
int DBSetDeprecateWarningsFile(DBfile *dbfile, int max_count)
```

- **Fortran Signature:**

```
integer function dbsetdepwarn(max_count)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which deprecation warning property should be set
max_count	Maximum number of warnings Silo will issue for any single API function.

- **Returned value:**

  The old maximum number of deprecate warnings

- **Description:**

  Some of Silo's API functions have been deprecated. Some options on Silo objects have also been deprecated. Finally, some conventional arrays, such as _visit_defvars, have been deprecated.

When an attempt to use a deprecated function, option or convention is detected, Silo will issue an error message on `stderr` and proceed normally. The default number of error messages any given deprecated function will report on `stderr` is 3. Note, this is on a per-deprecated function, option or convention basis. If this number is decreased to zero by calling `DBSetDeprecateWarnings(0)`, no warnings will be generated on `stderr`. If it is increased, more warnings will be issued.

Note that deprecated functions, options and conventions are guaranteed to operate correctly only in the first release in which they become deprecated. In subsequent releases, they may be removed entirely. So, it is wise to run your application for a while without turning off deprecation warnings to get some inventory of functions that require attention.

1.2.40 DBGetDeprecateWarnings()

1.2.41 DBGetDeprecateWarningsFile()

- **Summary:** Get maximum number of deprecated function warnings Silo will issue
- **C Signature:**

```
int DBGetDeprecateWarnings()
int DBGetDeprecateWarningsFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetdepwarn()
```

- **Arguments:**

  None

- **Returned value:**

  The current maximum number of deprecate warnings

- **Description:**

1.2.42 DBSetAllowLongStrComponents()

1.2.43 DBSetAllowLongStrComponentsFile()

- **Summary:** Allow for *long* string components of objects
- **C Signature:**

```
int DBSetAllowLongStrComponents(int allow)
int DBSetAllowLongStrComponentsFile(DBfile *dbfile, int allow)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	the file for which long string components should be allowed
allow	the setting for the flag

- **Returned value:**

  The previous value of the setting

- **Description:**

  In earlier versions of Silo, string valued components of user-defined objects (e.g. written with *DBWriteObject()* were limited in length to `1024` characters. This setting allows longer (arbitrary length) string components of user defined objects.

Warning: Longer than normal component strings can result in creating objects in Silo files that are not readable by older versions (<4.10.3) of the Silo library.

1.2.44 DB_VERSION_GE()

- **Summary:** Compile time macro to test silo version number
- **C Signature:**

```
DB_VERSION_GE(Maj,Min,Pat)
```

- **Arguments:**

Arg name	Description
`Maj`	Major version number digit
`Min`	Minor version number digit. A zero is equivalent to no minor digit.
`Pat`	Patch version number digit. A zero is equivalent to no patch digit.

- **Returned value:**

  True (non-zero) if the combination of major, minor and patch digits results in a version number of the Silo library that is greater (e.g. newer) than or equal to the version of the Silo library being compiled against. False (zero), otherwise.

- **Description:**

  This macro is useful for writing version-specific code that interacts with the Silo library. Note, however, that this macro appeared in version 4.6.1 of the Silo library and is not available in earlier versions of the library.

  As an example of use, the function `DBSetDeprecateWarnings()` was introduced in Silo version 4.6 and not available in earlier versions. You could use this macro like so...

```
#if DB_VERSION_GE(4,6,0)

DBSetDeprecateWarnings(0);

#endif
```

1.2.45 DBSetCompatibilityMode()

- **Summary:** Set compatibility mode for subsequent object creations
- **C Signature:**

```
int DBSetCompatibilityMode(int mode)
```

- **Fortran Signature:**

```
integer function dbsetcompat(mode)
```

- **Arguments:**

Arg name	Description
dbfile	The file for which HDF5 friendly names property should be set
mode	Mode to indicate how Silo library should behave for subsequent object creation operations. Pass either DB_COMPAT_OVER_PERF or DB_PERF_OVER_COMPAT. The default is DB_COMPAT_OVER_PERF. See description below for further details.

- **Returned value:**

  Previous setting for mode.

- **Description:**

  Silo's compatibility mode influences whether data can be exchanged between producers and consumers using dramatically different versions of Silo or an underlying I/O library such as HDF5 Silo depends on.

  By default, the Silo library operates to *maximize* compatibility of files created between newer and older versions of the library (or an underlying driver library such as HDF5). However, doing this may come at the cost of some lost performance. Better performance may be possible by using newer versions of Silo (or an underlying driver library such as HDF5) but the resulting data may not be readable by older versions of Silo (or older versions of an underlying driver library such as HDF5).

  Silo's compatibility mode is designed to enable producers and consumers to control the tradeoff between compatibility and performance. The compatibility mode can be set globally for the entire library with DBSetCompatibilityMode(). Alternatively, it can also be set for an individual Silo file when it is OR'd into the mode arg in the *DBCreate* or *DBOpen* calls. When it is OR'd into the mode arg, that setting takes precedent over the global setting.

  When DB_COMPAT_OVER_PERF is set (which is also the default), it means that when there is a choice in the way the Silo library behaves, compatibility will be prioritized over performance. In this mode, the Silo library endeavours to create data readable by the oldest version of Silo still likely in use (4.10) and/or the oldest version of the HDF5 library Silo is likely to be using (1.8).

  Compatability applies to *entire* files. A file written with DB_PERF_OVER_COMPAT from a newer Silo and/or HDF5 library may be unreadable by an application using an older Silo and/or HDF5 library.

  Tip: There are likely circumstances where setting compatibility mode to prioritize performance over compatibility will result in better performance. It is difficult to quantify *better* and it will vary based on the particular use-cases. However, *better* might be anywhere between 2x and 10x for specific operations.

  When performance considerations need to take precedence, callers are free to set the compatibility mode to DB_PERF_OVER_COMPAT which favors performance over compatibility. However, all stakeholders should be made aware that the Silo file may not be readable by applications using older verions of Silo (or older versions of an underlying driver library such as HDF5).

Danger: When DB_PERF_OVER_COMPAT is in effect (either for an individual file or for the whole library), files written may not be readable by all downstream consumers. This is more likely to be true if the data producer is using a newer version of Silo or HDF5 than a consumer and is less likely to be true if the data producer is using and older version of Silo or HDF5 than a consumer. In some cases, correcting downstream consumers may involve simply re-linking them with newer versions of Silo and/or HDF5. In other cases, correcting downstream consumers may require re-compiling them with newer versions of Silo and/or HDF5.

As Silo has evolved, various new features have been introduced either to Silo itself or the underlying driver I/O library Silo uses which can impact the backward compatibility.

An example of a Silo-level feature that impacted backward compatibility was the introduction of nameschemes for multi-block objects. Nameschemes were introduced in such a way that data producers using nameschemes produced files downstream consumers could no longer read correctly without modification. This is a bad outcome. This is especially true when an alternative would have been to have DBGetMultiXxx() methods (the methods that read multi-block objects) expand any nameschemes into the equivalent explicit lists of names thereby obeying the terms of the original, older DBGetMultiXxx() interface. Concievably, this is something that is possible to handle if DB_COMPAT_OVER_PERF is OR'd with DB_READ in a *DBOpen* call. Compatibility mode flags are not currently supported for DB_READ opens.

Nameschemes represent a backward compatibility issue that is restricted to *some* objects (e.g. multi-block objects) in a Silo file. Other objects of the Silo file will remain readable by older versions of Silo (or older versions of the underlying driver library such as HDF5). An example of a Silo feature that can impact compatibility of the *whole file* is the choice of virtual file driver (VFD). In some cases, a data producer can select a virtual file driver (often driven by I/O performance considerations) which can be handled by a data consumer only if the consumer had been previously set up to handle it. Any workflow involving a Silo data consumer that was not set up to handle that virtual file driver would not be able to even open the file.

The above examples underscore a commonly encountered compatibility tradeoff in persistent storage libraries like Silo. Compatibility (whether downstream consumers can read the data) is often pitted against performance. Best performance often requires data producers to utilize features of libraries that may break backward compatibility meaning downstream consumers using older versions of Silo (or older versions of the underlying driver library such as HDF5) may not be able to read the data without modification.

These issues extend also to the underlying I/O driver libraries Silo uses such as HDF5. Data producers using newer versions of HDF5 underneath Silo and using newer features that offer better performance can produce HDF5 files that downstream consumers may not be able to read without modification.

When compatibility is broken, in the most ideal cases a consumer may only need to be re-linked against a newer version of Silo and/or HDF5. This is the ideal case because it involves a minimum amount of work and can often be restricted to the consumer(s) needing it. In other cases, a consumer may need to be re-compiled and re-linked or worse, re-coded, re-compiled and re-linked. These latter cases are non-ideal because they often involve repercussions up and down the dependency chain. In these cases, no part of the workflow can be fixed until all parts of the workflow are fixed.

1.2.46 DBGetCompatibilityMode()

1.2.47 DBGetCompatibilityModeFile()

- **Summary:** Get compatibility mode
- **C Signature:**

```
int DBGetCompatibilityMode()
int DBGetCompatibilityModeFile(DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbgetcompat()
```

- **Arguments:**

  None

- **Returned value:**

  The current compatibility mode of the library or file.

- **Description:**

  Because compatibility mode can be set differently for a file than for the library globally, two methods are provided to retrieve it's value.

1.3 Files and File Structure

If you are looking for information regarding how to use Silo from a parallel application, please See the section on *Multi-Block Objects Parallel I/O*.

The Silo API is implemented on a number of different low-level drivers. These drivers control the low-level file format Silo generates. For example, Silo can generate PDB (Portable DataBase) and HDF5 formatted files. The specific choice of low-level file format is made at file creation time.

In addition, Silo files can themselves have directories (folders). That is, within a single Silo file, one can create directory hierarchies for storage of various objects. These directory hierarchies are analogous to the Unix file system. Directories serve to divide the name space of a Silo file so the user can organize content within a Silo file in a way that is natural to the application.

Note that the organization of objects into directories within a Silo file may have direct implications for how these collections of objects are presented to users by post-processing tools. For example, except for directories used to store multi-block objects (See *Multi-Block Objects Parallel I/O*), VisIt will use directories in a Silo file to create submenus within its Graphical User Interface (GUI). If VisIt opens a Silo file with two directories, "foo" and "bar", and there are various meshes and variables in each of these directories, then many of VisIt's GUI menus will contain submenus named "foo" and "bar" where the objects found in those directories will be placed in the GUI.

Silo also supports the concept of *grabbing* the low-level driver. For example, if Silo is using the HDF5 driver, an application can obtain the actual HDF5 file id and then use the native HDF5 API with that file id.

1.3.1 DBRegisterFileOptionsSet()

- **Summary:** Register a set of options for advanced control of the low-level I/O driver
- **C Signature:**

```
int DBRegisterFileOptionsSet(const DBoptlist *opts)
```

- **Fortran Signature:**

```
int dbregfopts(int optlist_id)
```

- **Arguments:**

Arg name	Description
opts	an options list object obtained from a DBMakeOptlist() call

- **Returned value:**

  -1 on failure. Otherwise, the integer index of a registered file options set is returned.

- **Description:**

  File options sets are used in concert with the `DB_HDF5_OPTS()` macro in `DBCreate` or `DBOpen` calls to provide advanced and fine-tuned control over the behavior of the underlying driver library and may be needed to affect memory usage and I/O performance as well as vary the behavior of the underlying I/O driver away from its default mode of operation.

  A file options set is nothing more than an optlist object (see *Optlists*), populated with file driver related options. A *registered* file options set is such an optlist that has been registered with the Silo library via a call to this method, `DBRegisterFileOptionsSet`. A maximum of 32 registered file options sets are currently permitted. Use `DBUnregisterFileOptionsSet` to free up a slot in the list of registered file options sets.

  Before a specific file options set may be used as part of a `DBCreate` or `DBOpen` call, the file options set must be registered with the Silo library. In addition, the associated optlist object should not be freed until after the last call to `DBCreate` or `DBOpen` in which it is needed.

  Presently, the options sets are defined for the HDF5 driver *only*. The table below defines and describes the various options. A key option is the selection of the HDF5 Virtual File Driver or VFD. See *`DBCreate`* for a description of the available VFDs.

  In the table of options below, some options are relevant to only a specific HDF5 VFD. Other options effect the behavior of the HDF5 library as a whole, regardless of which underlying VFD is used. This difference is notated in the scope column.

  All of the options described here relate to options documented in the HDF5 library's file access property lists.

  Note that all option names listed in left-most column of the table below have had their prefix "`DBOPT_H5_`" removed to save space in the table. So, for example, the real name of the `CORE_ALLOC_INC` option is `DBOPT_H5_CORE_ALLOC_INC`.

`DBOPT_H5_...`	Type	Meaning
`VFD`	`int`	Specifies which Virtual File Driver (VFD) the HDF5 library should use. Set the integer value for
`RAW_FILE_OPTS`	`int`	Applies only for the split VFD. Specifies a file options set to use for the raw data file. May be any
`RAW_EXTENSION`	`char*`	Applies only for the split VFD. Specifies the file extension/naming convention for raw data file. If
`META_FILE_OPTS`	`int`	Same as `DBOPT_H5_RAW_FILE_OPTS`, above, except for meta data file. See HDF5 reference manu
`META_EXTENSION`		Same as `DBOPT_H5_RAW_EXTENSION` above, except for meta data file. See HDF5 reference manu
`CORE_ALLOC_INC`	`int`	Applies only for core VFD. Specifies allocation increment. See HDF5 reference manual for H5Ps
`CORE_NO_BACK_STORE`	`int`	Applies only for core VFD. Specifies whether or not to store the file on close. See HDF5 referenc
`LOG_NAME`	`char*`	Applies only for the log VFD. This is primarily a debugging feature. Specifies name of the file to
`LOG_BUF_SIZE`	`int`	Applies only for the log VFD. This is primarily a debugging feature. Specifies size of the buffer to
`META_BLOCK_SIZE`	`int`	Applies the the HDF5 library as a whole (e.g. globally). Specifies the size of memory allocations
`SMALL_RAW_SIZE`	`int`	Applies to the HDF5 library as a whole (e.g. globally). Specifies a threshold below which allocati
`ALIGN_MIN`	`int`	Applies to the HDF5 library as a whole. Specified a size threshold above which all datasets are al
`ALIGN_VAL`	`int`	The alignment to be applied to datasets of size greater than `ALIGN_MIN`. See HDF5 reference man
`DIRECT_MEM_ALIGN`	`int`	Applies only to the direct VFD. Specifies the alignment option. See HDF5 reference manual for H
`DIRECT_BLOCK_SIZE`	`int`	Applies only to the direct VFD. Specifies the block size the underlying file system is using. See H
`DIRECT_BUF_SIZE`		Applies only to the direct VFD. Specifies a copy buffer size. See HDF5 reference manual for H5P
`MPIO_COMM`		
`MPIO_INFO`		
`MPIP_NO_GPFS_HINTS`		
`SIEVE_BUF_SIZE`	`int`	HDF5 sieve buf size. Only relevant if using either compression and/or checksumming. See HDF5
`CACHE_NELMTS`	`int`	HDF5 raw data chunk cache parameters. Only relevant if using either compression and/or checksu

DBOPT_H5_...	Type	Meaning
CACHE_NBYTES		HDF5 raw data chunk cache parameters. Only relevant if using either compression and/or checksu
CACHE_POLICY		HDF5 raw data chunk cache parameters. Only relevant if using either compression and/or checksu
FAM_SIZE	int	Size option for family VFD. See HDF5 reference manual for H5Pset_fapl_family. The family VFD
FAM_FILE_OPTS	int	VFD options for each file in family VFD. See HDF5 reference manual for H5Pset_fapl_family. Th
USER_DRIVER_ID	int	Specify some user-defined VFD. Permtis application to specify any user-defined VFD. See HDF5
USER_DRIVER_INFO		Specify user-defined VFD information struct. Permtis application to specify any user-defined VFI
SILO_BLOCK_SIZE	int	Block size option for Silo VFD. All I/O requests to/from disk will occur in blocks of this size.
SILO_BLOCK_COUNT	int	Block count option for Silo VFD. This is the maximum number of blocks the Silo VFD will main
SILO_LOG_STATS	int	Flag to indicate if Silo VFD should gather I/O performance statistics. This is primarily for debugg
SILO_USE_DIRECT	int	Flag to indicate if Silo VFD should attempt to use direct I/O. Tells the Silo VFD to use direct I/O
FIC_BUF	void*	The buffer of bytes to be used as the "file in core" to be opened in a DBOpen() call.
FIC_SIZE	int	Size of the buffer of bytes to be used as the "file in core" to be opened in a DBOpen() call.

1.3.2 DBUnregisterFileOptionsSet()

- **Summary:** Unregister a registered file options set
- **C Signature:**

```
int DBUnregisterFileOptionsSet(int opts_set_id)
```

- **Fortran Signature:**

```
int dbunregfopts(int optlist_id)
```

- **Arguments:**

Arg name	Description
opts_set_id	The identifer (obtained from a previous call to DBRegisterFileOptionsSet()) of a file options set to unregister.

- **Returned value:**

  Zero on success. -1 on failure.

- **Description:**

  Unregister a specific file options set identified by opts_set_id.

1.3.3 DBUnregisterAllFileOptionsSets()

- **Summary:** Unregister all file options sets
- **C Signature:**

```
int DBUnregisterAllFileOptionsSets()
```

- **Fortran Signature:**

```
int dbunregafopts()
```

- **Arguments:**

  None

- **Returned value:**

  Zero on success, -1 on failure.

- **Description:**

  Unregister all file options sets.

1.3.4 DBSetUnknownDriverPriorities()

- **Summary:** Set driver priorities for opening files with the DB_UNKNOWN driver.
- **C Signature:**

```
static const int *DBSetUnknownDriverPriorities(int *driver_ids)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
driver_ids	A -1 terminated list of driver ids such as DB_HDF5, DB_PDB, DB_HDF5_CORE, or any driver id constructed with the DB_HDF5_OPTS() macro.

- **Returned value:**

  The previous unknown driver prioritizatin as a -1 terminated array of integer values. The caller should **NOT** free the returned value.

- **Description:**

  When opening files with DB_UNKNOWN driver, Silo iterates over drivers, trying each until it successfuly opens a file.

  This call can be used to affect the order in which driver ids are attempted and can improve behavior and performance for opening files using DB_UNKNOWN driver.

  If any of the driver ids specified in driver_ids is constructed using the DB_HDF5_OPTS() macro, then the associated file options set must be registered with the Silo library.

1.3.5 DBGetUnknownDriverPriorities()

- **Summary:** Return the currently defined ordering of drivers the DB_UNKNOWN driver will attempt.
- **C Signature:**

```
static const int *DBGetUnknownDriverPriorities(void)
```

- **Fortran Signature:**

```
None
```

- **Description:**

  Returns a -1 terminated list of integer driver ids indicating the prioritzation of drivers used by the DB_UNKNOWN driver. The caller should **NOT** free the returned value.

1.3.6 DBCreate()

- **Summary:** Create a Silo output file.
- **C Signature:**

```
DBfile *DBCreate (char *pathname, int mode, int target,
    char *fileinfo, int filetype)
```

- **Fortran Signature:**

```
integer function dbcreate(pathname, lpathname, mode, target,
          fileinfo, lfileinfo, filetype, dbid)
```

  returns the newly created database file handle in dbid.

- **Arguments:**

Arg name	Description
pathname	Path name of file to create. This can be either an absolute or relative path.
mode	Creation mode. Pass DB_CLOBBER or DB_NOCLOBBER, optionally OR'd with DB_PERF_OVER_COMPAT or DB_COMPAT_OVER_PERF (see *DBSetCompatibilityMode*
target	Destination file format. In the distant past, this option was used to target binary numeric formats in the file to a specific host CPU architecture (such as Sun or Sgi or Cray). More recently, this argument has become less relevant and should most likely always be set to DB_LOCAL.
fileinfo	Character string containing descriptive information about the file's contents. This information is usually printed by applications when this file is opened. If no such information is needed, pass NULL for this argument.
filetype	Destination file type. Applications typically use one of either DB_PDB, which will create PDB files, or DB_HDF5, which will create HDF5 files. Other options include DB_PDBP, DB_HDF5_SEC2, DB_HDF5_STDIO, DB_HDF5_CORE, DB_HDF5_SPLIT or DB_FILE_OPTS(optlist_id) where optlist_id is a registered file options set. For a description of the meaning of these options as well as many other advanced features and control of underlying I/O behavior, see *DBRegisterFileOptionsSet*.

- **Returned value:**

  DBCreate returns a DBfile pointer on success and NULL on failure. If the pathname argument contains a *path* components (e.g. foo/bar/baz/file.silo), note that DBCreate creates only the file part of the path. The containing directories (folders) must already exist and be writeable.

- **Description:**

  The DBCreate function creates a Silo file and initializes it for writing data.

  Silo supports two underlying drivers for storing named arrays and objects of machine independent data. One is called the Portable DataBase Library (PDBLib or just PDB), and the other is Hierarchical Data Format, Version 5 (HDF5), http://www.hdfgroup.org/HDF5.

When Silo is configured with the `--with-pdb-proper=<path-to-PACT-PDB>` option, the Silo library supports both the PDB driver that is built-in to Silo (which is actually an ancient version of PACT's PDB referred to internally as *PDB Lite*) identified with a `filetype` of `DB_PDB` and a second variant of the PDB driver using a PACT installation (specified when Silo was configured) with a `filetype` of `DB_PDBP` (Note the trailing `P` for *PDB Proper*). PDB Proper is known to give far superior performance than PDB Lite on BG/P and BG/L class systems and so is recommended when using PDB driver on such systems.

For the HDF5 library, there are many more available options for fine tuned control of the underlying I/O through the use of HDF5's Virtual File Drivers (VFDs). For example, HDF5's `sec2` VFD uses Unix Manual Section 2 I/O routines (e.g. create/open/read/write/close) while the `stdio` VFD uses Standard I/O routines (e.g. fcreate/fopen/fread/fwrite/fclose).

Depending on the circumstances, the choice of VFD can have a profound impact on actual I/O performance. For example, on BlueGene systems the customized Silo block-based VFD (introduced to the Silo library in Version 4.8) has demonstrated excellent performance compared to the default HDF5 VFD; sec2. The remaining paragraphs describe each of the available Virtual File Drivers as well as parameters that govern their behavior.

DB_HDF5
: From among the several VFDs that come pre-packaged with the HDF5 library, this driver type uses whatever the HDF5 library defines as the default VFD. On non-Windows platforms, this is the Section 2 (see below) VFD. On Windows platforms, it is a Windows specific VFD.

DB_HDF5_SEC2
: Uses the I/O system interface defined in section 2 of the Unix manual. That is `create`, `open`, `read`, `write`, `close`. This is a VFD that comes pre-packaged with the HDF5 library. It does little to no optimization of I/O requests. For example, two I/O requests that abutt in file address space wind up being issued through the section 2 I/O routines as independent requests. This can be disasterous for high latency file systems such as might be available on BlueGene class systems.

DB_HDF5_STDIO
: Uses the Standard I/O system interface defined in Section 3 of the Unix manual. That is `fcreate`, `fopen`, `fread`, `fwrite`, `fclose`. This is a VFD that comes pre-packaged with the HDF5 library. It does little to no optimization of I/O requests. However, since it uses the stdio routines, it does benefit from whatever default buffering the implementation of the stdio interface on the given platform provides. Because section 2 routines are unbuffered, the sec2 VFD typically performs better when there are fewer, larger I/O requests while the stdio VFD performs better when there are more, smaller requests. Unfortunately, the metric for what constitutes a "small" or "large" request is system dependent. So, it helps to experiment with the different VFDs for the HDF5 driver by running some typically sized use cases. Some results on the Luster file system for tiny I/O requests (100's of bytes) showed that the stdio VFD can perform 100x or more better than the section 2. So, it pays to spend some time experimenting with this [Note: In future, it should be possible to manipulate the buffer used for a given Silo file opened via the stdio VFD as one would ordinarily do via such stdio calls as setvbuf(). However, due to limitations in the current implementation, that is not yet possible. When and if that becomes possible, to use something other than non-default stdio buffering, the Silo client will have to create and register an appropriate file options set.

DB_HDF5_CORE
: Uses a memory buffer for the file with the option of either writing the resultant buffer to disk or not. Conceptually, this VFD behaves more or less like a ramdisk. This is a VFD that comes pre-packaged with the HDF5 library. I/O performance is optimal in the sense that only a single I/O request for the entire file is issued to the underlying file system. However, this optimality comes at the expense of memory. The entire file must be capable of residing in memory. In addition, releases of HDF5 library prior to 1.8.2 support the core VFD only when creating a new file and not when open an existing file. Two parameters govern behavior of the core VFD. The *allocation increment* specifies the amount of memory the core VFD allocates, each time it needs to increase the buffer size to accomodate the (possibly growing) file. The *backing store* indicates whether the buffer should be saved to disk (if it has been changed) upon close. By default, using `DB_HDF5_CORE` as the driver type results in an allocation incriment of 1 Megabyte and a backing store option of TRUE, meaning it will store the file to disk upon close. To specify parameters other

than these default values, the Silo client will have to create and register an appropriate file options set, see *DBRegisterFileOptionsSet*.

DB_HDF5_SPLIT

Splits HDF5 I/O operations across two VFDs. One VFD is used for all raw data while the other VFD is used for everything else (e.g. meta data). For example, in Silo's `DBPutPointvar()` call, the data the caller passes in the vars argument is raw data. Everything else including the object's name, number of points, datatype, optlist options, etc. including all underlying HDF5 metadata gets treated as meta data. This is a VFD that comes pre-packaged with the HDF5 library. It results in two files being produced; one for the raw data and one for the meta data. The reason this can be a benefit is that tiny bits of metadata intermingling with large raw data operations can degrade performance overall. Separating the datastreams can have a profound impact on performance at the expense of two files being produced. Four parameters govern the behavior of the split VFD. These are the VFD and filename extension for the raw and meta data, respectively. By default, using `DB_HDF5_SPLIT` as the driver type results in Silo using sec2 and `"-raw"` as the VFD and filename extension for raw data and core (default params) and `""` (empty string) as the VFD and extension for meta data. To specify parameters other than these default values, the Silo client will have to create and register an appropriate file options set, see *DBRegisterFileOptionsSet*.

DB_HDF5_FAMILY

Allows for the management of files larger than 2^{32} bytes on 32-bit systems. The virtual file is decomposed into real files of size small enough to be managed on 32-bit systems. This is a VFD that comes pre-packaged with the HDF5 library. Two parameters govern the behavior of the family VFD. The size of each file in a family of files and the VFD used for the individual files. By default, using `DB_HDF5_FAMILY` as the driver type results in Silo using a size of 1 Gigabyte (2^{32}) and the default VFD for the individual files. To specify parameters other than these default values, the Silo client will have to create and register an appropriate file options set, see *DBRegisterFileOptionsSet*.

DB_HDF5_LOG

While doing the I/O for HDF5 data, also collects detailed information regarding VFD calls issued by the HDF5 library. The logging VFD writes detailed information regarding VFD operations to a logfile. This is a VFD that comes pre-packaged with the HDF5 library. However, the logging VFD is a different code base than any other VFD that comes pre-packaged with HDF5. So, while the logging information it produces is representative of the VFD calls made by HDF5 library to the VFD interface, it is **not** representative of the actual I/O requests made by the sec2 or stdio or other VFDs. Behavior of the logging VFD is governed by 3 parameters; the name of the file to which log information is written, a set of flags which are or'd together to specify the types of operations and information logged and, optionally, a buffer (which must be at least as large as the actual file being written) which serves to map the kind of HDF5 data (there are about 8 different kinds) stores at each byte in the file. By default, using `DB_HDF5_LOG` as the driver type results in Silo using a logfile name of "silo_hdf5_log.out", flags of `H5FD_LOG_LOC_IO|H5FD_LOG_NUM_IO|H5FD_LOG_TIME_IO|H5FD_LOG_ALLOC` and a `NULL` buffer for the mapping information. To specify parameters other than these default values, the Silo client will have to create and register an appropriate file options set, see *DBRegisterFileOptionsSet*.

Users interested in this VFD should consult HDF5's reference manual for the meaning of the flags as well as how to interepret logging VFD output.

DB_HDF5_DIRECT

On systems that support the `O_DIRECT` flag in section 2 `create/open` calls, this VFD will use direct I/O. This VFD comes pre-packaged with the HDF5 library. Most systems (both the system interfaces implementations for section 2 I/O as well as underlying file systems) do a lot of work to buffer and cache data to improve I/O performance. In some cases, however, this extra work can actually get in the way of good performance, particularly when the I/O operations are streaming like and large. Three parameters govern the behavior of the direct VFD. The alignment specifies memory alignment requirement of raw data buffers. That generally means that posix_memalign should be used to allocate any buffers you use to hold raw data passed in calls to the Silo library. The block size indicates the underlying file system block size and the copy buffer size gives the HDF5 library some additional flexibility in dealing with unaligned requests. Few

systems support the O_DIRECT flag and so this VFD is not often available in practice. However, when it is, using DB_HDF5_DIRECT as the driver type results in Silo using an alignment of 4 kilobytes (2^{12}), an alignment equal to the block size and a copy buffer size equal to 256 times the blocksize.

DB_HDF5_SILO
This is a custom VFD designed specifically to address some of the performance shortcommings of VFDs that come pre-packaged with the HDF5 library. The silo VFD is a very, very simple, block-based VFD. It decomposes the file into blocks, keeps some number of blocks in memory at any one time and issues I/O requests *only* in whole blocks using section 2 I/O routines. In addition, it sets up some parameters that control HDF5 library's allocation of meta data and raw data such that each block winds up consisting primirily of either raw or meta data but not both. It also disables meta data caching in HDF5 to reduce memory consumption of the HDF5 library to the bare minimum as there is no need for HDF5 to maintain cached metadata if it resides in blocks kept in memory in the VFD. This is a suitable VFD for most scientific computing applications that are dumping either post-processing or restart files as applications that do that tend to open the file, write a bunch of stuff from start to finish and close it or read a bunch of stuff from start to finish and close it. Two parameters govern the behavior of the silo VFD; the block size and the block count. The block size determines the size of individual blocks. All I/O requests will be issued in whole blocks. The block count determines the number of blocks the silo VFD is permitted to keep in memory at any one time. On BG/P class systems, good values are 1 Megabyte (2^{20}) block size and block count of 16 or 32. By default, the silo VFD uses a block size of 16 Kilobytes (2^{14}) and a block count also of 16. To specify parameters other than these default values, the Silo client will have to create and register an appropriate file options set, see *DBRegisterFileOptionsSet*.

DB_HDF5_MPIO and DB_HDF5_MPIOP
These have been removed from Silo as of version 4.10.3.

In Fortran, an integer represent the file's id is returned. That integer is then used as the database file id in all functions to read and write data from the file.

Note that regardless of what type of file is created, it can still be read on any machine.

See notes in the documentation on DBOpen regarding use of the DB_UNKNOWN driver type.

1.3.7 DBOpen()

- **Summary:** Open an existing Silo file.
- **C Signature:**

```
DBfile *DBOpen (char *name, int type, int mode)
```

- **Fortran Signature:**

```
integer function dbopen(name, lname, type, mode,
   dbid)
```

returns database file handle in dbid.

- **Arguments:**

Arg name	Description
name	Name of the file to open. Can be either an absolute or relative path.
type	The type of file to open. One of the predefined types, typically DB_UNKNOWN, DB_PDB, or DB_HDF5. However, there are other options as well as subtle but important issues in using them. So, read description, below for more details.
mode	The mode of the file to open. Pass DB_READ or DB_APPEND. Optionally, DB_APPEND can be OR'd with DB_PERF_OVER_COMPAT or DB_COMPAT_OVER_PERF (see *DBSetCompatibilityMode*. OR'ing of DB_READ with compatibility mode flags is not allowed.

- **Returned value:**

  DBOpen returns a DBfile pointer on success and a NULL on failure.

- **Description:**

  The DBOpen function opens an existing Silo file. If the file type passed here is DB_UNKNOWN, Silo will attempt to guess at the file type by iterating through the known types attempting to open the file with each driver until it succeeds. This iteration may incur a small performance penalty. In addition, use of DB_UNKNOWN can have other undesireable behavior described below. So, if at all possible, it is best to open using a specific type. See *DBGetDriverTypeFromPath*() for a function that uses cheap heuristics to determine the driver type given a candiate filename.

  When writing general purpose code to read Silo files and you cannot know for certain ahead of time what the correct driver to use is, there are a few options.

  First, you can iterate over the available driver ids, calling DBOpen() using each one until one of them succeds. But, that is exactly what the DB_UNKNOWN driver does so there is no need for a Silo client to have to write that code. In addition, if you have a specific preference of order of drivers, you can use DBSetUnknownDriverPriorities() to specify that ordering.

  Undesireable behavior with DB_UNKNOWN can occur when the specified file can be successfully opened using multiple available drivers and/or file options sets and it succceds with one or one using options the caller neither expected or intended and/or which offers poorer performance. See *DBSetUnknownDriverPriorities* for a way to specify the order of drivers tried by the DB_UNKNOWN driver.

  Indeed, in order to use a specific VFD (see *DBCreate*) in HDF5, it is necessary to pass the specific DB_HDF5_XXX argument in this call or to set the unknown driver priorities such that whatever specific HDF5 VFD(s) are desired are tried first before falling back to other, perhaps less desirable ones.

  The mode parameter allows a user to append to an existing Silo file. If a file is opened with a mode of DB_APPEND, the file will support write operations as well as read operations.

1.3.8 DBFlush()

- **Summary:**
- **C Signature:**

```
int DBFlush(DBfile *dbile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	the file to be flushed

- **Returned value:**

  Zero on success; -1 on failure.

- **Description:**

Flush any changes to a file to disk without having to actually close the file.

1.3.9 DBClose()

- **Summary:** Close a Silo database.
- **C Signature:**

```
int DBClose (DBfile *dbfile)
```

- **Fortran Signature:**

```
integer function dbclose(dbid)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.

- **Returned value:**

  DBClose returns zero on success and -1 on failure.

- **Description:**

  The DBClose function closes a Silo database.

  Note: For files produced by the HDF5 driver, the occasional bug in the Silo library can lead to situations where DBClose() might fail to actually close the file. There is logic in Silo to try to detect this if it is happening and then issue a warning or error message. However, a caller may not be paying attention to Silo function return values or error messages.

1.3.10 DBGetToc()

- **Summary:** Get the table of contents of a Silo database.
- **C Signature:**

```
DBtoc *DBGetToc (DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.

- **Returned value:**

  DBGetToc returns a pointer to a *DBtoc* structure on success and NULL on error.

- **Description:**

  The DBGetToc function returns a pointer to a *DBtoc* structure, which contains the names of the various Silo object contained in the Silo database. The returned pointer points into Silo private space and must not be modified or freed. Also, calls to DBSetDir will free the *DBtoc* structure, invalidating the pointer returned previously by DBGetToc.

1.3.11 File-level properties

There are a number of methods that control overall *behavior* either globally to the whole library or for a specific file. These are all documented in the *Global Library Behavior* section.

- *DBSetAllowOverwritesFile()*
- *DBGetAllowOverwritesFile()*
- *DBSetAllowEmptyObjectsFile()*
- *DBGetAllowEmptyObjectsFile()*
- *DBSetDataReadMask2File()*
- *DBGetDataReadMask2File()*
- *DBSetEnableChecksumsFile()*
- *DBGetEnableChecksumsFile()*
- *DBSetCompressionFile()*
- *DBGetCompressionFile()*
- *DBSetFriendlyHDF5NamesFile()*
- *DBGetFriendlyHDF5NamesFile()*
- *DBSetDeprecateWarningsFile()*
- *DBGetDeprecateWarningsFile()*

1.3.12 DBFileName()

- **Summary:**
- **C Signature:**

```
char const *DBFileName(DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`dbfile`	the Silo database file for which the name is to be queried.

- **Returned value:**

  Always succeeds. May return string containing "unknown".

1.3.13 `DBFileVersion()`

- **Summary:** Version of the Silo library used to create the specified file
- **C Signature:**

```
char const *DBFileVersion(DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file handle

- **Returned value:**

  A character string representation of the version number of the Silo library that was used to create the Silo file. The caller should **not** free the returned string.

- **Description:**

  Note, that this is distinct from (e.g. can be the same or different from) the version of the Silo library returned by the `DBVersion()` function.

  `DBFileVersion()`, here, returns the version of the Silo library that was used when `DBCreate()` was called on the specified file. `DBVersion()` returns the version of the Silo library the executable is currently linked with.

  Most often, these two will be the same. But, not always. Also note that although is possible that a single Silo file may have contents created within it from multiple versions of the Silo library, a call to this function will return *only* the version that was in use when `DBCreate()` was called; that is when the file was first created.

1.3.14 `DBFileVersionDigits()`

- **Summary:** Return integer digits of file version number
- **C Signature:**

```
int DBFileVersionDigits(const DBfile *dbfile,
    int *maj, int *min, int *pat, int *pre)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Silo database file handle
maj	Pointer to returned major version digit
min	Pointer to returned minor version digit
pat	Pointer to returned patch version digit
pre	Pointer to returned pre-release version digit (if any)

- **Returned value:**

  Zero on success. Negative value on failure.

1.3.15 DBFileVersionGE()

- **Summary:** Greater than or equal comparison for version of the Silo library a given file was created with
- **C Signature:**

```
int DBFileVersionGE(DBfile *dbfile, int Maj, int Min, int Pat)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file handle
Maj	Integer major version number
Min	Integer minor version number
Pat	Integer patch version number

- **Returned value:**

  One (1) if the version number of the library used to create the specified file is greater than or equal to the version number specified by Maj, Min, Pat arguments, zero (0) otherwise. A negative value is returned if a failure occurs.

1.3.16 DBVersionGEFileVersion()

- **Summary:** Compare library version with file version
- **C Signature:**

```
int DBVersionGEFileVersion(const DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Silo database file handle obtained with a call to DBOpen

- **Returned value:**

  Non-zero if the library version is greater than or equal to the file version. Zero otherwise.

1.3.17 DBSortObjectsByOffset()

- **Summary:** Sort list of object names by order of offset in the file
- **C Signature:**

```
int DBSortObjectsByOffset(DBfile *, int nobjs,
    const char *const *const obj_names, int *ordering)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
DBfile	Database file pointer.
nobjs	Number of object names in obj_names.
ordering	Returned integer array of relative order of occurence in the file of each object. For example, if ordering[i]==k, that means the object whose name is obj_names[i] occurs kth when the objects are ordered according to offset at which they exist in the file.

- **Returned value:**

  0 on succes; -1 on failure. The only possible reason for failure is if the HDF5 driver is being used to read the file and Silo is not compiled with HDF5 version 1.8 or later.

- **Description:**

  The intention of this function is to permit applications reading Silo files to order their reads in such a way that objects are read in the order in which they occur in the file. This can have a postive impact on I/O performance, particularly using a block-oriented VFD such as the Silo VFD as it can reduce and/or eliminate unnecessary block pre-emption. The degree to which `ordering` reads effects performance is not yet known.

1.3.18 DBMkDir()

- **Summary:** Create a new directory in a Silo file.
- **C Signature:**

```
int DBMkDir (DBfile *dbfile, char const *dirname)
```

- **Fortran Signature:**

```
integer function dbmkdir(dbid, dirname, ldirname, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
dirname	Name of the directory to create.

- **Returned value:**

  DBMkDir returns zero on success and -1 on failure.

- **Description:**

  The `DBMkDir` function creates a new directory in the Silo file as a child of the current directory (see *DBSetDir*). The directory name may be an absolute path name similar to `"/dir/subdir"`, or may be a relative path name similar to `"../../dir/subdir"`.

1.3.19 DBMkDirP()

- **Summary:** Create a new directory, as well as any necessary intervening directories, in a Silo file.
- **C Signature:**

```
int DBMkDirP(DBfile *dbfile, char const *dirname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`dirname`	Name of the directory to create.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The `DBMkDirP` function creates a new directory in the Silo file including making any necessary intervening directories. This is the Silo equivalent of Unix' `mkdir -p`. The directory name may be an absolute path name similar to `"/dir/subdir"`, or may be a relative path name similar to `"../../dir/subdir"`.

1.3.20 DBSetDir()

- **Summary:** Set the current directory within the Silo database.
- **C Signature:**

```
int DBSetDir (DBfile *dbfile, char const *pathname)
```

- **Fortran Signature:**

```
integer function dbsetdir(dbid, pathname, lpathname)
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`pathname`	Path name of the directory. This can be either an absolute or relative path name.

- **Returned value:**

  DBSetDir returns zero on success and -1 on failure.

- **Description:**

  The `DBSetDir` function sets the current directory within the given Silo database. Also, calls to `DBSetDir` will free the *DBtoc* structure, invalidating the pointer returned previously by `DBGetToc`. `DBGetToc` must be called again in order to obtain a pointer to the new directory's `DBtoc` structure.

1.3.21 DBGetDir()

- **Summary:** Get the name of the current directory.
- **C Signature:**

```
int DBGetDir (DBfile *dbfile, char *dirname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`dirname`	Returned current directory name. The caller must allocate space for the returned name. The maximum space used is `256` characters, including the `NULL` terminator.

- **Returned value:**

  DBGetDir returns zero on success and -1 on failure.

- **Description:**

  The `DBGetDir` function returns the name of the current directory.

1.3.22 DBCpDir()

- **Summary:** Copy a directory hierarchy from one Silo file to another.
- **C Signature:**

```
int DBCpDir(DBfile *srcFile, const char *srcDir,
    DBfile *dstFile, const char *dstDir)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
srcFile	Source database file pointer.
srcDir	Name of the directory within the source database file to copy.
dstFile	Destination database file pointer.
dstDir	Name of the top-level directory in the destination file. If an absolute path is given, then all components of the path except the last must already exist. Otherwise, the new directory is created relative to the current working directory in the file.

- **Returned value:**

  DBCpDir returns 0 on success, -1 on failure

- **Description:**

  `DBCpDir` copies an entire directory hierarchy from one Silo file to another.

  Note that this function is available only on the HDF5 driver and only if the Silo library has been compiled with HDF5 version 1.8 or later. This is because the implementation exploits functionality available only in versions of HDF5 1.8 and later.

1.3.23 DBCpListedObjects()

- **Summary:** Copy lists of objects from one Silo database to another
- **C Signature:**

```
int DBCpListedObjects(int nobjs,
    DBfile *srcDb, char const * const *srcObjList,
    DBfile *dstDb, char const * const *dstObjList)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
nobjs	The number of objects to be copied (e.g. number of strings in srcObjList)
srcDb	The Silo database to be used as the source of the copies
srcObjList	An array of nobj strings of the (path) names of objects to be copied. See description for interpretation of relative path names.
dstDb	The Silo database to be used as the destination of the copies.
dstObjList	[Optional] An optional array of nobj strings of the (path) names where the objects are to be copied in dstDb. If any entry in `dstObjList` is `NULL` or is a string of zero length, this indicates that object in the dstDb will have the same (path) name as the corresponding object (path) name given in `srcObjList`. If the entire `dstObjList` is `NULL`, then this is true for all objects. See description for interpretation of relative (path) names.

- **Returned value:**

  Returns 0 on success, -1 on failure

- **Description:**

  Note that this function is available only if both Silo databases are from the HDF5 driver and only if the Silo library has been compiled with HDF5 version 1.8 or later. This is because the implementation exploits functionality available only in versions of HDF5 1.8 and later.

  Directories required in the destination database to satisfy the path names given in the

  dstObjList are created as necessary. There is no need for the caller to pre-create any directories in the destination database.

  Relative path names in the `srcObjList` are interpreted relative to the source database's current working directory. Likewise, relative path names in the dstObjectList are interpreted relative to the destination databases's current working directory. Of course, if objects are specified using absolute path names, then copies will occur regardless of source or destination databases's current working directory.

  If an object specified in the `srcObjList` is itself a directory, then the entire directory tree rooted at that name will be copied to the destination database.

  If `dstObjList` is `NULL`, then it is assumed that all the objects to be copied will be copied to paths in the destination database that are intended to be the same as those in `srcObjList`. If `dstObjList` is non-`NULL`, but any given entry is either `NULL` or a string of zero length, than that object's destination name will be assumed the same as its equivalent `srcObjList` entry. Note, when using `NULL`s relative paths in `srcObjList` will appear in destination databases relative to the destination database's current working directory.

  If dstObjList[i] ends in a '/' character or identifies a directory that existed in the destination database either before `DBCpListedObjects` was called or that was created on behalf of a preceding object copy within the execution of DBCpListedObjects, then the source object will be copied to that directory with its original (source) name. This is equivalent to the behavior of the file system command `cp foo /gorfo/bar/` or `cp foo /gorfo/bar` when `bar` exists as a directory.

  Finally, users should be aware that if there are numeric architecture differences between the host where the source object data was produced and the host where this copy operation is being performed, then in all likelihood the destination copies of any floating point data may not match bit-for-bit with the source data. This is because data conversion may have been involved in the process of reading the data into memory and writing the copy back out.

  For example, suppose we have two databases…

  1. `DBfile *dbfile` ("dir.silo") containing…
     - `/ucd_dir/ucdmesh` (ucd mesh object)
     - `/tri_dir/trimesh` (ucd mesh object) <– current working directory
     - `/quad_dir/quadmesh` (quad mesh object)
  2. `DBfile *dbfile2` ("dir2.silo") containing…
     - `/tmp` <– current working directory

  And the following source and destination lists…

  - `char *srcObjs[] = {"trimesh", "../ucd_dir/ucdmesh", "/quad_dir/quadmesh", "trimesh"};`
  - `char *dstObjs[] = {"/tmp/foo/bar/gorfo", "../foogar", 0, "foo"};`

  Then, `DBCpListedObjects(4, dbfile, srcObjs, dbfile2, dstObjs)` does the following…

  1. Copies `trimesh` in `cwg` of `dbfile` to `/tmp/foo/bar/gorfo` in `dbfile2`
  2. Copies `../ucd_dir/ucdmesh` of `dbfile` to `/foogar` in `dbfile2`
  3. Copies `/quad_dir/quadmesh` to `cwg` (e.g. `/tmp`) `/tmp/quadmesh` in `dbfile2`

4. Copies `trimesh` in `cwg` of `dbfile` to `cwg/foo` (`/tmp/foo/trimesh` in dbfile2`

1.3.24 DBCp()

- **Summary:** Generalized copy function emulating unix `cp`
- **C Signature:**

```
int DBCp(char const *opts, DBfile *srcFile, DBfile *dstFile, ...)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`opts`	A space-separated options string.
`srcFile`	The file holding the source objects to be copied.
`dstFile`	The file holding the destination object paths into which source objects will be copied.
`...`	A `varargs` list of remaining arguments terminated in `DB_EOA` (end of arguments).

- **Return value:**

  0 on success. -1 on failure.

- **Description:**

  This is the Silo equivalent of a unix `cp` command obeying all unix semantics and applicable flags and works on either *above* any driver. This means the method can be used for PDB files, HDF5 files or even a mixture of drivers. In particular, it can be used to copy a whole file from one low-level driver to another.

  – Copy a single source object to a single destination object

  *
```
DBCp("", srcFile, dstFile, srcPATH, dstPATH, DB_EOA);`
```

  – Copy multiple source objects to single destination (dir) object

  *
```
DBCp("", srcFile, dstFile, srcPATH1, srcPATH2, ..., dstDIR, DB_EOA);`
```

  – Copy multiple source objects to multiple destination objects

  *
```
DBCp("-2", DBfile *srcFile, DBfile *dstFile,
        char const *srcPATH1, char const *dstPATH1,
        char const *srcPATH2, char const *dstPATH2,
        char const *srcPATH3, char const *dstPATH3, ..., DB_EOA);
```

  `srcFile` and `dstFile` may be the same Silo file. `srcFile` cannot be null. `dstFile` may be null in which case it is assumed same as `srcFile`. The argument list *must* be terminated by the `DB_EOA` sentinel. Just as for unix `cp`, the options and their meanings are…

  – `-R/-r`: recurse on any directory objects.

  – `-L/-P`: dereference links / never dereference links.

  – `-d`: preserve links.

  – `-s/-l`: don't actually copy, just sym/hard link (only possible when srcFile==dstFile).

There are some additional options specific to Silo's DBCp...

- -2: treat varargs list of args as src/dst path pairs and where any NULL dst is inferred to have same path as associated src except that relative paths are interpreted relative to dst file's cwg.
- -1: like -2 except caller passes only src paths. All dst paths are inferred to be same as associated src path. The dst file's cwg will then determine how any relative src paths are interpreted.
- -3 only 3 args follow the dstFile arg...
  - int N: number of objects in the following lists.
  - DBCAS_t srcPathNames: list of N source path names.
  - DBCAS_t dstPathNames: list of N destination path names.
  - In this case, a terminating DB_EOA is not necessary.
- -4: Like -3, except 3rd arg is treated as a single dst dir name.
  - int N: number of paths in srcPathNames.
  - DBCAS_t srcPathNames: list of N source path names.
  - char const *dstDIR: pre-existing destination dir path.
  - In this case, a terminating DB_EOA is not necessary.
- -5: Internal use only...like -4 except used only internally when DBCp recursively calls itself.

Other rules:

- If any src is a dir, then the operation is an error without -R/-r option.
- If cooresponding dst exists and is a dir, src is copied *into* (e.g. becomes a child of) dst.
- If cooresponding dst exists and is **not** a dir (e.g. is just a normal Silo object), then it is an error if src is not also the same kind of Silo object. The copy overwrites (destructively) dst. However, if the file space dst object occupies is smaller than that needed to copy src, behavior is indeterminate but may will result in the dst file (not just the dst object) being corrupted.

If none of the preceding numeric arguments are specified, then the varags list of args is treated as (default) where the last is a pre-existing destination directory and all the others are the paths of source objects to be copied into that directory.

Relative path names are interpreted relative to the current working directory of the associated (src or dst) file when DBCp is invoked.

In all the different ways this function can be invoked, there are really just two fundamentally different interpretations of the list(s) of names. Either each source path is paired also with a destination path or all source paths go into a single destination path which, just as for linux cp, must then also be a directory already present in the destination.

1.3.25 DBGrabDriver()

- **Summary:** Obtain the low-level driver file handle
- **C Signature:**

```
void *DBGrabDriver(DBfile *file)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.

- **Returned value:**

  A void pointer to the low-level driver's file handle on success. NULL(0) on failure.

- **Description:**

  This method is used to obtain the low-level driver's file handle. For example, one can use it to obtain the HDF5 hid_t returned from the H5Fopen/H5Fcreate call. The caller is responsible for casting the returned pointer to a pointer to the correct type. Use DBGetDriverType() to obtain information on the type of driver currently in use.

  When the low-level driver's file handle is grabbed, all Silo-level operations on file are prevented until file is UNgrabbed. For example, after a call to DBGrabDriver, calls to functions like DBPutQuadmesh or DBGetCurve will fail until the driver is UNgrabbed using DBUngrabDriver().

  Notes:

  As far as the integrity of a Silo file goes, grabbing is inherently dangerous. If the client is not careful, one can easily wind up corrupting the file for the Silo library (though all may be 'normal' for the underlying driver library). Therefore, to minimize the likelihood of corrupting the Silo file while it is grabbed, it is recommended that all operations with the low-level driver grabbed be confined to a separate sub-directory in the silo file. That is, one should not mix writing of Silo objects and low-level driver objects in the same directory. To achieve this, before grabbing, create the desired directory and descend into it using Silo's DBMkDir() and DBSetDir() functions. Then, grab the driver and do all the work with the low-level driver that is necessary. Finally, ungrab the driver and immediately ascend out of the directory using Silo's DBSetDir("..").

  For reasons described above, if problems occur on files that have been grabbed, users will likely be asked to reproduce the problem on a similar file that has **not** been grabbed to rule out possible corruption from grabbing.

1.3.26 DBUngrabDriver()

- **Summary:** Ungrab the low-level file driver
- **C Signature:**

```
int DBUngrabDriver(DBfile *file, const void *drvr_hndl)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
drvr_hndl	The low-level driver handle.

- **Returned value:**

  The driver type on success, DB_UNKNOWN on failure.

- **Description:**

  This function returns the Silo `file` to an ungrabbed state, permitting Silo calls to again proceed as normal.

1.3.27 DBGetDriverType()

- **Summary:** Get the type of driver for the specified file
- **C Signature:**

```
int DBGetDriverType(const DBfile *file)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`file`	A Silo database `file` handle.

- **Returned value:**

  `DB_UNKNOWN` for failure. Otherwise, the specified driver type is returned

- **Description:**

  This function returns the type of driver used for the specified `file`. If you want to ask this question without actually opening the file, use `DBGetDriverTypeFromPath`.

1.3.28 DBGetDriverTypeFromPath()

- **Summary:** Guess the driver type used by a file with the given pathname
- **C Signature:**

```
int DBGetDriverTypeFromPath(char const *path)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`path`	Path to a file on the file system

- **Returned value:**

  `DB_UNKNOWN` on failure to determine type. Otherwise, the driver type such as `DB_PDB` or `DB_HDF5`.

- **Description:**

  This function examines the first few bytes of the file for tell-tale signs of whether it is a PDB file or an HDF5 file.

  If it is a PDB file, it cannot distinguish between a file generated by `DB_PDB` driver and `DB_PDBP` (PDB Proper) driver. It will always return `DB_PDB` for a PDB file.

If the file is an HDF5, the function is currently not implemented to distiniguish between various HDF5 VFDs the file may have been generated with. It will always return `DB_HDF5` for an HDF5 file.

Note, this function will determine only whether the underlying file is a PDB or HDF5 file. It will not however, indicate whether the file is a PDB or HDF5 file that was indeed generated by Silo and is readable by Silo. See *DBInqFile* for a function that will indicate whether the file is indeed a Silo file. Note, however, that `DBInqFile` is a more expensive operation.

1.3.29 DBInqFile()

- **Summary:** Inquire if filename is a Silo file.
- **C Signature:**

```
int DBInqFile (char const *filename)
```

- **Fortran Signature:**

```
integer function dbinqfile(filename, lfilename, is_file)
```

- **Arguments:**

Arg name	Description
`filename`	Name of file.

- **Returned value:**

`DBInqFile` returns 0 if filename is not a Silo file, a positive number if filename is a Silo file, and a negative number if an error occurred.

- **Description:**

The `DBInqFile` function is mainly used for its return value, as seen above.

Prior to version 4.7.1 of the Silo library, this function could return false positives when the `filename` referred to a PDB file that was **not** created by Silo. The reason for this is that all this function really did was check whether or not `DBOpen` would succeed on the file.

Starting in version 4.7.1 of the Silo library, this function will attempt to count the number of Silo objects (not including directories) in the first non-empty directory it finds. If it cannot find any Silo objects in the file, it will return zero (0) indicating the file is **not** a Silo file.

Because very early versions of the Silo library did not store anything to a Silo file to distinguish it from a PDB file, it is conceivable that this function will return false negatives for very old, empty Silo files. But, that case should be rare.

Similar problems do not exist for HDF5 files because Silo's HDF5 driver has always stored information in the HDF5 file which helps to distinguish it as a Silo file.

1.3.30 DBInqFileHasObjects()

- **Summary:** Determine if an open file has any Silo objects
- **C Signature:**

```
int DBInqFileHasObjects(DBfile *dbfile)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	The Silo database file handle

- **Description:**

  Examine an open file for existence of any Silo objects.

1.3.31 _silolibinfo

- **Summary:** A character array written by Silo to root directory indicating the Silo library version number used to generate the file
- **C Signature:**

```
    int n;
    char vers[1024];
    sprintf(vers, "silo-4.6");
    n = strlen(vers);
    DBWrite(dbfile, "_silolibinfo", vers, &n, 1, DB_CHAR);
```

- **Description:**

  This is a simple array variable written at the root directory in a Silo file that contains the Silo library version string. It cannot be disabled.

1.3.32 _hdf5libinfo

- **Summary:** character array written by Silo to root directory indicating the HDF5 library version number used to generate the file
- **C Signature:**

```
    int n;
    char vers[1024];
    sprintf(vers, "hdf5-1.6.6");
    n = strlen(vers);
    DBWrite(dbfile, "_hdf5libinfo", vers, &n, 1, DB_CHAR);
```

- **Description:**

  This is a simple array variable written at the root directory in a Silo file that contains the HDF5 library version string. It cannot be disabled. Of course, it exists, only in files created with the HDF5 driver.

1.3.33 _was_grabbed

- **Summary:** single integer written by Silo to root directory whenever a Silo file has been grabbed.
- **C Signature:**

```
    int n=1;
    DBWrite(dbfile, "_was_grabbed", &n, &n, 1, DB_INT);
```

- **Description:**

  This is a simple array variable written at the root directory in a Silo whenever a Silo file has been grabbed by the `DBGrabDriver()` function. It cannot be disabled.

1.4 Meshes, Variables and Materials

If you are interested in learning how to deal with these objects in parallel, See *Multi-Block Objects and Parallel I/O*.

This section of the Silo API manual describes all the high-level Silo objects that are sufficiently self-describing as to be easily shared between a variety of applications.

Silo supports a variety of mesh types including simple 1D curves, structured meshes including block-structured Adaptive Mesh Refinement (AMR) meshes, point (or gridless) meshes consisting entirely of points, unstructured meshes consisting of the standard zoo of element types, fully arbitrary polyhedral meshes and Constructive Solid Geometry meshes described by boolean operations of primitive quadric surfaces.

In addition, Silo supports both piecewise constant (e.g. zone-centered) and piecewise-linear (e.g. node-centered) variables (e.g. fields) defined on these meshes. Silo also supports the decomposition of these meshes into materials (and material species) including cases where multiple materials are mixing within a single mesh element. Finally, Silo also supports the specification of expressions representing derived variables.

1.4.1 DBPutCurve()

- **Summary:** Write a curve object into a Silo file
- **C Signature:**

```
int DBPutCurve (DBfile *dbfile, char const *curvename,
    void const *xvals, void const *yvals, int datatype,
    int npoints, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputcurve(dbid, curvename, lcurvename, xvals,
   yvals, datatype, npoints, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
curvename	Name of the curve object
xvals	Array of length npoints containing the x-axis data values. Must be NULL when either DBOPT_XVARNAME or DBOPT_REFERENCE is used.
yvals	Array of length npoints containing the y-axis data values. Must be NULL when either DBOPT_YVARNAME or DBOPT_REFERENCE is used.
datatype	Data type of the xvals and yvals arrays. One of the predefined Silo types.
npoints	The number of points in the curve
optlist	Pointer to an option list structure containing additional information to be included in the compound array object written into the Silo file. Use NULL is there are no options.

- **Returned value:**

  DBPutCurve returns zero on success and -1 on failure.

- **Description:**

  The DBPutCurve function writes a curve object into a Silo file. A curve is a set of x/y points that describes a two-dimensional curve.

  Both the xvals and yvals arrays must have the same datatype.

  The following table describes the options accepted by this function. See the section titled "Using the Silo Option Parameter" for details on the use of this construct.

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value	0
DBOPT_XLABEL	char *	Label for the x-axis	NULL
DBOPT_YLABEL	char *	Label for the y-axis	NULL
DBOPT_XUNITS	char *	Character string defining the units for the x-axis	NULL
DBOPT_YUNITS	char *	Character string defining the units for the y-axis	NULL
DBOPT_XVARNAME	char *	Name of the domain (x) variable. This is the problem variable name, not the code variable name passed into the xvals argument.	NULL
DBOPT_YVARNAME	char *	Name of the domain (y) variable. This is problem variable name, not the code variable name passed into the yvals argument.	NULL
DBOPT_REFERENCE	char *	Name of the real curve object this object references. The name can take the form of "<file:/path-to-curve-object>" just as mesh names in the DBPutMultiMesh call. Note also that if this option is set, then the caller must pass NULL for both xvals and yvals arguments but must also pass valid information for all other object attributes including not only npoints and datatype but also any options.	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_COORDSYS	int	Coordinate system. One of: DB_CARTESIAN or DB_SPHERICAL	DB_CARTESIAN
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent "missing values" in the x or y data arrays	DB_MISSING_VALUE_NOT

In some cases, particularly when writing multi-part silo files from parallel clients, it is convenient to write curve data to something other than the "master" or "root" file. However, for a visualization tool to become aware of such objects, the tool is then required to traverse all objects in all the files of a multi-part file to find such objects. The DBOPT_REFERENCE option helps address this issue by permitting the writer to create knowledge of a curve object in the "master" or "root" file but put the actual curve object (the referenced object) wherever is most convenient. This output option would be useful for other Silo objects, meshes and variables, as well. However, it is currently only available for curve objects.

1.4.2 DBGetCurve()

- **Summary:** Read a curve from a Silo database.
- **C Signature:**

```
DBcurve *DBGetCurve (DBfile *dbfile, char const *curvename)
```

- **Fortran Signature:**

```
integer function dbgetcurve(dbid, curvename, lcurvename, maxpts,
   xvals, yvals, datatype, npts)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
curvename	Name of the curve to read.

- **Returned value:**

  Returns a pointer to a *DBcurve* structure on success and NULL on failure.

- **Description:**

  The DBGetCurve function allocates a *DBcurve* data structure, reads a curve from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.3 DBPutPointmesh()

- **Summary:** Write a point mesh object into a Silo file.
- **C Signature:**

```
int DBPutPointmesh (DBfile *dbfile, char const *name, int ndims,
    void const * const coords[], int nels,
    int datatype, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputpm(dbid, name, lname, ndims,
   x, y, z, nels, datatype, optlist_id,
   status)

void* x, y, z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the mesh.
`ndims`	Number of dimensions.
`coords`	Array of length `ndims` containing pointers to coordinate arrays.
`nels`	Number of elements (points) in mesh.
`datatype`	Datatype of the coordinate arrays. One of the predefined Silo data types.
`optlist`	Pointer to an option list structure containing additional information to be included in the mesh object written into the Silo file. Typically, this argument is `NULL`.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The `DBPutPointmesh` function accepts pointers to the coordinate arrays and is responsible for writing the mesh into a point-mesh object in the Silo file.

  A Silo point-mesh object contains all necessary information for describing a *point* mesh. This includes the coordinate arrays, the number of dimensions (1,2,3,...) and the number of points.

  The following table describes the options accepted by this function. See the section about *Options Lists* for details on the use of the `DBoptlist` construct.

  Optlist options:

Option Name	Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_XLABEL	char*	Character string defining the label associated with the X dimension.	NULL
DBOPT_YLABEL	char*	Character string defining the label associated with the Y dimension.	NULL
DBOPT_ZLABEL	char*	Character string defining the label associated with the Z dimension.	NULL
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_XUNITS	char*	Character string defining the units associated with the X dimension.	NULL
DBOPT_YUNITS	char*	Character string defining the units associated with the Y dimension.	NULL
DBOPT_ZUNITS	char*	Character string defining the units associated with the Z dimension.	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_MRGTREE_NAME	char*	Name of the mesh region grouping tree to be associated with this mesh.	NULL
DBOPT_NODENUM	void*	An array of length nnodes giving a global node number for each node in the mesh. By default, this array is treated as type int.	NULL
DBOPT_LLONGNZNUM	int	Indicates that the array passed for DBOPT_NODENUM option is of long long type instead of int.	0
DBOPT_LO_OFFSET	int	Zero-origin index of first non-ghost node. All points in the mesh before this one are considered ghost.	0
DBOPT_HI_OFFSET	int	Zero-origin index of last non-ghost node. All points in the mesh after this one are considered ghost.	nels-1
DBOPT_GHOST_NODE_LABELS	char*	Optional array of char values indicating the ghost labeling (DB_GHOSTTYPE_NOGHOST or DB_GHOSTTYPE_INTDUP) of each point	NULL
DBOPT_ALT_NODENUM_VARS	char**	A null terminated list of names of optional array(s) or DBpointvar objects indicating (multiple) alternative numbering(s) for nodes.	NULL

The following optlist options have been deprecated. Instead use MRG trees DBOPT_GROUPNUM|int|The group number to which this pointmesh belongs.|-1 (not in a group)

1.4.4 DBGetPointmesh()

- **Summary:** Read a point mesh from a Silo database.
- **C Signature:**

```
DBpointmesh *DBGetPointmesh (DBfile *dbfile, char const *meshname)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
meshname	Name of the mesh.

- **Returned value:**

  Returns a pointer to a *DBpointmesh* structure on success and NULL on failure.

- **Description:**

  The DBGetPointmesh function allocates a *DBpointmesh* data structure, reads a point mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.5 DBPutPointvar()

- **Summary:** Write a scalar/vector/tensor point variable object into a Silo file.
- **C Signature:**

```
int DBPutPointvar (DBfile *dbfile, char const *name,
    char const *meshname, int nvars, void const * cost vars[],
    int nels, int datatype, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable set.
meshname	Name of the associated point mesh.
nvars	Number of variables supplied in vars array.
vars	Array of length nvars containing pointers to value arrays.
nels	Number of elements (points) in variable.
datatype	Datatype of the value arrays. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutPointvar function accepts pointers to the value arrays and is responsible for writing the variables into a point-variable object in the Silo file.

  A Silo point-variable object contains all necessary information for describing a variable associated with a point mesh. This includes the number of arrays, the datatype of the variable, and the number of points. This function should be used when writing vector or tensor quantities. Otherwise, it is more convenient to use DBPutPointvar1.

  For tensor quantities, the question of ordering of tensor components arises. For symmetric tensor's Silo uses the Voigt Notation ordering. In 2D, this is T11, T22, T12. In 3D, this is T11, T22, T33, T23, T13, T12. For full tensor quantities, ordering is row by row starting with the top row.

  The following table describes the options accepted by this function. See the section about *Options Lists* for details on the use of the DBoptlist construct.

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_ASCII_LABEL	int	Indicate if the variable should be treated as single character, ascii values. A value of 1 indicates yes, 0 no.	0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_REGION_PNAMES	char **	A null pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be null and is used to indicate the end of the list of names. See *DBOPT_REGION_PNAMES*	NULL
DBOPT_CONSERVED	int	Indicates if the variable represents a physical quantity that must be conserved under various operations such as interpolation.	0
DBOPT_EXTENSIVE	int	Indicates if the variable represents a physical quantity that is extensive (as opposed to intensive). Note, while it is true that any conserved quantity is extensive, the converse is not true. By default and historically, all Silo variables are treated as intensive.	0
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent "missing values" variable data array(s). Default is DB_MISSING_VALUE_NOT_SET	DB_MISSING_VALUE_NOT

1.4.6 DBPutPointvar1()

- **Summary:** Write a scalar point variable object into a Silo file.
- **C Signature:**

```
int DBPutPointvar1 (DBfile *dbfile, char const *name,
    char const *meshname, void const *var, int nels, int datatype,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputpv1(dbid, name, lname, meshname,
   lmeshname, var, nels, datatype, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable.
meshname	Name of the associated point mesh.
var	Array containing data values for this variable.
nels	Number of elements (points) in variable.
datatype	Datatype of the variable. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutPointvar1 function accepts a value array and is responsible for writing the variable into a point-variable object in the Silo file.

  A Silo point-variable object contains all necessary information for describing a variable associated with a point mesh. This includes the number of arrays, the datatype of the variable, and the number of points. This function should be used when writing scalar quantities. To write vector or tensor quantities, one must use DBPutPointvar.

  See *DBPutPointvar* to a description of the options accepted by this function.

1.4.7 DBGetPointvar()

- **Summary:** Read a point variable from a Silo database.
- **C Signature:**

```
DBmeshvar *DBGetPointvar (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the variable.

- **Returned value:**

  Returns a pointer to a *DBmeshvar* structure on success and NULL on failure.

- **Description:**

  The DBGetPointvar function allocates a DBmeshvar data structure, reads a variable associated with a point mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.8 DBPutQuadmesh()

- **Summary:** Write a quad mesh object into a Silo file.
- **C Signature:**

```
int DBPutQuadmesh (DBfile *dbfile, char const *name,
    char const * const coordnames[], void const * const coords[],
    int dims[], int ndims, int datatype, int coordtype,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputqm(dbid, name, lname, xname,
   lxname, yname, lyname, zname, lzname, x,
   y, z, dims, ndims, datatype, coordtype,
   optlist_id, status)

void* x, y, z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
character* xname, yname, zname (if ndims<3, zname=0 ok, etc.)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the mesh.
coordnames	Array of length ndims containing pointers to the names to be provided when writing out the coordinate arrays. This parameter is currently ignored and can be set as NULL.
coords	Array of length ndims containing pointers to the coordinate arrays.
dims	Array of length ndims describing the dimensionality of the mesh. Each value in the dims array indicates the number of nodes contained in the mesh along that dimension. In order to specify a mesh with topological dimension lower than the geometric dimension, ndims should be the geometric dimension and the extra entries in the dims array provided here should be set to 1.
ndims	Number of geometric dimensions. Typically geometric and topological dimensions agree. Read the description for dealing with situations where this is not the case.
datatype	Datatype of the coordinate arrays. One of the predefined Silo data types.
coordtype	Coordinate array type. One of the predefined types: DB_COLLINEAR or DB_NONCOLLINEAR. Collinear coordinate arrays are always one-dimensional, regardless of the dimensionality of the mesh; non-collinear arrays have the same dimensionality as the mesh.
optlist	Pointer to an option list structure containing additional information to be included in the mesh object written into the Silo file. Typically, this argument is NULL.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutQuadmesh function accepts pointers to the coordinate arrays and is responsible for writing the mesh into a quad-mesh object in the Silo file.

  A Silo quad-mesh object contains all necessary information for describing a mesh. This includes the coordinate arrays, the rank of the mesh (1,2,3,...) and the type (collinear or non-collinear). In addition, other information is useful and is therefore optionally included (row-major indicator, time and cycle of mesh, offsets to *real* zones, plus coordinate system type.)

Typically, the number of geometric dimensions (e.g. size of coordinate tuple) and topological dimensions (e.g. dimension of elements shapes the mesh) agree. For example, this function is typically used to define a 3D arrangement of hexahedra or a 2D arrangement of quadrilaterals. However, this function can also be used to define a surface of quadrilaterals embedded in 3-space or a path of line segments embedded in 2- or 3-space. In these less common cases, the topological dimension is lower than the geometric dimension. The correct way to use this function to define such meshes is to use the `ndims` argument to specify the number of geometric dimensions and then to set those entries in the `dims` array that represent extra dimensions to one. For example, to specify a mesh of quadrilaterals in 3-space, set `ndims` to 3 but set dims[2] to 1. To specify a mesh of lines defining a path embedded in 3-space, `ndims` would again be 3 but dims[1] and dims[2] would both be 1. In fact, this works in general. For N geometric dimensions and N-k topological dimensions, set ndims=N and dims[N-1-k]...dims[N-1] to 1.

The following table describes the options accepted by this function. See the section about *Options Lists* for details on the use of the `DBoptlist` construct.

Optlist options:

Option Name	Data Type	Option Meaning	Default Value
DBOPT_COORDSYS	int	Coordinate system. One of:DB_CARTESIAN, DB_CYLINDRICAL, DB_SPHERICAL, DB_NUMERICAL, or DB_OTHER	DB_OTHER
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_FACETYPE	int	Zone face type. One of the predefined types: DB_RECTILINEAR or DB_CURVILINEAR	DB_RECTILINEAR
DBOPT_HI_OFFSET	int *	Array of length ndims which defines zero-origin offsets from the last node for the ending index along each dimension.	{0,0,...}
DBOPT_LO_OFFSET	int *	Array of ndims which defines zero-origin offsets from the first node for the starting index along each dimension.	{0,0,...}
DBOPT_XLABEL	char*	Character string defining the label associated with the X dimension	NULL
DBOPT_YLABEL	char*	Character string defining the label associated with the Y dimension	NULL
DBOPT_ZLABEL	char*	Character string defining the label associated with the Z dimension	NULL
DBOPT_MAJORORDER	int	Indicator for row-major (0) or column-major (1) storage for multidimensional arrays.	0
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_PLANAR	int	Planar value. One of: DB_AREA or DB_VOLUME	DB_OTHER
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	doubleProblem time value.		0.0
DBOPT_XUNITS	char*	Character string defining the units associated with the X dimension	NULL
DBOPT_YUNITS	char*	Character string defining the units associated with the Y dimension	NULL
DBOPT_ZUNITS	char*	Character string defining the units associated with the Z dimension	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_BASEINDEX	int[3]	Indicate the indices of the mesh within its group.	0,0,0
DBOPT_MRGTREE_NAME	char*	Name of the mesh region grouping tree to be associated with this mesh	NULL
DBOPT_GHOST_NODE_LABELS	char*	Optional array of char values indicating the ghost labeling DB_GHOSTTYPE_NOGHOST DB_GHOSTTYPE_INTDUP) of each nodeNULL	
DBOPT_GHOST_ZONE_LABELS	char*	Optional array of char values indicating the ghost labeling DB_GHOSTTYPE_NOGHOST DB_GHOSTTYPE_INTDUP) of each zoneNULL	
DBOPT_ALT_NODENUM_VARS	char**	A null terminated list of names of optional array(s) or DBquadvar objects indicating (multiple) alternative numbering(s) for nodesNULL	
DBOPT_ALT_ZONENUM_VARS	char**	A null terminated list of names of optional array(s) or DBquadvar objects indicating (multiple) alternative numbering(s) for zonesNULL	
The following options have been deprecated. Use MRG trees instead			
DBOPT_GROUPNUM	int	The group number to which this quadmesh belongs.	-1 (not in a group)

The options DB_LO_OFFSET and DB_HI_OFFSET should be used if the mesh being described uses the notion of "phoney" zones (i.e., some zones should be ignored.) For example, if a 2-D mesh had designated the first column and row, and the last two columns and rows as "phoney", then we would use: lo_off = {1,1} and hi_off = {2,2}.

1.4.9 DBGetQuadmesh()

- **Summary:** Read a Quad mesh from a Silo database.
- **C Signature:**

```
DBquadmesh *DBGetQuadmesh (DBfile *dbfile, char const *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
meshname	Name of the mesh.

- **Returned value:**

  Returns a pointer to a *DBquadmesh* structure on success and NULL on failure.

- **Description:**

  The DBGetQuadmesh function allocates a *DBquadmesh* data structure, reads a quadrilateral mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.10 DBPutQuadvar()

- **Summary:** Write a scalar/vector/tensor quad variable object into a Silo file.
- **C Signature:**

```
int DBPutQuadvar (DBfile *dbfile, char const *name,
    char const *meshname, int nvars,
    char const * const varnames[], void const * const vars[],
    int dims[], int ndims, void const * const mixvars[],
    int mixlen, int datatype, int centering,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputqv(dbid, vname, lvname, mname,
   lmname, nvars, varnames, lvarnames, vars, dims,
   ndims, mixvar, mixlen, datatype, centering, optlist_id,
   status)
```

varnames contains the names of the variables either in a matrix of characters form (if fortran2DStrLen is non-zero) or in a vector of characters form (if fortran2DStrLen is zero) with the varnames length(s) being found in the lvarnames integer array, var is essentially a matrix of size x where var-size is determined by dims

and ndims. The first *row* of the var matrix is the first component of the quadvar. The second *row* of the var matrix goes out as the second component of the quadvar, etc.

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable.
meshname	Name of the mesh associated with this variable (written with DBPutQuadmesh or DBPutUcdmesh). If no association is to be made, this value should be NULL.
nvars	Number of sub-variables which comprise this variable. For a scalar array, this is one. If writing a vector quantity, however, this would be two for a 2-D vector and three for a 3-D vector.
varnames	Array of length nvars containing pointers to character strings defining the names associated with each sub-variable.
vars	Array of length nvars containing pointers to arrays defining the values associated with each subvariable. For true edge- or face-centering (as opposed to DB_EDGECENT centering when ndims is 1 and DB_FACECENT centering when ndims is 2), each pointer here should point to an array that holds ndims sub-arrays, one for each of the i-, j-, k-oriented edges or i-, j-, k-intercepting faces, respectively. Read the description for more details.
dims	Array of length ndims which describes the dimensionality of the data stored in the vars arrays. For DB_NODECENT centering, this array holds the number of nodes in each dimension. For DB_ZONECENT centering, DB_EDGECENT centering when ndims is 1 and DB_FACECENT centering when ndims is 2, this array holds the number of zones in each dimension. Otherwise, for DB_EDGECENT and DB_FACECENT centering, this array should hold the number of nodes in each dimension.
ndims	Number of dimensions.
mixvars	Array of length nvars containing pointers to arrays defining the mixed-data values associated with each subvariable. If no mixed values are present, this should be NULL.
mixlen	Length of mixed data arrays, if provided.
datatype	Datatype of the variable. One of the predefined Silo data types.
centering	Centering of the subvariables on the associated mesh. One of the predefined types:DB_NODECENT, DB_EDGECENT, DB_FACECENT or DB_ZONECENT. Note that DB_EDGECENT centering on a 1D mesh is treated identically to DB_ZONECENT centering. Likewise for DB_FACECENT centering on a 2D mesh.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutQuadvar function writes a variable associated with a quad mesh into a Silo file. A quad-var object contains the variable values.

  For node- (or zone-) centered data, the question of which value in the vars array goes with which node (or zone) is determined implicitly by a one-to-one correspondence with the multi-dimensional array list of nodes (or zones) defined by the logical indexing for the associated mesh's nodes (or zones).

  Edge- and face-centered data require a little more explanation. We can group edges according to their logical orientation. In a 2D mesh of Nx by Ny nodes, there are (Nx-1)Ny i-oriented edges and Nx(Ny-1) j-oriented edges. Likewise, in a 3D mesh of Nx by Ny by Nz nodes, there are

  (Nx-1)NyNz i-oriented edges, Nx(Ny-1)Nz, j-oriented edges and NxNy(Nz-1) k-oriented edges. Each group of edges is almost the same size as a normal node-centered variable. So, for conceptual convenience we in fact

treat them that way and treat the extra slots in them as phony data. So, in the case of edge-centered data, each of the pointers in the `vars` argument to `DBPutQuadvar` is interpreted to point to an array that is `ndims` times the product of nodal sizes (NxNyNz). The first part of the array (of size NxNy nodes for 2D or NxNyNz nodes for 3D) holds the i-oriented edge data, the next part the j-oriented edge data, etc.

A similar approach is used for face centered data. In a 3D mesh of Nx by Ny by Nz nodes, there are Nx(Ny-1)(Nz-1) i-intercepting faces, (Nx-1)Ny(Nz-1) j-intercepting faces and (Nx-1)(Ny-1)Nz k-intercepting faces. Again, just as for edge-centered data, each pointer in the `vars` array is interpreted to point to an array that is `ndims` times the product of nodal sizes. The first part holds the i-intercepting face data, the next part the j-interception face data, etc.

Unlike node- and zone-centered data, there does not necessarily exist in Silo an explicit list of edges or faces. As an aside, the `DBPutFacelist` call is really for writing the external faces of a mesh so that a downstream visualization tool need not have to compute them when it displays the mesh. Now, requiring the caller to create explicit lists of edges and/or faces in order to handle edge- or face-centered data results in unnecessary additional data being written to a Silo file. This increases file size as well as the time to write and read the file. To avoid this, we rely upon implicit lists of edges and faces.

Finally, since the zones of a one dimensional mesh are basically edges, the case of `DB_EDGECENT centering` for a one dimensional mesh is treated identically to the `DB_ZONECENT` case. Likewise, since the zones of a two dimensional mesh are basically faces, the `DB_FACECENT centering` for a two dimensional mesh is treated identically to the `DB_ZONECENT` case.

Other information can also be included. This function is useful for writing vector and tensor fields, whereas the companion function, DBPutQuadvar1, is appropriate for writing scalar fields.

For tensor quantities, the question of ordering of tensor components arises. For symmetric tensor's Silo uses the Voigt Notation ordering. In 2D, this is `T11`, `T22`, `T12`. In 3D, this is `T11`, `T22`, `T33`, `T23`, `T13`, `T12`. For full tensor quantities, ordering is row by row starting with the top row.

Notes:

The following table describes the options accepted by this function. See the section about *Options Lists* for details on the use of the `DBoptlist` construct.

Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_COORDSYS	int	Coordinate system. One of:DB_CARTESIAN, DB_CYLINDRICAL, DB_SPHERICAL, DB_NUMERICAL, or DB_OTHER	DB_OTHER
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_FACETYPE	int	The face type. One of the predefined types: DB_RECTILINEAR or DB_CURVILINEAR	DB_RECTILINEAR
DBOPT_LABEL	char*	Character string defining the label associated with this variable	NULL
DBOPT_MAJORORDER	int	Indicator for row-major (0) or column-major (1) storage for multidimensional arrays.	0
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_UNITS	char*	Character string defining the units associated with this variable	NULL
DBOPT_USESPECMF	int	Boolean DB_OFF orDB_ON) value specifying whether or not to weight the variable by the species mass fraction when using material species data	DB_OFF
DBOPT_ASCII_LABEL	int	Indicate if the variable should be treated as single character, ascii values. A value of 1 indicates yes, 0 no.	0
DBOPT_CONSERVED	int	Indicates if the variable represents a physical quantity that must be conserved under various operations such as interpolation.	0
DBOPT_EXTENSIVE	int	Indicates if the variable represents a physical quantity that is extensive (as opposed to intensive). Note, while it is true that any conserved quantity is extensive, the converse is not true. By default and historically, all Silo variables are treated as intensive.	0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_REGION_PNAMES	char**	Pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be null and is used to indicate the end of the list of names. See *DBOPT_REGION_PNAMES*	NULL
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent "missing values" variable data array(s). Default isDB_MISSING_VALUE_NOT_SET	DB_MISSING_VALUE_NOT

1.4.11 DBPutQuadvar1()

- **Summary:** Write a scalar quad variable object into a Silo file.
- **C Signature:**

```
int DBPutQuadvar1 (DBfile *dbfile, char const *name,
    char const *meshname, void const *var, int const dims[],
    int ndims, void const *mixvar, int mixlen, int datatype,
    int centering, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputqv1(dbid, name, lname, meshname,
   lmeshname, var, dims, ndims, mixvar, mixlen,
   datatype, centering, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable.
meshname	Name of the mesh associated with this variable (written with DBPutQuadmesh or DBPutUcdmesh.) If no association is to be made, this value should be NULL.
var	Array defining the values associated with this variable. For true edge- or face-centering (as opposed to DB_EDGECENT centering when ndims is 1 and DB_FACECENT centering when ndims is 2), each pointer here should point to an array that holds ndims sub-arrays, one for each of the i-, j-, k-oriented edges or i-, j-, k-intercepting faces, respectively. Read the description for DBPutQuadvar more details.
dims	Array of length ndims which describes the dimensionality of the data stored in the var array. For DB_NODECENT centering, this array holds the number of nodes in each dimension. For DB_ZONECENT centering, DB_EDGECENT centering when ndims is 1 and DB_FACECENT centering when ndims is 2, this array holds the number of zones in each dimension. Otherwise, for DB_EDGECENT and DB_FACECENT centering, this array should hold the number of nodes in each dimension.
ndims	Number of dimensions.
mixvar	Array defining the mixed-data values associated with this variable. If no mixed values are present, this should be NULL.
mixlen	Length of mixed data arrays, if provided.
datatype	Datatype of sub-variables. One of the predefined Silo data types.
centering	Centering of the subvariables on the associated mesh. One of the predefined types:DB_NODECENT, DB_EDGECENT, DB_FACECENT or DB_ZONECENT. Note that DB_EDGECENT centering on a 1D mesh is treated identically to DB_ZONECENT centering. Likewise for DB_FACECENT centering on a 2D mesh.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutQuadvar1 function writes a scalar variable associated with a quad mesh into a Silo file. A quad-var object contains the variable values, plus the name of the associated quad-mesh. Other information can also be included. This function should be used for writing scalar fields, and its companion function, DBPutQuadvar, should be used for writing vector and tensor fields.

  For edge- and face-centered data, please refer to the description for DBPutQuadvar for a more detailed explanation.

  Notes:

  See *DBPutQuadvar* for a description of options accepted by this function.

1.4.12 DBGetQuadvar()

- **Summary:** Read a Quad mesh variable from a Silo database.
- **C Signature:**

```
DBquadvar *DBGetQuadvar (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the variable.

- **Returned value:**

  Returns a pointer to a *DBquadvar* structure on success and NULL on failure.

- **Description:**

  The DBGetQuadvar function allocates a *DBquadvar* data structure, reads a variable associated with a quadrilateral mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.13 DBPutUcdmesh()

- **Summary:** Write a UCD mesh object into a Silo file.
- **C Signature:**

```
int DBPutUcdmesh (DBfile *dbfile, char const *name, int ndims,
    char const * const coordnames[], void const * const coords[],
    int nnodes, int nzones, char const *zonel_name,
    char const *facel_name, int datatype,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputum(dbid, name, lname, ndims,
   x, y, z, xname, lxname, yname,
   lyname, zname, lzname, datatype, nnodes nzones, zonel_name,
   lzonel_name, facel_name, lfacel_name, optlist_id, status)

void *x,y,z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
character* xname,yname,zname (same rules)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the mesh.
ndims	Number of spatial dimensions represented by this UCD mesh.
coordnames	Array of length ndims containing pointers to the names to be provided when writing out the coordinate arrays. This parameter is currently ignored and can be set as NULL.
coords	Array of length ndims containing pointers to the coordinate arrays.
nnodes	Number of nodes in this UCD mesh.
nzones	Number of zones in this UCD mesh.
zonelist	Name of the zonelist structure associated with this variable [written with DBPutZonelist]. If no association is to be made or if the mesh is composed solely of arbitrary, polyhedral elements, this value should be NULL. If a polyhedral-zonelist is to be associated with the mesh, do not pass the name of the polyhedral-zonelist here. Instead, use the DBOPT_PHZONELIST option described below. For more information on arbitrary, polyhedral zonelists, see below and also see the documentation for DBPutPHZonelist.
facelist	Name of the facelist structure associated with this variable [written with DBPutFacelist]. If no association is to be made, this value should be NULL.
datatype	Datatype of the coordinate arrays. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the mesh object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use NULL for this argument.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutUcdmesh function accepts pointers to the coordinate arrays and is responsible for writing the mesh into a UCD mesh object in the Silo file.

  A Silo UCD mesh object contains all necessary information for describing a mesh. This includes the coordinate arrays, the rank of the mesh (1,2,3,...) and the type (collinear or non-collinear.) In addition, other information is useful and is therefore included (time and cycle of mesh, plus coordinate system type).

  A Silo UCD mesh may be composed of either zoo-type elements or arbitrary, polyhedral elements or a mixture of both zoo-type and arbitrary, polyhedral elements. The zonelist (connectivity) information for zoo-type elements is written with a call to DBPutZonelist. When there are only zoo-type elements in the mesh, this is the only zonelist information associated with the mesh. However, the caller can optionally specify the name of an arbitrary, polyhedral zonelist written with a call to DBPutPHZonelist using the DBOPT_PHZONELIST option. If the mesh consists solely of arbitrary, polyhedral elements, the only zonelist associated with the mesh will be the one written with the call to DBPutPHZonelist.

  When a mesh is composed of both zoo-type elements and polyhedral elements, it is assumed that all the zoo-type elements come first in the mesh followed by all the polyhedral elements. This has implications for any DBPutUcdvar calls made on such a mesh. For zone-centered data, the variable array should be organized so that values corresponding to zoo-type zones come first followed by values corresponding to polyhedral zones. Also, since both the zoo-type zonelist and the polyhedral zonelist support hi- and lo- offsets for ghost zones, the ghost-zones of a mesh may consist of zoo-type or polyhedral zones or a mixture of both.

  Notes:

  See *DBCalcExternalFacelist* or "DBCalcExternalFacelist2" on page 2-230 for an automated way of computing the facelist needed for this call.

  The order in which nodes are defined in the zonelist is important, especially for 3D cells. Nodes defining a 2D

Fig. 1: Example usage of UCD zonelist and external facelist variables.

cell should be supplied in either clockwise or counterclockwise order around the cell. The node, edge and face ordering and orientations for the predefined 3D cell types are illustrated below.

Fig. 2: Node, edge and face ordering for zoo-type UCD zone shapes.

Given the node ordering in the left-most column, there is indeed an algorithm for determining the other orderings for each cell type.

For edges, each edge is identified by a pair of integer indices; the first being the "tail" of an arrow oriented along the edge and the second being the "head" with the smaller node index always placed first (at the tail). Next, the ordering of edges is akin to a lexicographic ordering of these pairs of integers. This means that we start with the lowest node number of a cell shape, zero, and find all edges with node zero as one of the points on the edge. Each such edge will have zero as its tail. Since they all start with node 0 as the tail, we order these edges from smallest to largest "head" node. Then we go to the next lowest node number on the cell that has edges that have yet to have been placed in the ordering. We find all the edges from that node (that have not already been placed in the ordering) from smallest to largest "head" node. We continue this process until all the edges on the cell have been placed in the ordering.

For faces, a similar algorithm is used. Starting with the lowest numbered node on a face, we enumerate the nodes over a face using the right hand rule for the normal to the face pointing away from the innards of the cell. When one places the thumb of the right hand in the direction of this normal, the direction of the fingers curling around it identify the direction we go to identify the nodes of the face. Just as for edges, we start identifying faces for the lowest numbered node of the cell (0). We find all faces that share this node. Of these, the face that enumerates the next lowest node number as we traverse the nodes using the right hand rule, is placed first in the ordering. Then, the face that has the next lowest node number and so on.

An example using arbitrary polyhedrons for some zones is illustrated, below. The nodes of a `DB_ZONETYPE_POLYHEDRON` are specified in the following fashion: First specify the number of faces in the polyhedron. Then, for each face, specify the number of nodes in the face followed by the nodes that make up the face. The nodes should be ordered such that they are numbered in a counter-clockwise fashion when viewed from the outside (e.g. right-hand rules yields an outward facing normal). For a fully arbitrarily connected mesh, see `DBPutPHZonelist()`. In addition, for a sequence of consecutive zones of type `DB_ZONETYPE_POLYHEDRON` in a zonelist, the shapesize entry is taken to be the sum of all the associated positions occupied in the nodelist data. So, for the example in Figure 0-3 on page 106, the shapesize entry for the `DB_ZONETYPE_POLYEDRON` segment of the zonelist is '53' because for the two arbitrary polyhedral zones in the zonelist, 53 positions in the nodelist array are used.

Fig. 3: align: "center" alt: "Example usage of UCD zonelist combining a hex and 2 polyhedra."
Example usage of UCD zonelist combining a hex and 2 polyhedra.

This example is intended to illustrate the representation of arbitrary polyhedra. So, although the two polyhedra represent a hex and pyramid which would ordinarily be handled just fine by a 'normal' zonelist, they are expressed using arbitrary connectivity here.

The following table describes the options accepted by this function:

Optlist options:

Option Name	Data Type	Option Meaning	Default Value
DBOPT_COORDSYS	int	Coordinate system. One of:DB_CARTESIAN, DB_CYLINDRICAL, DB_SPHERICAL, DB_NUMERICAL, or DB_OTHER	DB_OTHER
DBOPT_NODENUM	void*	An array of length nnodes giving a global node number for each node in the mesh. By default, this array is treated as type int	NULL
DBOPT_LLONGNZNUM	int	Indicates that the array passed for DBOPT_NODENUM option is of long long type instead of int.	0
DBOPT_CYCLE	int	Problem cycle value	0
DBOPT_FACETYPE	int	Zone face type. One of the predefined types: DB_RECTILINEAR or DB_CURVILINEAR	DB_RECTILINEAR
DBOPT_XLABEL	char*	Character string defining the label associated with the X dimension	NULL
DBOPT_YLABEL	char*	Character string defining the label associated with the Y dimension	NULL
DBOPT_ZLABEL	char*	Character string defining the label associated with the Z dimension	NULL
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_PLANAR	int	Planar value. One of: DB_AREA or DB_VOLUME	DB_NONE
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_XUNITS	char*	Character string defining the units associated with the X dimension	NULL
DBOPT_YUNITS	char*	Character string defining the units associated with the Y dimension	NULL
DBOPT_ZUNITS	char*	Character string defining the units associated with the Z dimension	NULL
DBOPT_PHZONELIST	char*	Character string holding the name for a polyhedral zonelist object to be associated with the mesh	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_MRGTREE_NAME	char*	Name of the mesh region grouping tree to be associated with this mesh	NULL
DBOPT_TOPO_DIM	int	Used to indicate the topological dimension of the mesh apart from its spatial dimension.	-1 (not specified)
DBOPT_TV_CONNECTIVITY	int	A non-zero value indicates that the connectivity of the mesh varies with time	0
DBOPT_DISJOINT_MODE	int	Indicates if any elements in the mesh are disjoint. There are two possible modes. One is DB_ABUTTING indicating that elements abut spatially but actually reference different node ids (but spatially equivalent nodal positions) in the node list. The other is DB_FLOATING where elements neither share nodes in the nodelist nor abut spatially	DB_NONE
DBOPT_GHOST_NODE_LABELS	char*	Optional array of char values indicating the ghost labeling DB_GHOSTTYPE_NOGHOST or DB_GHOSTTYPE_INTDUP) of each point	NULL
DBOPT_ALT_NODENUM_VARS	char*	Null terminated list of names of optional array(s) or DBpointvar objects indicating (multiple) alternative numbering(s) for nodes	NULL
The following options have been deprecated. Use MRG trees instead			
DBOPT_GROUPNUM	int	The group number to which this quadmesh belongs.	-1 (not in a group)

1.4.14 DBPutUcdsubmesh()

- **Summary:** Write a subset of a parent, ucd mesh, to a Silo file
- **C Signature:**

```
int DBPutUcdsubmesh(DBfile *file, const char *name,
    const char *parentmesh, int nzones, const char *zlname,
    const char *flname, DBoptlist const *opts)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
name	The name of the ucd submesh object to create.
parentmesh	The name of the parent ucd mesh this submesh is a portion of.
nzones	The number of zones in this submesh.
zlname	The name of the zonelist object.
fl	[OPT] The name of the facelist object.
opts	Additional options.

- **Returned value:**

  A positive number on success; -1 on failure

- **Description:**

  Do not use this method.

  It is an extremely limited, inefficient and soon to be retired way of trying to define subsets of a ucd mesh. Instead, use a Mesh Region Grouping (MRG) tree. See *DBMakeMrgtree*.

1.4.15 DBGetUcdmesh()

- **Summary:** Read a UCD mesh from a Silo database.
- **C Signature:**

```
DBucdmesh *DBGetUcdmesh (DBfile *dbfile, char const *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
meshname	Name of the mesh.

- **Returned value:**

  Rreturns a pointer to a *DBucdmesh* structure on success and NULL on failure.

- **Description:**

  The DBGetUcdmesh function allocates a *DBucdmesh* data structure, reads a UCD mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.16 DBPutZonelist()

- **Summary:** Write a zonelist object into a Silo file.
- **C Signature:**

```
int DBPutZonelist (DBfile *dbfile, char const *name, int nzones,
    int ndims, int const nodelist[], int lnodelist, int origin,
    int const shapesize[], int const shapecnt[], int nshapes)
```

- **Fortran Signature:**

```
integer function dbputzl(dbid, name, lname, nzones,
   ndims, nodelist, lnodelist, origin, shapesize, shapecnt,
   nshapes, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the zonelist structure.
nzones	Number of zones in associated mesh.
ndims	Number of spatial dimensions represented by associated mesh.
nodelist	Array of length lnodelist containing node indices describing mesh zones.
lnodelist	Length of nodelist array.
origin	Origin for indices in the nodelist array. Should be zero or one.
shapesize	Array of length nshapes containing the number of nodes used by each zone shape.
shapecnt	Array of length nshapes containing the number of zones having each shape.
nshapes	Number of zone shapes.

- **Returned value:**

  Returns zero on success or -1 on failure.

- **Description:**

  Do not use this method. Use DBPutZonelist2() instead.

  The DBPutZonelist function writes a zonelist object into a Silo file. The name assigned to this object can in turn be used as the zonel_name parameter to the DBPutUcdmesh function.

  See *DBPutUcdmesh* for a full description of the zonelist data structures.

1.4.17 DBPutZonelist2()

- **Summary:** Write a zonelist object containing ghost zones into a Silo file.
- **C Signature:**

```
int DBPutZonelist2 (DBfile *dbfile, char const *name, int nzones,
    int ndims, int const nodelist[], int lnodelist, int origin,
    int lo_offset, int hi_offset, int const shapetype[],
    int const shapesize[], int const shapecnt[], int nshapes,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputzl2(dbid, name, lname, nzones,
   ndims, nodelist, lnodelist, origin, lo_offset, hi_offset,
   shapetype, shapesize, shapecnt, nshapes, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the zonelist structure.
nzones	Number of zones in associated mesh.
ndims	Number of spatial dimensions represented by associated mesh.
nodelist	Array of length lnodelist containing node indices describing mesh zones.
lnodelist	Length of nodelist array.
origin	Origin for indices in the nodelist array. Should be zero or one.
lo_offset	The number of ghost zones at the beginning of the nodelist.
hi_offset	The number of ghost zones at the end of the nodelist.
shapetype	Array of length nshapes containing the type of each zone shape. See description below.
shapesize	Array of length nshapes containing the number of nodes used by each zone shape.
shapecnt	Array of length nshapes containing the number of zones having each shape.
nshapes	Number of zone shapes.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use NULL for this argument.

- **Returned value:**

  Returns zero on success or -1 on failure.

- **Description:**

  The DBPutZonelist2 function writes a zonelist object into a Silo file. The name assigned to this object can in turn be used as the zonel_name parameter to the DBPutUcdmesh function.

  The allowed shape types are described in the following table:

  Optlist options:

Type	Description
DB_ZONETYPE_BEAM	A line segment
DB_ZONETYPE_POLYGON	A polygon where nodes are enumerated to form a polygon
DB_ZONETYPE_TRIANGLE	A triangle
DB_ZONETYPE_QUAD	A quadrilateral
DB_ZONETYPE_POLYHEDRON	A polyhedron with nodes enumerated to form faces and faces are enumerated to form a polyhedron
DB_ZONETYPE_TET	A tetrahedron
DB_ZONETYPE_PYRAMID	A pyramid
DB_ZONETYPE_PRISM	A prism
DB_ZONETYPE_HEX	A hexahedron

Notes:

The following table describes the options accepted by this function:

Option Name	Data Type	Option Meaning	Default Value
DBOPT_ZONENUM	void*	Array of global zone numbers, one per zone in this zonelist. By default, this is assumed to be of type int	NULL
DBOPT_LLONGNZNUM	int	Indicates that the array passed for DBOPT_ZONENUM option is of long long type instead of int.	0
DBOPT_GHOST_ZONE_LABELS	char*	Optional array of char values indicating the ghost labeling DB_GHOSTTYPE_NOGHOST or DB_GHOSTTYPE_INTDUP) of each zone	NULL
DBOPT_ALT_ZONENUM_VARS	char**	A null terminated list of names of optional array(s) or DBucdvar objects indicating (multiple) alternative numbering(s) for zones	NULL

For a description of how the nodes for the allowed shapes are enumerated, see *DBPutUcdmesh*.

1.4.18 DBPutPHZonelist()

- **Summary:** Write an arbitrary, polyhedral zonelist object into a Silo file.
- **C Signature:**

```
int DBPutPHZonelist (DBfile *dbfile, char const *name, int nfaces,
    int const *nodecnts, int lnodelist, int const *nodelist,
    char const *extface, int nzones, int const *facecnts,
    int lfacelist, int const *facelist, int origin,
    int lo_offset, int hi_offset, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the zonelist structure.
nfaces	Number of faces in the zonelist. Note that faces shared between zones should only be counted once.
nodecnts	Array of length nfaces indicating the number of nodes in each face. That is nodecnts[i] is the number of nodes in face i.
lnodelist	Length of the succeeding nodelist array.
nodelist	Array of length lnodelist listing the nodes of each face. The list of nodes for face i begins at index Sum(nodecnts[j]) for j=0…i-1.
extface	An optional array of length nfaces where extface[i]!=0x0 means that face i is an external face. This argument may be NULL.
nzones	Number of zones in the zonelist.
facecnts	Array of length nzones where facecnts[i] is number of faces for zone i.
lfacelist	Length of the succeeding facelist array.
facelist	Array of face ids for each zone. The list of faces for zone i begins at index Sum(facecnts[j]) for j=0…i-1. Note, however, that each face is identified by a signed value where the sign is used to indicate which ordering of the nodes of a face is to be used. A face id >= 0 means that the node ordering as it appears in the nodelist should be used. Otherwise, the value is negative and it should be 1-complimented to get the face's true id. In addition, the node ordering for such a face is the opposite of how it appears in the nodelist. Finally, node orders over a face should be specified such that a right-hand rule yields the outward normal for the face relative to the zone it is being defined for.
origin	Origin for indices in the nodelist array. Should be zero or one.
lo-offset	Index of first real (e.g. non-ghost) zone in the list. All zones with index less than (<) lo-offset are treated as ghost-zones.
hi-offset	Index of last real (e.g. non-ghost) zone in the list. All zones with index greater than (>) hi-offset are treated as ghost zones.

- **Returned value:**

  Returns zero on success or -1 on failure.

- **Description:**

  The DBPutPHZonelist function writes a polyhedral-zonelist object into a Silo file. The name assigned to this object can in turn be used as the parameter in the DBOPT_PHZONELIST option for the DBPutUcdmesh function.

  The following table describes the options accepted by this function:

Option Name	Data Type	Option Meaning	Default Value
DBOPT_ZONENUM	void*	Array of global zone numbers, one per zone in this zonelist. By default, it is assumed this array is of type int*	NULL
DBOPT_LLONGNZNUM	int	Indicates that the array passed for DBOPT_ZONENUM option is of long long type instead of int.	0
DBOPT_GHOST_ZONE_LABELS	char*	Optional array of char values indicating the ghost labeling DB_GHOSTTYPE_NOGHOST or DB_GHOSTTYPE_INTDUP) of each zone	NULL
DBOPT_ALT_ZONENUM_VARS	char**	A null terminated list of names of optional array(s) or DBucdvar objects indicating (multiple) alternative numbering(s) for zones	NULL

In interpreting the diagram above, numbers correspond to nodes while letters correspond to faces. In addition, the letters are drawn such that they will always be in the lower, right hand corner of a face if you were standing outside the object looking towards the given face. In the example code below, the list of nodes for a given face begin with the node nearest its corresponding letter.

For toplogically 2D meshes, two different approaches are possible for creating a polyhedral zonelist. One is to simple have a single list of "faces" representing the polygons of the 2D mesh. The other is to create an explicit list of "edges" and then define each polygon in terms of the edges it comprises. Either is appropriate.

```
#define NNODES 12
#define NFACES 11
#define NZONES 2

/* coordinate arrays */
float x[NNODES] = {0.0, 1.0, 2.0, 0.0, 1.0, 2.0, 0.0, 1.0, 2.0, 0.0, 1.0, 2.0};
float y[NNODES] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0};
float z[NNODES] = {0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0};

/* facelist where we enumerate the nodes over each face */
int nodecnts[NFACES] = {4,4,4,4,4,4,4,4,4,4,4};
int lnodelist = 4*NFACES;

/*                            a           b           c       */
int nodelist[4*NFACES] = {1,7,6,0,    2,8,7,1     4,1,0,3,

/*                            d           e           f       */
5,2,1,4,    3,9,10,4,   4,10,11,5,

/*                            g           h           i       */
9,6,7,10,   10,7,8,11,  0,6,9,3,

/*                             j          K                   */
1,7,10,4,   5,11,8,2};

/* zonelist where we enumerate the faces over each zone */
int facecnts[NZONES] = {6,6};
int lfacelist = 6*NZONES;
int facelist[6*NZONES] = {0,2,4,6,8,-9,   1,3,5,7,9,10};
```

1./images/ucd_hex_outward_normals.gif

Figure 0-4: Example of a polyhedral zonelist representation for two hexahedral elements.

1.4.19 DBGetPHZonelist()

- **Summary:** Read a polyhedral-zonelist from a Silo database.
- **C Signature:**

```
DBphzonelist *DBGetPHZonelist (DBfile *dbfile,
    char const *phzlname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
phzlname	Name of the polyhedral-zonelist.

- **Returned value:**

  Returns a pointer to a *DBphzonelist* structure on success and NULL on failure.

- **Description:**

  The DBGetPHZonelist function allocates a *DBphzonelist* data structure, reads a polyhedral-zonelist from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.20 DBPutFacelist()

- **Summary:** Write a facelist object into a Silo file.
- **C Signature:**

```
int DBPutFacelist (DBfile *dbfile, char const *name, int nfaces,
    int ndims, int const nodelist[], int lnodelist, int origin,
    int const zoneno[], int const shapesize[],
    int const shapecnt[], int nshapes, int const types[],
    int const typelist[], int ntypes)
```

- **Fortran Signature:**

```
integer function dbputfl(dbid, name, lname, ndims nodelist,
   lnodelist, origin, zoneno, shapesize, shapecnt, nshaps,
   types, typelist, ntypes, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the facelist structure.
nfaces	Number of external faces in associated mesh.
ndims	Number of spatial dimensions represented by the associated mesh.
nodelist	Array of length lnodelist containing node indices describing mesh faces.
lnodelist	Length of nodelist array.
origin	Origin for indices in nodelist array. Either zero or one.
zoneno	Array of length nfaces containing the zone number from which each face came. Use a NULL for this parameter if zone numbering info is not wanted.
shapesize	Array of length nshapes containing the number of nodes used by each face shape (for 3-D meshes only).
shapecnt	Array of length nshapes containing the number of faces having each shape (for 3-D meshes only).
nshapes	Number of face shapes (for 3-D meshes only).
types	Array of length nfaces containing information about each face. This argument is ignored if ntypes is zero, or if this parameter is NULL.
typelist	Array of length ntypes containing the identifiers for each type. This argument is ignored if ntypes is zero, or if this parameter is NULL.
ntypes	Number of types, or zero if type information was not provided.

- **Returned value:**

  Returns zero on success or -1 on failure.

- **Description:**

  The `DBPutFacelist` function writes a facelist object into a Silo file. The `name` given to this object can in turn be used as a parameter to the `DBPutUcdmesh` function.

  See *DBPutUcdmesh* for a full description of the facelist data structures.

1.4.21 DBPutUcdvar()

- **Summary:** Write a scalar/vector/tensor UCD variable object into a Silo file.
- **C Signature:**

```
int DBPutUcdvar (DBfile *dbfile, char const *name,
    char const *meshname, int nvars,
    char const * const varnames[], void const * const vars[],
    int nels, void const * const mixvars[], int mixlen,
    int datatype, int centering, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the variable.
`meshname`	Name of the mesh associated with this variable (written with DBPutUcdmesh).
`nvars`	Number of sub-variables which comprise this variable. For a scalar array, this is one. If writing a vector quantity, however, this would be two for a 2-D vector and three for a 3-D vector.
`varnames`	Array of length `nvars` containing pointers to character strings defining the names associated with each subvariable.
`vars`	Array of length `nvars` containing pointers to arrays defining the values associated with each subvariable.
`nels`	Number of elements in this variable.
`mixvars`	Array of length `nvars` containing pointers to arrays defining the mixed-data values associated with each subvariable. If no mixed values are present, this should be `NULL`.
`mixlen`	Length of mixed data arrays (i.e., mixvars).
`datatype`	Datatype of sub-variables. One of the predefined Silo data types.
`centering`	Centering of the sub-variables on the associated mesh. One of the predefined types:`DB_NODECENT`, `DB_EDGECENT`, `DB_FACECENT`, `DB_ZONECENT` or `DB_BLOCKCENT`. See below for a discussion of `centering` issues.
`optlist`	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use `NULL` for this argument.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutUcdvar function writes a variable associated with an UCD mesh into a Silo file. Note that variables can be node-centered, zone-centered, edge-centered or face-centered.

  For node- (or zone-) centered data, the question of which value in the vars array goes with which node (or zone) is determined implicitly by a one-to-one correspondence with the list of nodes in the DBPutUcdmesh call (or zones in the DBPutZonelist or DBPutZonelist2 call). For example, the 237th value in a zone-centered vars array passed here goes with the 237th zone in the zonelist passed in the DBPutZonelist2 (or DBPutZonelist) call.

  Edge- and face-centered data require a little more explanation. Unlike node- and zone-centered data, there does not exist in Silo an explicit list of edges or faces. As an aside, the DBPutFacelist call is really for writing the external faces of a mesh so that a downstream visualization tool need not have to compute them when it displays the mesh. Now, requiring the caller to create explicit lists of edges and/or faces in order to handle edge- or face-centered data results in unnecessary additional data being written to a Silo file. This increases file size as well as the time to write and read the file. To avoid this, we rely upon implicit lists of edges and faces.

  We define implicit lists of edges and faces in terms of a traversal of the zonelist structure of the associated mesh. The position of an edge (or face) in its list is determined by the order of its first occurrence in this traversal. The traversal algorithm is to visit each zone in the zonelist and, for each zone, visit its edges (or faces) in local order. See *figure showing UCD zoo and node, edge and face orderings* Because this traversal will wind up visiting edges multiple times, the first time an edge (or face) is encountered is what determines its position in the implicit edge (or face) list.

  If the zonelist contains arbitrary polyhedra or the zonelist is a polyhedral zonelist (written with DBPutPH-Zonelist), then the traversal algorithm involves visiting each zone, then each face for a zone and finally each edge for a face.

  Note that DBPutUcdvar() can also be used to define a block-centered variable on a multi-block mesh by specifying a multi-block mesh name for the meshname and DB_BLOCKCENT for the centering. This is useful in defining, for example, multi-block variable extents.

  Other information can also be included. This function is useful for writing vector and tensor fields, whereas the companion function, DBPutUcdvar1, is appropriate for writing scalar fields.

  For tensor quantities, the question of ordering of tensor components arises. For symmetric tensor's Silo uses the Voigt Notation ordering. In 2D, this is T11, T22, T12. In 3D, this is T11, T22, T33, T23, T13, T12. For full tensor quantities, ordering is row by row starting with the top row.

  The following table describes the options accepted by this function:

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_COORDSYS	int	Coordinate system. One of:DB_CARTESIAN, DB_CYLINDRICAL, DB_SPHERICAL, DB_NUMERICAL, or DB_OTHER	DB_OTHER
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_LABEL	char*	Character strings defining the label associated with this variable	NULL
DBOPT_ORIGIN	int	Origin for arrays. Zero or one.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_UNITS	char*	Character string defining the units associated with this variable	NULL
DBOPT_USESPECMF	int	Boolean DB_OFF orDB_ON) value specifying whether or not to weight the variable by the species mass fraction when using material species data	DB_OFF
DBOPT_ASCII_LABEL	int	Indicate if the variable should be treated as single character, ascii values. A value of 1 indicates yes, 0 no.	0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_REGION_PNAMES	char**	Pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be null and is used to indicate the end of the list of names. See *DBOPT_REGION_PNAMES*	NULL
DBOPT_CONSERVED	int	Indicates if the variable represents a physical quantity that must be conserved under various operations such as interpolation.	0
DBOPT_EXTENSIVE	int	Indicates if the variable represents a physical quantity that is extensive (as opposed to intensive). Note, while it is true that any conserved quantity is extensive, the converse is not true. By default and historically, all Silo variables are treated as intensive.	0
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent “missing values” in the variable data arrays. Default isDB_MISSING_VALUE_NOT_SET	DB_MISSING_VALUE_NOT

1.4.22 DBPutUcdvar1()

- **Summary:** Write a scalar UCD variable object into a Silo file.
- **C Signature:**

```
int DBPutUcdvar1 (DBfile *dbfile, char const *name,
    char const *meshname, void const *var, int nels,
    void const *mixvar, int mixlen, int datatype, int centering,
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputuv1(dbid, name, lname, meshname,
   lmeshname, var, nels, mixvar, mixlen, datatype,
   centering, optlist_id, staus)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable.
meshname	Name of the mesh associated with this variable (written with either DBPutUcdmesh).
var	Array of length nels containing the values associated with this variable.
nels	Number of elements in this variable.
mixvar	Array of length mixlen containing the mixed-data values associated with this variable. If mixlen is zero, this value is ignored.
mixlen	Length of mixvar array. If zero, no mixed data is present.
datatype	Datatype of variable. One of the predefined Silo data types.
centering	Centering of the sub-variables on the associated mesh. One of the predefined types:DB_NODECENT, DB_EDGECENT, DB_FACECENT or DB_ZONECENT.
optlist	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use NULL for this argument.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  DBPutUcdvar1 writes a variable associated with an UCD mesh into a Silo file. Note that variables will be either node-centered or zone-centered. Other information can also be included. This function is useful for writing scalar fields, whereas the companion function, DBPutUcdvar, is appropriate for writing vector and tensor fields.

  See *DBPutUcdvar* for a description of options accepted by this function.

1.4.23 DBGetUcdvar()

- **Summary:** Read a UCD variable from a Silo database.
- **C Signature:**

```
DBucdvar *DBGetUcdvar (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the variable.

- **Returned value:**

  Returns a pointer to a *DBucdvar* structure on success and NULL on failure.

- **Description:**

  The DBGetUcdvar function allocates a *DBucdvar* data structure, reads a variable associated with a UCD mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.24 DBPutCsgmesh()

- **Summary:** Write a CSG mesh object to a Silo file
- **C Signature:**

```
DBPutCsgmesh(DBfile *dbfile, const char *name, int ndims,
    int nbounds,
    const int *typeflags, const int *bndids,
    const void *coeffs, int lcoeffs, int datatype,
    const double *extents, const char *zonel_name,
    DBoptlist const *optlist);
```

- **Fortran Signature:**

```
integer function dbputcsgm(dbid, name, lname, ndims,
   nbounds, typeflags, bndids, coeffs, lcoeffs, datatype,
   extents, zonel_name, lzonel_name, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
name	Name to associate with this DBcsgmesh object
ndims	Number of spatial and topological dimensions of the CSG mesh object
nbounds	Number of boundaries in the CSG mesh description.
typeflags	Integer array of length nbounds of type information for each boundary. This is used to encode various information about the type of each boundary such as, for example, plane, sphere, cone, general quadric, etc as well as the number of coefficients in the representation of the boundary. For more information, see the description, below.
bndids	Optional integer array of length nbounds which are the explicit integer identifiers for each boundary. It is these identifiers that are used in expressions defining a region of the CSG mesh. If the caller passes NULL for this argument, a natural numbering of boundaries is assumed. That is, the boundary occurring at position i, starting from zero, in the list of boundaries here is identified by the integer i.
coeffs	Array of length lcoeffs of coefficients used in the representation of each boundary or, if the boundary is a transformed copy of another boundary, the coefficients of the transformation. In the case where a given boundary is a transformation of another boundary, the first entry in the coeffs entries for the boundary is the (integer) identifier for the referenced boundary. Consequently, if the datatype for coeffs isDB_FLOAT, there is an upper limit of about 16.7 million (2^24) boundaries that can be referenced in this way.
lcoeffs	Length of the coeffs array.
datatype	The data type of the data in the coeffs array.
zonel_name	Name of CSG zonelist to be associated with this CSG mesh object
extents	Array of length 2*ndims of spatial extents, xy(z)-minimums followed by xy(z)-maximums.
optlist	Pointer to an option list structure containing additional information to be included in the CSG mesh object written into the Silo file. Use NULL if there are no options.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The word *mesh* in this function name is probably somewhat misleading because it suggests a discretization of a domain into a *mesh*. In fact, a CSG (Constructive Solid Geometry) *mesh* in Silo is a continuous, analytic

representation of the geometry of some computational domain. Nonetheless, most of Silo's concepts for meshes, variables, materials, species and multi-block objects apply equally well in the case of a CSG *mesh* and so that is what it is called, here. Presently, Silo does not have functions to discretize this kind of mesh. It has only the functions for storing and retrieving it. Nonetheless, a future version of Silo may include functions to discretize a CSG mesh.

A CSG mesh is constructed by starting with a list of analytic boundaries, that is curves in 2D or surfaces in 3D, such as planes, spheres and cones or general quadrics. Each boundary is defined by an analytic expression (an equation) of the form f(x,y,z)=0 (or, in 2D, f(x,y)=0) in which the highest exponent for x, y or z is 2. That is, all the boundaries are quadratic (or *quadric*) at most.

The table below describes how to use the `typeflags` argument to define various kinds of boundaries in 3 dimensions.

Optlist options:

typeflag	num-coeffs	coefficients and equation
DBCSG_QUADRIC_G	10	
DBCSG_SPHERE_PR	4	
DBCSG_ELLIPSOID_PRRR	6	
DBCSG_PLANE_G	4	
DBCSG_PLANE_X	1	
DBCSG_PLANE_Y	1	
DBCSG_PLANE_Z	1	
DBCSG_PLANE_PN	6	
DBCSG_PLANE_PPP	9	
DBCSG_CYLINDER_PNLR	8	to be completed
DBCSG_CYLINDER_PPR	7	to be completed
DBCSG_BOX_XYZXYZ	6	to be completed
DBCSG_CONE_PNLA	8	to be completed
DBCSG_CONE_PPA		to be completed
DBCSG_POLYHEDRON_KF	K	6K
DBCSG_HEX_6F	36	to be completed
DBCSG_TET_4F	24	to be completed
DBCSG_PYRAMID_5F	30	to be completed
DBCSG_PRISM_5F	30	to be completed

The table below defines an analogous set of `typeflags` for creating boundaries in two dimensions..

typeflag	num-coeffs	coefficients and equation
DBCSG_QUADRATIC_G	6	
DBCSG_CIRCLE_PR	3	
DBCSG_ELLIPSE_PRR	4	
DBCSG_LINE_G	3	
DBCSG_LINE_X	1	
DBCSG_LINE_Y	1	
DBCSG_LINE_PN	4	
DBCSG_LINE_PP	4	
DBCSG_BOX_XYXY	4	to be completed
DBCSG_POLYGON_KP	K	2K
DBCSG_TRI_3P	6	to be completed
DBCSG_QUAD_4P	8	to be completed

By replacing the '=' in the equation for a boundary with either a '<' or a '>', whole regions in 2 or 3D space can be defined using these boundaries. These regions represent the set of all points that satisfy the inequality. In addition, regions can be combined to form new regions by unions, intersections and differences as well other operations (See *DBPutCSGZonelist*).

In this call, only the analytic boundaries used in the expressions to define the regions are written. The expressions defining the regions themselves are written in a separate call, `DBPutCSGZonelist`.

If you compare this call to write a CSG mesh to a Silo file with a similar call to write a UCD mesh, you will notice that the boundary list here plays a role similar to that of the nodal coordinates of a UCD mesh. For the UCD mesh, the basic geometric primitives are points (nodes) and a separate call, DBPutZonelist, is used to write out the information that defines how points (nodes) are combined to form the zones of the mesh.

Similarly, here the basic geometric primitives are analytic boundaries and a separate call, DBPutCSGZonelist, is used to write out the information that defines how the boundaries are combined to form regions of the mesh.

The following table describes the options accepted by this function. See the section titled "Using the Silo Option Parameter" for details on the use of the `DBoptlist` construct.

Option Name	Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_XLABEL	char*	Character string defining the label associated with the X dimension	NULL
DBOPT_YLABEL	char*	Character string defining the label associated with the Y dimension	NULL
DBOPT_ZLABEL	char*	Character string defining the label associated with the Z dimension	NULL
DBOPT_XUNITS	char*	Character string defining the units associated with the X dimension	NULL
DBOPT_YUNITS	char*	Character string defining the units associated with the Y dimension	NULL
DBOPT_ZUNITS	char*	Character string defining the units associated with the Z dimension	NULL
DBOPT_BNDNAMES	char**	Array of nboundaries character strings defining the names of the individual boundaries	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_MRGTREE_NAME	char*	Name of the mesh region grouping tree to be associated with this mesh	NULL
DBOPT_TV_CONNECTIVITY	int	A non-zero value indicates that the connectivity of the mesh varies with time	0
DBOPT_DISJOINT_MODE	int	Indicates if any elements in the mesh are disjoint. There are two possible modes. One is DB_ABUTTING indicating that elements abut spatially but actually reference different node ids (but spatially equivalent nodal positions) in the node list. The other is DB_FLOATING where elements neither share nodes in the nodelist nor abut spatially	DB_NONE
DBOPT_ALT_NODENUMVARS	char**	null terminated list of names of optional array(s) or DBcsgvar objects indicating (multiple) alternative numbering(s) for boundaries	NULL
The following options have been deprecated. Use MRG trees instead			
DBOPT_GROUPNUM	int	The group number to which this quadmesh belongs.	-1 (not in a group)

1.4.25 DBGetCsgmesh()

- **Summary:** Get a CSG mesh object from a Silo file
- **C Signature:**

```
DBcsgmesh *DBGetCsgmesh(DBfile *dbfile, const char *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
meshname	Name of the CSG mesh object to read

- **Returned value:**

  A pointer to a *DBcsgmesh* structure on success and NULL on failure.

1.4.26 DBPutCSGZonelist()

- **Summary:** Put a CSG zonelist object in a Silo file.
- **C Signature:**

```
int DBPutCSGZonelist(DBfile *dbfile, const char *name, int nregs,
    const int *typeflags,
    const int *leftids, const int *rightids,
    const void *xforms, int lxforms, int datatype,
    int nzones, const int *zonelist,
    DBoptlist *optlist);
```

- **Fortran Signature:**

```
integer function dbputcsgzl(dbid, name, lname, nregs,
   typeflags, leftids, rightids, xforms, lxforms, datatype,
   nzones, zonelist, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
name	Name to associate with the DBcsgzonelist object
nregs	The number of regions in the regionlist.
typeflags	Integer array of length nregs of type information for each region. Each entry in this array is one of either DB_INNER, DB_OUTER, DB_ON, DB_XFORM, DB_SWEEP, DB_UNION, DB_INTERSECT, and DB_DIFF.The symbols,DB_INNER, DB_OUTER, DB_ON, DB_XFORM and DB_SWEEP represent unary operators applied to the referenced region (or boundary). The symbols DB_UNION, DB_INTERSECT, and DB_DIFF represent binary operators applied to two referenced regions.For the unary operators, DB_INNER forms a region from a boundary (See *DBPutCsgmesh*) by replacing the '=' in the equation representing the boundary with '<'. Likewise, DB_OUTER forms a region from a boundary by replacing the '=' in the equation representing the boundary with '>'. Finally, DB_ON forms a region (of topological dimension one less than the mesh) by leaving the '=' in the equation representing the boundary as an '='. In the case ofDB_INNER, DB_OUTER and DB_ON, the corresponding entry in the leftids array is a reference to a boundary in the boundary list (See *DBPutCsgmesh*).For the unary operator,DB_XFORM, the corresponding entry in the leftids array is a reference to a region to be transformed while the corresponding entry in the rightids array is the index into the xform array of the row-by-row coefficients of the affine transform. The unary operator DB_SWEEP is not yet implemented.
leftids	Integer array of length nregs of references to other regions in the regionlist or boundaries in the boundary list (See *DBPutCsgmesh*). Each referenced region in the leftids array forms the left operand of a binary expression (or single operand of a unary expression) involving the referenced region or boundary.
rightids	Integer array of length nregs of references to other regions in the regionlist. Each referenced region in the rightids array forms the right operand of a binary expression involving the region or, for regions which are copies of other regions with a transformation applied, the starting index into the xforms array of the row-by-row, affine transform coefficients. If for a given region no right reference is appropriate, put a value of '-1' into this array for the given region.
xforms	Array of length lxforms of row-by-row affine transform coefficients for those regions that are copies of other regions except with a transformation applied. In this case, the entry in the leftids array indicates the region being copied and transformed and the entry in the rightids array is the starting index into this xforms array for the transform coefficients. This argument may be NULL.
lxforms	Length of the xforms array. This argument may be zero if xforms is NULL.
datatype	The data type of the values in the xforms array. Ignored if xforms is NULL.
nzones	The number of zones in the CSG mesh. A zone is really just a completely defined region.
zonelist	Integer array of length nzones of the regions in the regionlist that form the actual zones of the CSG mesh.
optlist	Pointer to an option list structure containing additional information to be included in this object when it is written to the Silo file. Use NULL if there are no options.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  A CSG mesh is a list of curves in 2D or surfaces in 3D. These are analytic expressions of the boundaries of objects that can be expressed by quadratic equations in x, y and z.

  The zonelist for a CSG mesh is constructed by first defining regions from the mesh boundaries. For example, given the boundary for a sphere, we can create a region by taking the inside DB_INNER) of that boundary or by taking the outside DB_OUTER). In addition, regions can also be created by boolean operations (union, intersect, diff) on other regions. The table below summarizes how to construct regions using the typeflags argument.

  Optlist options:

op. symbol name	type	meaning
DBCSG_INNER	unary	specifies the region created by all points satisfying the equation defining the boundary with < replacing =, left operand indicates the boundary, right operand ignored
DBCSG_OUTER	unary	specifies the region created by all points satisfying the equation defining the boundary with > replacing =, left operand indicates the boundary, right operand ignored
DBCSG_ON	unary	specifies the region created by all points satisfying the equation defining the boundary left operand indicates the boundary, right operand ignored
DBCSG_UNION	binary	take the union of left and right operands left and right operands indicate the regions
DBCSG_INTERSECT	binary	take the intersection of left and right operands left and right operands indicate the regions
DBCSG_DIFF	binary	subtract the right operand from the left left and right operands indicate the regions
DBCSG_COMPLIMENT	unary	take the compliment of the left operand, left operand indicates the region, right operand ignored
DBCSG_XFORM	unary	to be implemented
DBCSG_SWEEP	unary	to be implemented

However, not all regions in a CSG `zonelist` form the actual zones of a CSG mesh. Some regions exist only to facilitate the construction of other regions. Only certain regions, those that are completely constructed, form the actual zones. Consequently, the `zonelist` for a CSG mesh involves both a list of regions (as well as the definition of those regions) and then a list of zones (which are really just completely defined regions).

The following table describes the options accepted by this function. See the section titled "Using the Silo Option Parameter" for details on the use of the `DBoptlist` construct.

Option Name	Data Type	Option Meaning	Default Value
DBOPT_REGNAMES	char**	Array of `nregs` character strings defining the names of the individual regions	NULL
DBOPT_ZONENAMES	char**	Array of `nzones` character strings defining the names of individual zones	NULL
DBOPT_ALT_ZONENUM_VARS	char**	A null terminated list of names of optional array(s) or `DBcsgvar` objects indicating (multiple) alternative numbering(s) for zones	NULL

Figure 0-5: A relatively simple object to represent as a CSG mesh. It models an A/C vent outlet for a 1994 Toyota Tercel. It consists of two zones. One is a partially-spherical shaped ring housing (darker area). The other is a lens-shaped fin used to direct airflow (lighter area).

The table below describes the contents of the boundary list (written in the `DBPutCsgmesh` call)

typeflags	id	coefficients	name (optional)
DBCSG_SPHERE_PR	0	0.0, 0.0, 0.0, 5.0	"housing outer shell"
DBCSG_PLANE_X	1	-2.5	"housing front"
DBCSG_PLANE_X	2	2.5	"housing back"
DBCSG_CYLINDER_PPR	3	0.0, 0.0, 0.0, 1.0, 0.0. 0.0, 3.0	"housing cavity"
DBCSG_SPHERE_PR	4	0.0, 0.0, 49.5, 50.0	"fin top side"
DBCSG_SPHERE_PR	5	0.0. 0.0, -49.5, 50.0	"fin bottom side"

The code below writes this CSG mesh to a silo file

```
int *typeflags={DBCSG_SPHERE_PR, DBCSG_PLANE_X, DBCSG_PLANE_X,
                DBCSG_CYLINDER_PPR, DBCSG_SPHERE_PR, DBCSG_SPHERE_PR};
float *coeffs = {0.0, 0.0, 0.0, 5.0, 1.0, 0.0, 0.0, -2.5,
                 1.0, 0.0, 0.0, 2.5, 1.0, 0.0, 0.0, 0.0, 3.0,
                 0.0, 0.0, 49.5, 50.0, 0.0.
                 0.0, -49.5, 50.0};

DBPutCsgmesh(dbfile, "csgmesh", 3, typeflags, NULL,
    coeffs, 25, DB_FLOAT, "csgzl", NULL);
```

The table below describes the contents of the regionlist, written in the `DBPutCSGZonelist` call.

typeflags	regid	left-ids	right-ids	notes
DBCSG_INNER	0	0	-1	creates inner sphere region from boundary 0
DBCSG_INNER	1	1	-1	creates front half-space region from boundary 1
DBCSG_OUTER	2	2	-1	creates back half-space region from boundary 2
DBCSG_INNER	3	3	-1	creates inner cavity region from boundary 3
DBCSG_INTERSECT	4	0	1	cuts front of sphere by intersecting regions 0 &1
DBCSG_INTERSECT	5	4	2	cuts back of sphere by intersecting regions 4 & 2
DBCSG_DIFF	6	5	3	creates cavity in sphere by removing region 3
DBCSG_INNER	7	4	-1	creates large sphere region for fin upper surface from boundary 4
DBCSG_INNER	8	5	-1	creates large sphere region for fin lower surface from boundary 5
DBCSG_INTERSECT	9	7	8	creates lens-shaped fin with razor edge protruding from sphere housing by intersecting regions 7 & 8
DBCSG_INTERSECT	10	9	0	cuts razor edge of lens-shaped fin to sphere housing

The table above creates 11 regions, only 2 of which form the actual zones of the CSG mesh. The 2 complete zones are for the spherical ring housing and the lens-shaped fin that sits inside it. They are identified by region ids 6 and 10. The other regions exist solely to facilitate the construction. The code to write this CSG `zonelist` to a silo file is given below.

```
int nregs = 11;
int *typeflags={DBCSG_INNER, DBCSG_INNER, DBCSG_OUTER, DBCSG_INNER,
                DBCSG_INTERSECT, DBCSG_INTERSECT, DBCSG_DIFF,
                DBCSG_INNER, DBCSG_INNER, DBCSG_INTERSECT, DBCSG_INTERSECT};
int *leftids={0,1,2,3,0,4,5,4,5,7,9};
int *rightids={-1,-1,-1,-1,1,2,3,-1,-1,8,0};
int nzones = 2;
int *zonelist = {6, 10};

DBPutCSGZonelist(dbfile, "csgzl", nregs, typeflags,
    leftids, rightids, NULL, 0, DB_INT,
    nzones, zonelist, NULL);
```

1.4.27 DBGetCSGZonelist()

- **Summary:** Read a CSG mesh zonelist from a Silo file
- **C Signature:**

```
DBcsgzonelist *DBGetCSGZonelist(DBfile *dbfile,
    const char *zlname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
zlname	Name of the CSG mesh zonelist object to read

- **Returned value:**

  A pointer to a *DBcsgzonelist* structure on success and NULL on failure.

1.4.28 DBPutCsgvar()

- **Summary:** Write a CSG mesh variable to a Silo file
- **C Signature:**

```
int DBPutCsgvar(DBfile *dbfile, const char *vname,
    const char *meshname, int nvars,
    const char * const varnames[],
    const void * const vars[], int nvals, int datatype,
    int centering, DBoptlist const *optlist);
```

- **Fortran Signature:**

```
integer function dbputcsgv(dbid, vname, lvname, meshname,
   lmeshname, nvars, var_ids, nvals, datatype, centering,
   optlist_id, status)

integer* var_ids (array of "pointer ids" created using dbmkptr)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
vname	The name to be associated with this DBcsgvar object
meshname	The name of the CSG mesh this variable is associated with
nvars	The number of subvariables comprising this CSG variable
varnames	Array of length nvars containing the names of the subvariables
vars	Array of pointers to variable data
nvals	Number of values in each of the vars arrays
datatype	The type of data in the vars arrays (e.g.DB_FLOAT, DB_DOUBLE)
centering	The centering of the CSG variable DB_ZONECENT orDB_BNDCENT)
optlist	Pointer to an option list structure containing additional information to be included in this object when it is written to the Silo file. Use NULL if there are no options

- **Description:**

  The DBPutCsgvar function writes a variable associated with a CSG mesh into a Silo file. Note that variables will be either zone-centered or boundary-centered.

  Just as UCD variables can be zone-centered or node-centered, CSG variables can be zone-centered or boundary-centered. For a zone-centered variable, the value(s) at index i in the vars array(s) are associated with the ith region (zone) in the DBcsgzonelist object associated with the mesh. For a boundary-centered variable, the value(s) at index i in the vars array(s) are associated with the ith boundary in the DBcsgbnd list associated with the mesh.

  Other information can also be included via the optlist:

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_LABEL	char*	Character strings defining the label associated with this variable	NULL
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_UNITS	char*	Character string defining the units associated with this variable	NULL
DBOPT_USESPECMF	int	Boolean DB_OFF orDB_ON) value specifying whether or not to weight the variable by the species mass fraction when using material species data	DB_OFF
DBOPT_ASCII_LABEL	int	Indicate if the variable should be treated as single character, ascii values. A value of 1 indicates yes, 0 no.	0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_REGION_PNAMES	char**	A pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be null and is used to indicate the end of the list of names. See *DBOPT_REGION_PNAMES*	NULL
DBOPT_CONSERVED	int	Indicates if the variable represents a physical quantity that must be conserved under various operations such as interpolation.	0
DBOPT_EXTENSIVE	int	Indicates if the variable represents a physical quantity that is extensive (as opposed to intensive). Note, while it is true that any conserved quantity is extensive, the converse is not true. By default and historically, all Silo variables are treated as intensive.	0
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent “missing values” in the x or y data arrays. Default isDB_MISSING_VALUE_NOT_SET	DB_MISSING_VALUE_NOT

1.4.29 DBGetCsgvar()

- **Summary:** Read a CSG mesh variable from a Silo file
- **C Signature:**

```
DBcsgvar *DBGetCsgvar(DBfile *dbfile, const char *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
varname	Name of CSG variable object to read

- **Returned value:**

  A pointer to a *DBcsgvar* structure on success and NULL on failure.

1.4.30 DBPutMaterial()

- **Summary:** Write a material data object into a Silo file.
- **C Signature:**

```
int DBPutMaterial (DBfile *dbfile, char const *name,
    char const *meshname, int nmat, int const matnos[],
    int const matlist[], int const dims[], int ndims,
    int const mix_next[], int const mix_mat[],
    int const mix_zone[], void const *mix_vf, int mixlen,
    int datatype, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputmat(dbid, name, lname, meshname,
   lmeshname, nmat, matnos, matlist, dims, ndims,
   mix_next, mix_mat, mix_zone, mix_vf, mixlien, datatype,
   optlist_id, status)

void* mix_vf
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the material data object.
meshname	Name of the mesh associated with this information.
nmat	Number of materials.
matnos	Array of length nmat containing material numbers.
matlist	Array whose dimensions are defined by dims and ndims. It contains the material numbers for each single-material (non-mixed) zone, and indices into the mixed data arrays for each multi-material (mixed) zone. A negative value indicates a mixed zone, and its absolute value is used as an index into the mixed data arrays.
dims	Array of length ndims which defines the dimensionality of the matlist array.
ndims	Number of dimensions in matlist array.
mix_next	Array of length mixlen of indices into the mixed data arrays (one-origin).
mix_mat	Array of length mixlen of material numbers for the mixed zones.
mix_zone	Optional array of length mixlen of back pointers to originating zones. The origin is determined by DBOPT_ORIGIN. Even if mixlen > 0, this argument is optional.
mix_vf	Array of length mixlen of volume fractions for the mixed zones. Note, this can actually be either single- or double-precision. Specify actual type in datatype.
mixlen	Length of mixed data arrays (or zero if no mixed data is present). If mixlen > 0, then the "mix_" arguments describing the mixed data arrays must be nonNULL.
datatype	Volume fraction data type. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the material object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use NULL for this argument.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

Note that material functionality, even mixing materials, can now be handled, often more conveniently and efficiently, via a Mesh Region Grouping (MRG) tree. Users are encouraged to consider an MRG tree as an alternative to DBPutMaterial(). See *DBMakeMrgtree*.

The DBPutMaterial function writes a material data object into the current open Silo file. The minimum required information for a material data object is supplied via the standard arguments to this function. The optlist argument must be used for supplying any information not requested through the standard arguments.

The following table describes the options accepted by this function.

Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_LABEL	char*	Character string defining the label associated with material data	NULL
DBOPT_MAJORORDER	int	Indicator for row-major (0) or column-major (1) storage for multidimensional arrays.	0
DBOPT_ORIGIN	int	Origin for mix_zone. Zero or one.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_MATNAMES	char**	Array of strings defining the names of the individual materials	NULL
DBOPT_MATCOLORS	char**	Array of strings defining the names of colors to be associated with each material. The color names are taken from the X windows color database. If a color name begins with a'#' symbol, the remaining 6 characters are interpreted as the hexadecimal RGB value for the color	NULL
DBOPT_HIDEFROMGUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_ALLOWMAT0	int	If set to non-zero, indicates that a zero entry in the matlist array is actually not a valid material number but is instead being used to indicate an 'unused' zone.	0

The model used for storing material data is the most efficient for VisIt, and works as follows:

One zonal array, matlist, contains the material number for a clean zone or an index into the mixed data arrays if the zone is mixed. Mixed zones are marked with negative entries in matlist, so you must take abs(matlist[i]) to get the actual 1-origin mixed data index. All indices are 1-origin to allow matlist to use zero as a material number.

The mixed data arrays are essentially a linked list of information about the mixed elements within a zone. Each mixed data array is of length mixlen. For a given index i, the following information is known about the i'th element:

mix_zone[i]
The index of the zone which contains this element. The origin is determined by DBOPT_ORIGIN.

mix_mat[i]
The material number of this element

mix_vf[i]
The volume fraction of this element

mix_next[i]
The 1-origin index of the next material entry for this zone, else 0 if this is the last entry.

Figure 0-6: Example using mixed data arrays for representing material information

1.4.31 DBGetMaterial()

- **Summary:** Read material data from a Silo database.
- **C Signature:**

```
DBmaterial *DBGetMaterial (DBfile *dbfile, char const *mat_name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
mat_name	Name of the material variable to read.

- **Returned value:**

  Returns a pointer to a *DBmaterial* structure on success and NULL on failure.

- **Description:**

  The DBGetMaterial function allocates a DBmaterial data structure, reads material data from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.32 DBPutMatspecies()

- **Summary:** Write a material species data object into a Silo file.
- **C Signature:**

```
int DBPutMatspecies (DBfile *dbfile, char const *name,
    char const *matname, int nmat, int const nmatspec[],
    int const speclist[], int const dims[], int ndims,
    int nspecies_mf, void const *species_mf, int const mix_spec[],
    int mixlen, int datatype, DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputmsp(dbid, name, lname, matname,
   lmatname, nmat, nmatspec, speclist, dims, ndims,
   species_mf, species_mf, mix_spec, mixlen, datatype, optlist_id,
   status)

void *species_mf
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the material species data object.
matname	Name of the material object with which the material species object is associated.
nmat	Number of materials in the material object referenced by matname.
nmatspec	Array of length nmat containing the number of species associated with each material.
speclist	Array of dimension defined by ndims and dims of indices into the species_mf array. Each entry corresponds to one zone. If the zone is clean, the entry in this array must be positive or zero. A positive value is a 1-origin index into the species_mf array. A zero can be used if the material in this zone contains only one species. If the zone is mixed, this value is negative and is used to index into the mix_spec array in a manner analogous to the mix_mat array of the DBPutMaterial() call.
dims	Array of length ndims that defines the shape of the speclist array. To create an empty matspecies object, set every entry of dims to zero. See description below.
ndims	Number of dimensions in the speclist array.
nspecies_mf	Length of the species_mf array. To create a homogeneous matspecies object (which is not quite empty), set nspecies_mf to zero. See description below.
species_mf	Array of length nspecies_mf containing mass fractions of the material species. Note, this can actually be either single or double precision. Specify type in datatype argument.
mix_spec	Array of length mixlen containing indices into the species_mf array. These are used for mixed zones. For every index j in this array, mix_list[j] corresponds to the DBmaterial structure's material mix_mat[j] and zone mix_zone[j].
mixlen	Length of the mix_spec array.
datatype	The datatype of the mass fraction data in species_mf. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

- **Returned value:**

  DBPutMatspecies returns zero on success and -1 on failure.

- **Description:**

  The DBPutMatspecies function writes a material species data object into a Silo file. The minimum required information for a material species data object is supplied via the standard arguments to this function. The optlist argument must be used for supplying any information not requested through the standard arguments.

  It is easiest to understand material species information by example. First, in order for a material species object in Silo to have meaning, it must be associated with a material object. A material species object by itself with no corresponding material object cannot be correctly interpreted.

  So, suppose you had a problem which contains two materials, brass and steel. Now, neither brass nor steel are themselves pure elements on the periodic table. They are instead alloys of other (pure) metals. For example, common yellow brass is, nominally, a mixture of Copper (Cu) and Zinc (Zn) while tool steel is composed primarily of Iron (Fe) but mixed with some Carbon (C) and a variety of other elements.

  For this example, lets suppose we are dealing with Brass (65% Cu, 35% Zn), T-1 Steel (76.3% Fe, 0.7% C, 18% W, 4% Cr,1% V) and O-1 Steel (96.2% Fe, 0.90% C,1.4% Mn, 0.50% Cr, 0.50% Ni, 0.50% W). Since T-1 Steel and O-1 Steel are composed of different elements, we wind up having to represent each type of steel as a different material in the material object. So, the material object would have 3 materials; Brass, T-1 Steel and O-1 Steel.

  Brass is composed of 2 species, T-1 Steel, 5 species and O-1 Steel, 6. (Alternatively, one could opt to characterize both T-1 Steel and O-1 Steel has having 7 species, Fe, C, Mn, Cr, Ni, W, V where for T-1 Steel, the Mn and Ni components are always zero and for O-1 Steel the V component is always zero. In that case, you would need only 2 materials in the associated material object.)

The material species object would be defined such that nmat=3 and nmatspec={2,5,6}. If the composition of Brass, T-1 Steel and O-1 Steel is constant over the whole mesh, the `species_mf` array would contain just 2 + 5 + 6 = 13 entries…

Brass T1 Steel O1 Steel													
species_mf	.65	.35	.763	.007	.18	.04	.001	.962	.009	.014	.005	.005	.005
element	Cu	Zn	Fe	C	W	Cr	V	Fe	C	Mn	Cr	Ni	W
index	1	2	3	4	5	6	7	8	9	10	11	12	13

If all of the zones in the mesh are clean (e.g. not mixing in material) and have the same composition of species, the `speclist` array would contain a 1 for every Brass zone (1-origin indexing would mean it would index `species_mf[0]`), a 3 for every T-1 Steel zone and a 8 for every O-1 Steel zone. However, if some cells had a Brass mixture with an extra 1% Cu, then you could create another two entries at positions 14 and 15 in the `species_mf` array with the values 0.66 and 0.34, respectively, and the `speclist` array for those cells would point to 14 instead of 1.

The `speclist` entries indicate only where to start reading species mass fractions from the `species_mf` array for a given zone. How do we know how many values to read? The associated material object indicates which material is in the zone. The entry in the `nmatspec` array for that material indicates how many mass fractions there are.

As simulations evolve, the relative mass fractions of species comprising each material vary away from their nominal values. In this case, the `species_mf` array would grow to accommodate all the variations of combinations of mass fraction for each material and the entries in the `speclist` array would vary so that each zone would index the correct position in the `species_mf` array.

Finally, when zones contain mixing materials the `speclist` array needs to specify the `species_mf` entries for each of the materials in the zone. In this case, negative values are assigned to the `speclist` entries for these zones and the linked-list like structure of the associated material (e.g. mix_next, mix_mat, mix_vf, mix_zone args of the `DBPutMaterial()` call) is used to traverse them.

To create a completely empty matspecies object, the caller needs to pass a `dims` array containing all zeros. In this case, the `speclist` and mix_speclist args are never dereferenced and no data for these are written to the file or returned by a subsequent `DBGetMatspecies()` call. Alternatively, the caller may have what amounts to an empty species object due to nspecies_mf==0. However, in that situation, the caller is unfortunately still required to pass fully populated `speclist` and, if mixing is involved, mix_speclist arrays. Worse, the only valid values in these arrays are zeros. In this case, the resulting matspecies object isn't so much empty as it is homogeneous in that all materials consist of only a single species.

The following table describes the options accepted by this function:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_MAJORORDER	INTEGER	Indicator for row-major (0) or column-major (1) storage for multidimensional arrays.	0
DBOPT_ORIGIN	INTEGER	Origin for arrays. Zero or one.	0
DBOPT_HIDE_FROM_GUI	INTEGER	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_SPECNAMES	char**	Array of strings defining the names of the individual species. The length of this array is the sum of the values in the `nmatspec` argument to this function	NULL
DBOPT_SPECCOLORS	char**	Array of strings defining the names of colors to be associated with each species. The color names are taken from the X windows color database. If a color `name` begins with a'#' symbol, the remaining 6 characters are interpreted as the hexadecimal `RGB` value for the color. The length of this array is the sum of the values in the `nmatspec` argument to this function	NULL

1.4.33 DBGetMatspecies()

- **Summary:** Read material species data from a Silo database.
- **C Signature:**

```
DBmatspecies *DBGetMatspecies (DBfile *dbfile,
    char const *ms_name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
ms_name	Name of the material species data to read.

- **Returned value:**

  Returns a pointer to a *DBmatspecies* structure on success and NULL on failure.

- **Description:**

  The DBGetMatspecies function allocates a *DBmatspecies* data structure, reads material species data from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.34 DBPutDefvars()

- **Summary:** Write a derived variable definition(s) object into a Silo file.
- **C Signature:**

```
int DBPutDefvars(DBfile *dbfile, const char *name, int ndefs,
    const char * const names[], int const *types,
    const char * const defns[], DBoptlist cost *optlist[]);
```

- **Fortran Signature:**

```
integer function dbputdefvars(dbid, name, lname, ndefs,
   names, lnames, types, defns, ldefns, optlist_id,
   status)
```

character*N names (See *dbset2dstrlen*) character*N defns (See *dbset2dstrlen*)

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the derived variable definition(s) object.
ndefs	number of derived variable definitions.
names	Array of length ndefs of derived variable names
types	Array of length ndefs of *derived variable types*
defns	Array of length ndefs of derived variable definitions.
optlist	Array of length ndefs pointers to option list structures containing additional information to be included with each derived variable. The options available are the same as those available for the respective variables.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBPutDefvars function is used to put definitions of derived variables in the Silo file. That is variables that are derived from other variables in the Silo file or other derived variable definitions. One or more variable definitions can be written with this function. Note that only the definitions of the derived variables are written to the file with this call. The variables themselves are not in any way computed by Silo.

  If variable references within the defns strings do not have a leading slash ('/') (indicating an absolute name), they are interpreted relative to the directory into which the Defvars object is written. For the defns string, in cases where a variable's name includes special characters (such as / . { } [ ] + - =), the entire variable reference should be bracketed by < and > characters.

  The interpretation of the defns strings written here is determined by the post-processing tool that reads and interprets these definitions. Since in common practice that tool tends to be VisIt, the discussion that follows describes how VisIt would interpret this string.

  The table below illustrates examples of the contents of the various array arguments to DBPutDefvars for a case that defines 6 derived variables.

names	types	defns
"totaltemp"	DB_VARTYPE_SCALAR	"nodet+zonetemp"
"<stress/sz>"	DB_VARTYPE_SCALAR	"<stress/sx>-<stress/sy>"
"vel"	DB_VARTYPE_VECTOR	"{Vx, Vy, Vz}"
"speed"	DB_VARTYPE_SCALAR	"magntidue(vel)"
"dev_stress"	DB_VARTYPE_TENSOR	"{<stress/sx>,<stress/txy>,<stress/txz>}, { 0, <stress/sy>, <stress/tyz>}, { 0, 0, <stress/sz>} }"

  The first entry (0) defines a derived scalar variable named "totaltemp" which is the sum of variables whose names are "nodet" and "zonetemp". The next entry (1) defines a derived scalar variable named "sz" in a group of variables named "stress" (the slash character ('/') is used to group variable names much the way file pathnames are grouped in Linux). Note also that the definition of "sz" uses the special bracketing characters (<) and (>) for the variable references due to the fact that these variable references have a slash character (/) in them.

  The third entry (2) defines a derived vector variable named "vel" from three scalar variables named "Vx", "Vy", and "Vz" while the fourth entry (3) defines a scalar variable, "speed" to be the magnitude of the vector variable named "vel". The last entry (4) defines a deviatoric stress tensor. These last two cases demonstrate that derived variable definitions may reference other derived variables.

The last few examples demonstrate the use of two operators, `{}`, and `magnitude()`. We call these expression *operators*. In VisIt, there are numerous expression operators to help define derived variables including such things as `sqrt()`, `round()`, `abs()`, `cos()`, `sin()`, `dot()`, `cross()` as well as comparison operators, `gt()`, `ge()`, `lt()`, `le()`, `eq()`, and the conditional `if()`. Furthermore, the list of expression operators in VisIt grows regularly. Only a few examples are illustrated here. For a more complete list of the available expression operators and their syntax, the reader is referred to the Expressions portion of the VisIt user's manual.

1.4.35 DBGetDefvars()

- **Summary:** Get a derived variables definition object from a Silo file.
- **C Signature:**

```
DBdefvars DBGetDefvars(DBfile *dbfile, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	The name of the DBdefvars object to read

- **Returned value:**

  Returns a pointer to a *DBdefvars* structure on success and NULL on failure.

- **Description:**

  The DBGetDefvars function allocates a *DBdefvars* data structure, reads the object from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.4.36 DBInqMeshname()

- **Summary:** Inquire the mesh name associated with a variable.
- **C Signature:**

```
int DBInqMeshname (DBfile *dbfile, char const *varname,
    char *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Variable name.
meshname	Returned mesh name. The caller must allocate space for the returned name. The maximum space used is 256 characters, including the NULL terminator.

- **Returned value:**

  DBInqMeshname returns zero on success and -1 on failure.

- **Description:**

  The DBInqMeshname function returns the name of a mesh associated with a mesh variable. Given the name of a variable to access, one must call this function to find the name of the mesh before calling DBGetQuadmesh or DBGetUcdmesh.

1.4.37 DBInqMeshtype()

- **Summary:** Inquire the mesh type of a mesh.
- **C Signature:**

```
int DBInqMeshtype (DBfile *dbfile, char const *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
meshname	Mesh name.

- **Returned value:**

  DBInqMeshtype returns the mesh type on success and -1 on failure.

- **Description:**

  The DBInqMeshtype function returns the type of the given mesh. The value returned is described in the following table:

Mesh Type	Returned Value
Multi-Block	DB_MULTIMESH
UCD	DB_UCDMESH
Pointmesh	DB_POINTMESH
Quad (Collinear)	DB_QUAD_RECT
Quad (Non-Collinear)	DB_QUAD_CURV
CSG	DB_CSGMESH

1.5 Multi-Block Objects and Parallel I/O

Individual pieces of mesh created with a number of DBPutXxxmesh() calls can be assembled together into larger, multi-block objects. Likewise for variables and materials defined on these meshes.

In Silo, multi-block objects are really just lists of all the individual pieces of a larger, coherent object. For example, a multi-mesh object is really just a long list of object names, each name being the string passed as the name argument to a DBPutXxxmesh() call.

A key feature of multi-block object is that references to the individual pieces include the option of specifying the name of the Silo file in which a piece is stored. This option is invoked when the colon operator (`:`) appears in the name of an individual piece. All characters before the colon specify the name of a Silo file in the host file system. All characters after a colon specify a directory path within the Silo file where the object lives.

The fact that multi-block objects can reference individual pieces that reside in different Silo files means that Silo, a serial I/O library, can be used very effectively and scalably in parallel without resorting to writing a file per processor. The technique used to affect parallel I/O in this manner with Silo is Multiple Independent File Parallel I/O.

A separate convenience interface, PMPIO, is provided for this purpose. The `PMPIO` interface provides almost all of the functionality necessary to use Silo in the MIF paradigm. The application is required to implement a few callback functions. The `PMPIO` interface is described at the end of this section.

The functions described in this section of the manual include…

1.5.1 DBPutMultimesh()

- **Summary:** Write a multi-block mesh object into a Silo file.
- **C Signature:**

```
int DBPutMultimesh (DBfile *dbfile, char const *name, int nmesh,
    char const * const meshnames[], int const meshtypes[],
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputmmesh(dbid, name, lname, nmesh,
   meshnames, lmeshnames, meshtypes, optlist_id, status)
```

`character*N meshnames` (See *dbset2dstrlen*.)

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the multi-block mesh object.
`nmesh`	Number of meshes pieces (blocks) in this multi-block object.
`meshnames`	Array of length `nmesh` containing pointers to the names of each of the mesh blocks written with a `DBPutXxxmesh()` call. See below for description of how to populate `meshnames` when the pieces are in different files as well as `DBOPT_MB_FILE`
`meshtypes`	Array of length `nmesh` containing the type of each mesh block such as `DB_QUAD_RECT`, `DB_QUAD_CURV`, `DB_UCDMESH`, `DB_POINTMESH`, and `DB_CSGMESH`. Be sure to see description, below, for `DBOPT_MB_BLOCK_TYPE` option to use single, constant value when all pieces are the same type.
`optlist`	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a `NULL` if there are no options.

- **Returned value:**

DBPutMultimesh returns zero on success and -1 on failure.

- **Description:**

The `DBPutMultimesh` function writes a multi-block mesh object into a Silo file. It accepts as input the names of the various sub-meshes (blocks) which are part of this mesh.

The mesh blocks may be stored in different sub-directories within a Silo file and, optionally, even in different Silo files altogether. So, the name of each mesh block is specified using its full Silo path name. The full Silo pathname is the form…

[:]

The existence of a colon (':') anywhere in meshnames[i] indicates that the ith mesh block name is specified using both the Silo filename and the path in the file. All characters before the colon are the Silo file pathname within the file system on which the file(s) reside. Use whatever slash character (" for Windows or '/' for Unix) is appropriate for the underlying file system in this part of the string only. Silo will automatically handle changes in the slash character in this part of the string if this data is ever read on a different file system. All characters after the colon are the path of the object within the Silo file and must use only the '/' slash character.

Use the keyword "EMPTY" for any block for which the associated mesh object does not exist. This convention is often convenient in cases where there are many related multi-block objects and/or that evolve in time in such a way that some blocks do not exist for some times.

The individual mesh names referenced here CANNOT be the names of other multi-block meshes. In other words, it is not valid to create a multi-mesh that references other multi-meshes.

For example, in the case where the are 6 blocks to be assembled into a larger mesh named 'multi-mesh' in the file 'foo.silo' and the blocks are stored in three files as in the figure below,

Figure 0-7: Strings for multi-block objects.

the array of strings to be passed as the meshnames argument of DBPutMultimesh are illustrated. Note that the two pieces of mesh that are in the same file as the multi-mesh object itself, 'multi-mesh', do **not** require the colon and filename option. Only those pieces of the multi-mesh object that are in different files from the one the multi-block object itself resides in require the colon and filename option.

You may pass NULL for the meshnames argument and instead use the namescheme options, DBOPT_MB_FILE_NS and DBOPT_MB_BLOCK_NS described in the table of options, below. This is particularly important for meshes consisting of O(105) or more blocks because it saves substantial memory and I/O time. See *DBMakeNamescheme* for how to specify nameschemes.

Note, however, that with the DBOPT_MB_FILE|BLOCK_NS options, you are specifying only the string that a reader will later use in a call to DBMakeNamescheme() to create a namescheme object suitable for generating the meshnames and not the namescheme object itself.

For convenience, two namescheme options are supported. One namescheme maps block numbers to filenames. The other maps block numbers to object names. A reader is required to then combine both to generate the complete block name for each mesh block. Optionally and where appropriate, one can specify a block namescheme only. External array references may be used in the nameschemes. Any such array names found in the namescheme are assumed to be the names of simple, 1D, integer arrays written with a DBWrite() call and existing in the same directory as the multi-block object . Finally, keep in mind that in the nameschemes, blocks are numbered starting from zero.

If you are using the namescheme options and have EMPTY blocks, since the meshnames argument is NULL, you can use the DBOPT_MB_EMPTY_COUNT|LIST options to explicitly enumerate any empty blocks instead of having to incroporate them into your nameschemes.

Similarly, when the mesh consists of blocks of all the same type, you may pass NULL for the meshtypes argumnt and instead use the DBOPT_MB_BLOCK_TYPE option to specify a single, constant block type for all blocks. This option can result in important savings for large numbers of blocks.

Finally, note that what is described here for the mulitmesh object in the way of name for the individual blocks applies to all multi-block objects (e.g. DBPutMultixxx).

Notes:

The following table describes the options accepted by this function:

Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_BLOCKORIGIN	int	The origin of the block numbers.	1
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_EXTENTS_SIZE	int	Number of values in each extent tuple	0
DBOPT_EXTENTS	double*	Pointer to an array of length nmesh * DBOPT_EXTENTS_SIZE doubles where each group of DBOPT_EXTENTS_SIZE doubles is an extent tuple for the mesh coordinates (see below). DBOPT_EXTENTS_SIZE must be set for this option to work correctly.	NULL
DBOPT_ZONECOUNTS	int*	Pointer to an array of length nmesh indicating the number of zones in each block.	NULL
DBOPT_HAS_EXTERNAL_ZONES	int*	Pointer to an array of length nmesh indicating for each block whether that block has zones external to the whole multi-mesh object. A non-zero value at index i indicates block i has external zones. A value of 0 (zero) indicates it does not.	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_MRGTREE_NAME	char*	Name of the mesh region grouping tree to be associated with this multimesh.	NULL
DBOPT_TV_CONNECTIVITY	int	A non-zero value indicates that the connectivity of the mesh varies with time.	0
DBOPT_DISJOINT_MODE	int	Indicates if any elements in the mesh are disjoint. There are two possible modes. One is DB_ABUTTING indicating that elements abut spatially but actually reference different node ids (but spatially equivalent nodal positions) in the node list. The other is DB_FLOATING where elements neither share nodes in the nodelist nor abut spatially.	DB_NONE
DBOPT_TOPO_DIM	int	Used to indicate the topological dimension of the mesh apart from its spatial dimension.	-1 (not specified)
DBOPT_MB_BLOCK_TYPE	int	Constant block type for all blocks	(not specified)
DBOPT_MB_FILE_NS	char*	Multi-block file namescheme. This is a namescheme, indexed by block number, to generate filename in which each block is stored.	NULL
DBOPT_MB_BLOCK_NS	char*	Multi-block block namescheme. This is a namescheme, indexed by block number, used to generate names of each block object apart from the file in which it may reside.	NULL
DBOPT_MB_EMPTY_LIST	int*	When namescheme options are used, there is no meshnames argument in which to use the keyword 'EMPTY' for empty blocks. Instead, the empty blocks can be enumerated here, indexed from zero.	NULL
DBOPT_MB_EMPTY_COUNT	int	Number of entries in the argument to DBOPT_MB_EMPTY_LIST	0
The options specified below have been deprecated. Use Mesh Region Group (MRG) trees instead.			
DBOPT_GROUPORIGIN	int	The origin of the group numbers.	1
DBOPT_NGROUPS	int	The total number of groups in this multimesh object.	0
DBOPT_ADJACENCY_NAME	char*	Name of a multi-mesh, nodal adjacency object written with a call to adj.	NULL
DBOPT_GROUPINGS_SIZE	int	Number of integer entries in the associated groupings array	0
DBOPT_GROUPINGS	int*	Integer array of length specified by DBOPT_GROUPINGS_SIZE containing information on how different mesh blocks are organized into, pos-	NULL

There is a class of options for DBMulti- objects that is VERY IMPORTANT in helping to accelerate performance in down-stream post-processing tools. We call these Down-stream Performance Options. In order of utility, these options are DBOPT_EXTENTS, DBOPT_MIXLENS and DBOPT_MATLISTS and DBOPT_ZONECOUNTS. Although these options are creating redundant data in the Silo database, the data is stored in a manner that is far more convenient to down-stream applications that read Silo databases. Therefore, the user is strongly encouraged to make use of these options.

Regarding the DBOPT_EXTENTS option, see the notes for *DBPutMultivar*. Note, however, that here the extents are for the coordinates of the mesh.

Regarding the DBOPT_ZONECOUNTS option, this option will help down-stream post-processing tools to select an appropriate static load balance of blocks to processors.

Regarding the DBOPT_HAS_EXTERNAL_ZONES option, this option will help down-stream post-processing tools accelerate computation of external boundaries. When a block is known not to contain any external zones, it can be quickly skipped in the computation. Note that while false positives can negatively effect only performance during downstream external boundary calculations, false negatives will result in serious errors.

In other words, it is ok for a block that does not have external zones to be flagged as though it does. In this case, all that will happen in down-stream post-processing tools is that work to compute external faces that could have been avoided will be wasted. However, it is not ok for a block that has external zones to be flagged as though it does not. In this case, down-stream post-processing tools will skip boundary computation when it should have been computed.

Three options, DBOPT_GROUPINGS_SIZE, DBOPT_GROUPINGS are deprecated. Instead, use MRG trees to handle grouping. Also, see notes regarding *_visit_domain_groups* variable convention.

1.5.2 DBGetMultimesh()

- **Summary:** Read a multi-block mesh from a Silo database.
- **C Signature:**

```
DBmultimesh *DBGetMultimesh (DBfile *dbfile, char const *meshname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
meshname	Name of the multi-block mesh.

- **Returned value:**

  Returns a pointer to a *DBmultimesh* structure on success and NULL on failure.

- **Description:**

  The DBGetMultimesh function allocates a *DBmultimesh* data structure, reads a multi-block mesh from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.5.3 DBPutMultimeshadj()

- **Summary:** Write some or all of a multi-mesh adjacency object into a Silo file.
- **C Signature:**

```
int DBPutMultimeshadj(DBfile *dbfile, char const *name,
    int nmesh, int const *mesh_types, int const *nneighbors,
    int const *neighbors, int const *back,
    int const *nnodes, int const * const nodelists[],
    int const *nzones, int const * const zonelists[],
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-mesh adjacency object.
nmesh	The number of mesh pieces in the corresponding multi-mesh object. This value must be identical in repeated calls to DBPutMultimeshadj.
mesh_types	Integer array of length nmesh indicating the type of each mesh in the corresponding multi-mesh object. This array must be identical to that which is passed in the DBPutMultimesh call and in repeated calls to DBPutMultimeshadj.
nneighbors	Integer array of length nmesh indicating the number of neighbors for each mesh piece. This array must be identical in repeated calls to DBPutMultimeshadj. In the argument descriptions to follow, let . That is, let be the sum of the first k entries in the nneighbors array.
neighbors	Array of integers enumerating for each mesh piece all other mesh pieces that neighbor it. Entries from index to index enumerate the neighbors of mesh piece k. This array must be identical in repeated calls to DBPutMultimeshadj.
back	Array of integers enumerating for each mesh piece, the local index of that mesh piece in each of its neighbors lists of neighbors. Entries from index to index enumerate the local indices of mesh piece k in each of the neighbors of mesh piece k. This argument may be NULL. In any case, this array must be identical in repeated calls to DBPutMultimeshadj.
nnodes	Array of integers indicating for each mesh piece, the number of nodes that it shares with each of its neighbors. Entries from index to index indicate the number of nodes that mesh piece k shares with each of its neighbors. This array must be identical in repeated calls to DBPutMultimeshadj. This argument may be NULL.
nodelists	Array of pointers to arrays of integers. Entries from index to index enumerate the nodes that mesh piece k shares with each of its neighbors. The contents of a specific nodelist array depend on the types of meshes that are neighboring each other (See description below). nodelists[m] may be NULL even if nnodes[m] is non-zero. See below for a description of repeated calls to DBPutMultimeshadj. This argument must be NULL if nnodes is NULL.
nzones	Array of integers indicating for each mesh piece, the number of zones that are adjacent with each of its neighbors. Entries from index to index indicate the number of zones that mesh piece k has adjacent to each of its neighbors. This array must be identical in repeated calls to DBPutMultimeshadj. This argument may be NULL.
zonelists	Array of pointers to arrays of integers. Entries from index to index enumerate the zones that mesh piece k has adjacent with each of its neighbors. The contents of a specific zonelist array depend on the types of meshes that are neighboring each other (See description below). zonelists[m] may be NULL even if nzones[m] is non-zero. See below for a description of repeated calls to DBPutMultimeshadj. This argument must be NULL if nzones is NULL.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

- **Description:**

  Note: This object suffers from scalability issues above about 10^5 blocks.

  The functionality this object provides is now more efficiently and conveniently handled via Mesh Region Grouping (MRG) trees. Users are encouraged to use MRG trees as an alternative to DBPutMultimeshadj(). See *DBMakeMrgtree*.

  DBPutMultimeshadj is another down-stream performance option (See *DBPutMultimesh*). It is an alternative to including ghost-zones (See *DBPutMultimesh*) in the mesh and may help to reduce file size, particularly for unstructured meshes.

  A multi-mesh adjacency object informs down-stream, post-processing tools such as VisIt how nodes and/or zones, should be shared between neighboring mesh blocks of a multi-block mesh to eliminate post-processing discontinuity artifacts along the boundaries between the pieces. If neither this information is provided nor ghost

zones are stored in the file, post-processing tools must then infer this information from global node or zone ids (if they exist) or, worse, by matching coordinates which is a time-consuming and unreliable process.

DBPutMultimeshadj is used to indicate how various mesh pieces in a multi-mesh object abut by specifying for each mesh piece, the nodes it shares with other mesh pieces and/or the zones is has adjacent to other mesh pieces. Note the important distinction in how nodes and zones are classified here. Nodes are shared between mesh pieces while zones are merely adjacent between mesh pieces. In a call to DBPutMultimeshadj, a caller may write information for either shared nodes or adjacent zones, or both.

In practice, applications tend to use the same mesh type for every mesh piece. Thus, for ucd and point meshes, the nodelist (or zonelist) arrays will consists of pairs of integers where the first of the pair identifies a node (or zone) in the given mesh while the second identifies the shared node (or adjacent zone) in a neighbor. Likewise, for quad meshes, the nodelist (or zonelist) arrays will consists of 15 integers the first 6 of which identify a slab of nodes (or zones) in the given quad mesh. The second set of 6 integers identify the slab of shared nodes (or zones) in a neighbor quad mesh and the last 3 integers indicate the orientation of the neighbor quad mesh relative to the given quad mesh. For example the entries (1,2,3) for these 3 integers mean that all axes are aligned. The entries (-2,1,3) mean that the -J axis of the neighbor mesh piece aligns with the +I axis of the given mesh piece, the +I axis of the neighbor mesh piece aligns with the +J axis of the given mesh piece, and the +K axes both align the same way.

The specific contents of a given nodelist array depend on the types of meshes between which it enumerates shared nodes. The table below describes the contents of nodelist array m given the different mesh types that it may enumerate shared nodes for.

	DB_POINT or DB_UCD	DB_QUAD
DB_POINT or DB_UCD	2*nnodes[m] pairs of integers	nnodes[m]+6 integersThe first nnodes[m] integers identify the nodes in the given point or ucd mesh block.The next 6 integers identify ijk bounds of the corresponding nodes in the quad mesh neighbor block.
DB_QUAD	6+nnodes[m] integers.The first 6 integers identify ijk bounds of the nodes in the given quad mesh block.The last nnodes[m] integers identify the nodes in the neighbor point or ucd mesh block.	15 integersThe first set of 6 integers identify ijk bounds of nodes in the given quad mesh block.The second set of 6 integers identify ijk bounds of nodes in the neighbor quad mesh block.The next 3 integers specify the orientation of the neighbor quad mesh block *relative* to the given mesh block.

This function is designed so that it may be called multiple times, each time writing a different portion of multi-mesh adjacency information to the object. On the first call, space is allocated in the Silo file for the entire object. The required space is determined by the contents of all but the nodelists (and/or zonelists) arrays. The contents of the nodelists (and/or zonelists) arrays are the only arguments that are permitted to vary from call to call and then they may vary only in which entries are NULL and non-NULL. Whenever an entry is NULL and the corresponding entry in nnodes (or nzones) array is non-zero, the assumption is that the information is provided in some other call to DBPutMultimeshadj.

1.5.4 DBGetMultimeshadj()

- **Summary:** Get some or all of a multi-mesh nodal adjacency object
- **C Signature:**

```
DBmultimeshadj *DBGetMultimeshadj(DBfile *dbfile,
    char const *name,
    int nmesh, int const *mesh_pieces)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
name	Name of the multi-mesh nodal adjacency object
nmesh	Number of mesh pieces for which nodal adjacency information is being obtained. Pass zero if you want to obtain all nodal adjacency information in a single call.
mesh_pieces	Integer array of length nmesh indicating which mesh pieces nodal adjacency information is desired for. May pass NULL if nmesh is zero.

- **Returned value:**

A pointer to a fully or partially populated *DBmultimeshadj* object or NULL on failure.

- **Description:**

DBGetMultimeshadj returns a nodal adjacency object. This function is designed so that it may be called multiple times to obtain information for different mesh pieces in different calls. The nmesh and mesh_pieces arguments permit the caller to specify for which mesh pieces adjacency information shall be obtained.

1.5.5 DBPutMultivar()

- **Summary:** Write a multi-block variable object into a Silo file.
- **C Signature:**

```
int DBPutMultivar (DBfile *dbfile, char const *name, int nvar,
    char const * const varnames[], int const vartypes[],
    DBoptlist const *optlist);
```

- **Fortran Signature:**

```
integer function dbputmvar(dbid, name, lname, nvar,
   varnames, lvarnames, vartypes, optlist_id, status)
```

character*N varnames (See *dbset2dstrlen*.)

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-block variable.
nvar	Number of variables associated with the multi-block variable.
varnames	Array of length nvar containing pointers to the names of the variables written with DBPutXxxvar() call. See *DBPutMultimesh* for description of how to populate varnames when the pieces are in different files as well as `DBOPT_MB_BLOCK
vartypes	Array of length nvar containing the types of the variables such as DB_POINTVAR, DB_QUADVAR, or DB_UCDVAR. See *DBPutMultimesh*, for DBOPT_MB_BLOCK_TYPE option to use single, constant value when all pieces are the same type.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

- **Returned value:**

  DBPutMultivar returns zero on success and -1 on failure.

- **Description:**

  The `DBPutMultivar` function writes a multi-block variable object into a Silo file.

  Notes:

  The following table describes the options accepted by this function:

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_BLOCKORIGIN	int	The origin of the block numbers.	1
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_EXTENTS_SIZE	int	Number of values in each extent tuple	0
DBOPT_EXTENTS	double*	Pointer to an array of length nvar * DBOPT_EXTENTS_SIZE doubles where each group of DBOPT_EXTENTS_SIZE doubles is an extent tuple (see below). DBOPT_EXTENTS_SIZE must be set for this option to work correctly.	NULL
DBOPT_MMESH_NAME	char*	Name of the multimesh this variable is associated with. Note, this option is very important as down-stream post processing tools are otherwise required to guess as to the mesh a given variable is associated with. Sometimes, the tools can guess wrong.	NULL
DBOPT_TENSOR_RANK	int	Specify the variable type; one of either DB_VARTYPE_SCALAR, DB_VARTYPE_VECTOR DB_VARTYPE_TENSOR, DB_VARTYPE_SYMTENSOR,DB_VARTYPE_ARRAYDB_VARTYPE_LABEL	DB_VARTYPE_SCALAR
DBOPT_REGION_PNAMES	char**	Null-pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be null and is used to indicate the end of the list of names. See *DBOPT_REGION_PNAMES*.	NULL
DBOPT_CONSERVED	int	Indicates if the variable represents a physical quantity that must be conserved under various operations such as interpolation.	0
DBOPT_EXTENSIVE	int	Indicates if the variable represents a physical quantity that is extensive (as opposed to intensive). Note, while it is true that any conserved quantity is extensive, the converse is not true. By default and historically, all Silo variables are treated as intensive.	0
DBOPT_MB_BLOCK_TYPE	int	Constant block type for all blocks	(not specified)
DBOPT_MB_FILE_NS	char*	Multi-block file namescheme. This is a namescheme, indexed by block number, to generate filename in which each block is stored.	NULL
DBOPT_MB_BLOCK_NS	char*	Multi-block block namescheme. This is a namescheme, indexed by block number, used to generate names of each block object apart from the file in which it may reside.	NULL
DBOPT_MB_EMPTY_LIST	int*	When namescheme options are used, there is no varnames argument in which to use the keyword 'EMPTY' for empty blocks. Instead, the empty blocks can be enumerated here, indexed from zero.	NULL
DBOPT_MB_EMPTY_COUNT	int	Number of entries in the argument to DBOPT_MB_EMPTY_LIST	0
DBOPT_MISSING_VALUE	double	Specify a numerical value that is intended to represent "missing values" in the x or y data arrays. Default is DB_MISSING_VALUE_NOT_SET	DB_MISSING_VALUE_N
The options below have been deprecated. Use MRG trees instead.			
DBOPT_GROUPORIGIN	int	The origin of the group numbers.	1
DBOPT_NGROUPS	int	The total number of groups in this multimesh object.	0

Regarding the DBOPT_EXTENTS option, an extent tuple is a tuple of the variable's minimum value(s) followed by the variable's maximum value(s). If the variable is a single, scalar variable, each extent tuple will be 2 values of the form {min,max}. Thus, DBOPT_EXTENTS_SIZE will be 2. If the variable consists of nvars subvariables (e.g. the nvars argument in any of DBPutPointvar, DBPutQuadvar, DBPutUcdvar is greater than 1), then each extent tuple is 2*nvars values of each subvariable's minimum value followed by each subvariable's maximum value. In this case, DBOPT_EXTENTS_SIZE will be 2*nvars.

For example, if we have a multi-var object of a 3D velocity vector on 2 blocks, then DBOPT_EXTENTS_SIZE will be 2*3=6 and the DBOPT_EXTENTS array will be an array of 2*6 doubles organized as follows…

{Vx_min_0, Vy_min_0, Vz_min_0, Vx_max_0, Vy_max_0, Vz_max_0,

Vx_min_1, Vy_min_1, Vz_min_1, Vx_max_1, Vy_max_1, Vz_max_1}

Note that if ghost zones are present in a block, the extents must be computed such that they include contributions from data in the ghost zones. On the other hand, if a variable has mixed components, that is component values on materials mixing within zones, then the extents should **not** include contributions from the mixed variable values.

1.5.6 DBGetMultivar()

- **Summary:** Read a multi-block variable definition from a Silo database.
- **C Signature:**

```
DBmultivar *DBGetMultivar (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the multi-block variable.

- **Returned value:**

  DBGetMultivar returns a pointer to a *DBmultivar* structure on success and NULL on failure.

- **Description:**

  The DBGetMultivar function allocates a *DBmultivar* data structure, reads a multi-block variable from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.5.7 DBPutMultimat()

- **Summary:** Write a multi-block material object into a Silo file.
- **C Signature:**

```
int DBPutMultimat (DBfile *dbfile, char const *name, int nmat,
    char const * const matnames[], DBoptlist const *optlist)
```

- **Fortran Signature:**

```
integer function dbputmmat(dbid, name, lname, nmat,
   matnames, lmatnames, optlist_id, status)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-material object.
nmat	Number of material blocks provided.
matnames	Array of length nmat containing pointers to the names of the material block objects, written with DBPutMaterial(). See *DBPutMultimesh* for description of how to populate matnames when the pieces are in different files as well as `DBOPT_MB_BLOCK
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options

- **Returned value:**

  DBPutMultimat returns zero on success and -1 on error.

- **Description:**

  The DBPutMultimat function writes a multi-material object into a Silo file.

  Notes:

  The following table describes the options accepted by this function:

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_BLOCKORIGIN	int	The origin of the block numbers.	1
DBOPT_NMATNOS	int	Number of material numbers stored in the DBOPT_MATNOS option.	0
DBOPT_MATNOS	int *	Pointer to an array of length DBOPT_NMATNOS containing a complete list of the material numbers used in the Multimat object. DBOPT_NMATNOS must be set for this to work correctly.	NULL
DBOPT_MATNAMES	char**	Pointer to an array of length DBOPT_NMATNOS containing a complete list of the material names used in the Multimat object. DBOPT_NMATNOS must be set for this to work correctly.	NULL
DBOPT_MATCOLORS	char**	Array of strings defining the names of colors to be associated with each material. The color names are taken from the X windows color database. If a color name begins with a'#' symbol, the remaining 6 characters are interpreted as the hexadecimal RGB value for the color. DBOPT_NMATNOS must be set for this to work correctly.	NULL
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_MIXLENS	int*	Array of nmat ints which are the values of the mixlen arguments in each of the individual block's material objects.	
DBOPT_MATCOUNTS	int*	Array of nmat counts indicating the number of materials actually in each block.	NULL
DBOPT_MATLISTS	int*	Array of material numbers in each block. Length is the sum of values in DBOPT_MATCOUNTS. DBOPT_MATCOUNTS must be set for this option to work correctly.	NULL
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_ALLOWMAT0	int	If set to non-zero, indicates that a zero entry in the matlist array is actually not a valid material number but is instead being used to indicate an 'unused' zone.	0
DBOPT_MMESH_NAME	char*	Name of the multimesh this material is associated with. Note, this option is very important as down-stream post processing tools are otherwise required to guess as to the mesh a given material is associated with. Sometimes, the tools can guess wrong.	NULL
DBOPT_MB_FILE_NS	char*	Multi-block file namescheme. This is a namescheme, indexed by block number, to generate filename in which each block is stored.	NULL
DBOPT_MB_BLOCK_NS	char*	Multi-block block namescheme. This is a namescheme, indexed by block number, used to generate names of each block object apart from the file in which it may reside.	NULL
DBOPT_MB_EMPTY_LIST	int*	When namescheme options are used, there is no varnames argument in which to use the keyword 'EMPTY' for empty blocks. Instead, the empty blocks can be enumerated here, indexed from zero.	NULL
DBOPT_MB_EMPTY_COUNT	int	Number of entries in the argument to DBOPT_MB_EMPTY_LIST	0
The options below have been deprecated. Use MRG trees instead.			
DBOPT_GROUPORIGIN	int	The origin of the group numbers.	1
DBOPT_NGROUPS	int	The total number of groups in this multimesh object.	0

Regarding the DBOPT_MIXLENS option, this option will help down-stream post-processing tools to select an appropriate load balance of blocks to processors. Material mixing and material interface reconstruction have a big effect on cost of certain post-processing operations.

Regarding the DBOPT_MATLISTS options, this option will give down-stream post-processing tools better knowledge of how materials are distributed among blocks.

1.5.8 DBGetMultimat()

- **Summary:** Read a multi-block material object from a Silo database
- **C Signature:**

```
DBmultimat *DBGetMultimat (DBfile *dbfile, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
name	Name of the multi-block material object

- **Returned value:**

  DBGetMultimat returns a pointer to a *DBmultimat* structure on success and NULL on failure.

- **Description:**

  The DBGetMultimat function allocates a *DBmultimat* data structure, reads a multi-block material from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.5.9 DBPutMultimatspecies()

- **Summary:** Write a multi-block species object into a Silo file.
- **C Signature:**

```
int DBPutMultimatspecies (DBfile *dbfile, char const *name,
    int nspec, char const * const specnames[],
    DBoptlist const *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-block species structure.
nspec	Number of species objects provided.
specnames	Array of length nspec containing pointers to the names of each of the species. See *DBPutMultimesh* for description of how to populate specnames when the pieces are in different files as well as `DBOPT_MB_BLOCK
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

- **Returned value:**

  DBPutMultimatspecies returns zero on success and -1 on failure.

- **Description:**

  The DBPutMultimatspecies function writes a multi-block material species object into a Silo file. It accepts as input descriptions of the various sub-species (blocks) which are part of this mesh.

  Notes:

  The following table describes the options accepted by this function:

  Optlist options:

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_BLOCKORIGIN	int	The origin of the block numbers.	1
DBOPT_MATNAME	char*	Character string defining the name of the multi-block material with which this object is associated.	NULL
DBOPT_NMAT	int	The number of materials in the associated material object.	0
DBOPT_NMATSPEC	int *	Array of length DBOPT_NMAT containing the number of material species associated with each material. DBOPT_NMAT must be set for this to work correctly.	NULL
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_TIME	float	Problem time value.	0.0
DBOPT_DTIME	double	Problem time value.	0.0
DBOPT_HIDE_FROM_GUI	int	Specify a non-zero value if you do not want this object to appear in menus of downstream tools	0
DBOPT_SPECNAMES	char**	Array of strings defining the names of the individual species. DBOPT_NMATSPEC must be set for this to work correctly. The length of this array is the sum of the values in the argument to the DBOPT_NMATSPEC option.	NULL
DBOPT_SPECCOLORS	char**	Array of strings defining the names of colors to be associated with each species. The color names are taken from the X windows color database. If a color name begins with a'#' symbol, the remaining 6 characters are interpreted as the hexadecimal RGB value for the color. DBOPT_NMATSPEC must be set for this to work correctly. The length of this array is the sum of the values in the argument to the DBOPT_NMATSPEC option.	NULL
DBOPT_MB_FILENS	char*	Multi-block file namescheme. This is a namescheme, indexed by block number, to generate filename in which each block is stored.	NULL
DBOPT_MB_BLOCKNS	char*	Multi-block block namescheme. This is a namescheme, indexed by block number, used to generate names of each block object apart from the file in which it may reside.	NULL
DBOPT_MB_EMPTY_LIST	int*	When namescheme options are used, there is no varnames argument in which to use the keyword 'EMPTY' for empty blocks. Instead, the empty blocks can be enumerated here, indexed from zero.	NULL
DBOPT_MB_EMPTY_COUNT	int	Number of entries in the argument to DBOPT_MB_EMPTY_LIST	0
The options below have been deprecated. Use MRG trees instead.			
DBOPT_GROUPORIGIN	int	The origin of the group numbers.	1
DBOPT_NGROUPS	int	The total number of groups in this multimesh object.	0

1.5.10 DBGetMultimatspecies()

- **Summary:** Read a multi-block species from a Silo database.
- **C Signature:**

```
DBmultimesh *DBGetMultimatspecies (DBfile *dbfile,
    char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-block material species.

- **Returned value:**

  DBGetMultimatspecies returns a pointer to a *DBmultimatspecies* structure on success and NULL on failure.

- **Description:**

  The DBGetMultimatspecies function allocates a *DBmultimatspecies* data structure, reads a multi-block material species from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.5.11 DBOpenByBcast()

- **Summary:** Specialized, read-only open method for parallel applications needing all processors to read all (or most of) a given Silo file

- **C Signature:**

```
DBfile *DBOpenByBcast(char const *filename, MPI_Comm comm,
    int rank_of_root)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
filename	name of the Silo file to open
comm	MPI communicator to use for the broadcast operation
rank_of_root	MPI rank of the processor in the communicator comm that shall serve as the root of the broadcast (typically 0).

- **Returned value:**

  A Silo database file handle just as returned from DBOpen or DBCreate except that the file is read-only. Available only for reading Silo files produced via the HDF5 driver.

- **Description:**

  This is an experimental interface! It is not fully integrated into the Silo library.

  In many parallel applications, there is a master or root file that all processors need all (or most of) the information from in order to bootstrap opening a larger collection of Silo files (similar to PMPIO)

  This method is provided to perform the operation in a way that is friendly to the underlying file system by opening the file on a single processor using the HDF5 file-in-core feature and then broadcasting the "file" buffer to all processors which then turn around and open the buffer as a file. In this way, the application can avoid many processors interacting with and potentially overwhelming the file system.

But, there are some important limitations too. First, it works only for reading Silo files. Next, the entire Silo file is loaded into a buffer in memory and the broadcast in its entirety to all other processors. If only some processors need only some of the data from the file, then there is potentially a lot of memory and communication wasted for parts of the file not used on various processors.

When the file is closed with `DBClose()` all memory used by the file is released.

This method is not compiled into libsilo[h5].a. Instead, you are required to obtain the bcastopen.c source file (which is installed to the include dir of the Silo install point) and compile it into your application and then include a line of this form…

```
extern `DBfile` *DBOpenByBcast(char const *, MPI_Comm, int);
```

in your application.

Note that you can find an example of its use in the Silo source release "tests" directory in the source file "bcastopen_test.c"

1.5.12 PMPIO_Init()

- **Summary:** Initialize a MIF Parallel I/O interaction with the Silo library
- **C Signature:**

```
PMPIO_baton_t *PMPIO_Init(int numFiles, PMPIO_iomode_t ioMode,
    MPI_Comm mpiComm, int mpiTag,
    PMPIO_CreateFileCallBack createCb,
    PMPIO_OpenFileCallBack openCb,
    PMPIO_CloseFileCallBack closeCB,
    void *userData)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
numFiles	The number of individual Silo files to generate. Note, this is the number of parallel I/O streams that will be running simultaneously during I/O. A value of 1 causes PMPIO to behave serially. A value equal to the number of processors causes PMPIO to create a file-per-processor. Both values are unwise. For most parallel HPC platforms, values between 8 and 64 are appropriate.
ioMode	Choose one of either PMPIO_READ or PMPIO_WRITE. Note, you can not use PMPIO to handle both read and write in the same interaction.
mpiComm	The MPI communicator you would like PMPIO to use when passing the tiny baton messages it needs to coordinate access to the underlying Silo files. See documentation on MPI for a description of MPI communicators.
mpiTag	The MPI message tag you would like PMPIO to use when passing the tiny baton messages it needs to coordinate access to the underlying Silo files.
createCB	The file creation callback function. This is a function you implement that PMPIO will call when the first processor in each group needs to create the Silo file for the group. It is needed only for PMPIO_WRITE operations. If default behavior is acceptable, pass PMPIO_DefaultCreate here.
openCB	The file open callback function. This is a function you implement that PMPIO will call when the second and subsequent processors in each group need to open a Silo file. It is needed for both PMPIO_READ and PMPIO_WRITE operations. If default behavior is acceptable, pass PMPIO_DefaultOpen here.
closeCB	The file close callback function. This is a function you implement that PMPIO will call when a processor in a group needs to close a Silo file. If default behavior is acceptable, pass PMPIO_DefaultClose here.
userData	[OPT] Arbitrary user data that will be passed back to the various callback functions. Pass NULL(0) if this is not needed.

- **Returned value:**

  A pointer to a PMPIO_baton_t object to be used in subsequent PMPIO calls on success. NULL on failure.

- **Description:**

  The PMPIO interface was designed to be separate from the Silo library. To use it, you must include the PMPIO header file, pmpio.h, after the MPI header file, mpi.h, in your application. This interface was designed to work with any serial library and not Silo specifically. For example, these same routines can be used with raw HDF5 or PDB files if so desired.

  The PMPIO interface decomposes a set of P processors into N groups and then provides access, in parallel, to a separate Silo file per group. This is the essence of MIF Parallel I/O.

  For PMPIO_WRITE operations, each processor in a group creates its own Silo sub-directory within the Silo file to write its data to. At any one moment, only one processor from each group has a file open for writing. Hence, the I/O is serial within a group. However, because a processor in each of the N groups is writing to its own Silo file, simultaneously, the I/O is parallel across groups.

  The number of files, N, can be chosen wholly independently of the total number of processors permitting the application to tune N to the underlying file system. If N is set to 1, the result will be serial I/O to a single file. If N is set to P, the result is one file per processor. Both of these are poor choices.

  Typically, one chooses N based on the number of available I/O channels. For example, a parallel application running on 2,000 processors and writing to a file system that supports 8 parallel I/O channels could select N=8 and achieve nearly optimum I/O performance and create only 8 Silo files.

  On every processor, the sequence of PMPIO operations takes the following form…

```
PMPIO_baton_t *bat = PMPIO_Init(...);

dbFile = (DBfile *) PMPIO_WaitForBaton(bat, ...);
```

(continues on next page)

(continued from previous page)

```
/* local work (e.g. DBPutXxx() calls) for this processor
.
.
.
*/

PMPIO_HandOffBaton(bat, ...);

PMPIO_Finish(bat);
```

For a given PMPIO group of processors, only one processor in the group is in the *local work* block of the above code. All other processors have either completed it or are waiting their predecessor to finish. However, every PMPIO group will have one processor working in the *local work* block, concurrently, to different files.

After PMPIO_Finish(), there is still one final step that PMPIO DOES **not** HELP with. That is the creation of the multi-block objects that reference the individual pieces written by all the processors with DBPutXXX calls in the "local work" part of the above sequence. It is the application's responsibility to correctly assembly the names of all these pieces and then create the multi-block objects that reference them. Ordinarily, the application designates one processor to write these multi-block objects and one of the N Silo files to write them to. Again, this last step is not something PMPIO will help with.

MIF Parallel I/O is a simple and effective I/O strategy that has been used by codes like Ale3d and SAMRAI for many years and has shown excellent scaling behavior. A drawback of this approach is, of course, that multiple files are generated. However, when used appropriately, this number of files is typically small (e.g. 8 to 64). In addition, our experience has been that concurrent, parallel I/O to a single file which also supports sufficient variation in size, shape and pattern of I/O requests from processor to processor is a daunting challenge to perform scalably. So, while MIF Parallel I/O is not truly concurrent, parallel I/O, it has demonstrated that it is not only highly flexible and highly scalable but also very easy to implement and for these reasons, often a superior choice to true, concurrent, parallel I/O.

1.5.13 PMPIO_CreateFileCallBack()

- **Summary:** The PMPIO file creation callback
- **C Signature:**

```
typedef void *(*PMPIO_CreateFileCallBack)(const char *fname,
    const char *dname, void *udata);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
fname	The name of the Silo file to create.
dname	The name of the directory within the Silo file to create.
udata	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

- **Returned value:**

  A void pointer to the created file handle.

- **Description:**

  This defines the PMPIO file creation callback interface.

  Your implementation of this file creation callback should minimally do the following things.

  For PMPIO_WRITE operation, your implementation should DBCreate() a Silo file of name fname, DBMkDir() a directory of name dname for the first processor of a group to write to and DBSetDir() to that directory.

  For PMPIO_READ operations, your implementation of this callback is never called.

  The PMPIO_DefaultCreate function does only the minimal work, returning a void pointer to the created DBfile Silo file handle.

1.5.14 PMPIO_OpenFileCallBack()

- **Summary:** The PMPIO file open callback
- **C Signature:**

```
typedef void *(*PMPIO_OpenFileCallBack)(const char *fname,
    const char *dname, PMPIO_iomode_t iomode, void *udata);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
fname	The name of the Silo file to open.
dname	The name of the directory within the Silo file to work in.
iomode	The iomode of this PMPIO interaction. This is the value passed as ioMode argument to PMPIO_Init().
udate	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

- **Returned value:**

  A void pointer to the opened file handle that was.

- **Description:**

  This defines the PMPIO open file callback.

  Your implementation of this open file callback should minimally do the following things.

  For PMPIO_WRITE operations, it should DBOpen() the Silo file named fname, DBMkDir() a directory named dname and DBSetDir() to directory dname.

  For PMPIO_READ operations, it should DBOpen() the Silo file named fname and then DBSetDir() to the directory named dname.

  The PMPIO_DefaultOpen function does only the minimal work, returning a void pointer to the opened DBfile Silo handle.

1.5.15 PMPIO_CloseFileCallBack()

- **Summary:** The PMPIO file close callback
- **C Signature:**

```
typedef void  (*PMPIO_CloseFileCallBack)(void *file, void *udata);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	void pointer to the file handle (DBfile pointer).
udata	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

- **Returned value:**

  void

- **Description:**

  This defines the PMPIO close file callback interface.

  Your implementation of this callback function should simply close the file. It us up to the implementation to know the correct time of the file handle passed as the void pointer file.

  The PMPIO_DefaultClose function simply closes the Silo file.

1.5.16 PMPIO_WaitForBaton()

- **Summary:** Wait for exclusive access to a Silo file
- **C Signature:**

```
void *PMPIO_WaitForBaton(PMPIO_baton_t *bat,
    const char *filename, const char *dirname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
filename	The name of the Silo file this processor will create or open.
dirname	The name of the directory within the Silo file this processor will work in.

- **Returned value:**

NULL (0) on failure. Otherwise, for PMPIO_WRITE operations the return value is whatever the create or open file callback functions return. For PMPIO_READ operations, the return value is whatever the open file callback function returns.

- **Description:**

  All processors should call this function as the next PMPIO function to call following a call to PMPIO_Init().

  For all processors that are the *first* processors in their groups, this function will return immediately after having called the file creation callback specified in PMPIO_Init(). Typically, this callback will have created a file with the name filename and a directory in the file with the name dirname as well as having set the current working directory to dirname.

  For all processors that are not the *first* in their groups, this call will block, waiting for the processor preceding it to finish its work on the Silo file for the group and pass the baton to the next processor.

  A typical naming convention for filename is something like "my_file_%03d.silo" where the "%03d" is replaced with the group rank (See *PMPIO_GroupRank* of the processor. Likewise, a typical naming convention for dirname is something like "domain_%03d" where the "%03d" is replaced with the rank-in-group (See *PMPIO_RankInGroup* of the processor.

1.5.17 PMPIO_HandOffBaton()

- **Summary:** Give up all access to a Silo file
- **C Signature:**

```
void PMPIO_HandOffBaton(const PMPIO_baton_t *bat, void *file)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
file	A void pointer to the Silo DBfile object.

- **Returned value:**

  void

- **Description:**

  When a processor has completed all its work on a Silo file, it gives up access to the file by calling this function. This has the effect of closing the Silo file and then passing the baton to the next processor in the group.

1.5.18 PMPIO_Finish()

- **Summary:** Finish a MIF Parallel I/O interaction with the Silo library
- **C Signature:**

```
void PMPIO_Finish(PMPIO_baton *bat)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().

- **Returned value:**

  void

- **Description:**

  After a processor has finished a PMPIO interaction with the Silo library, call this function to free the baton object associated with the interaction.

1.5.19 PMPIO_GroupRank()

- **Summary:** Obtain group rank (e.g. which PMPIO group) of a processor
- **C Signature:**

```
int PMPIO_GroupRank(const PMPIO_baton_t *bat, int rankInComm)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
rankInComm	Rank of processor in the MPI communicator passed in PMPIO_Init() for which group rank is to be queried.

- **Returned value:**

  The 'group rank' of the queiried processor. In other words, the group number of the queried processor, indexed from zero.

- **Description:**

  This is a convenience function to help applications identify which PMPIO group a given processor belongs to.

1.5.20 PMPIO_RankInGroup()

- **Summary:** Obtain the rank of a processor *within* its PMPIO group
- **C Signature:**

```
int PMPIO_RankInGroup(const PMPIO_baton_t *bat, int rankInComm)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
rankInComm	Rank of the processor in the MPI communicator used in PMPIO_Init() to be queried.

- **Returned value:**

  The rank of the queried processor within its PMPIO group.

- **Description:**

  This is a convenience function for applications to determine which processor a given processor is *within* its PMPIO group.

1.6 Part Assemblies, AMR, Slide Surfaces, Nodesets and Other Arbitrary Mesh Subsets

This section of the API manual describes Mesh Region Grouping (MRG) trees and Groupel Maps. MRG trees describe the decomposition of a mesh into various regions such as parts in an assembly, materials (even mixing materials), element blocks, processor pieces, nodesets, slide surfaces, boundary conditions, etc. Groupel maps describe the, problem sized, details of the subsetted regions. MRG trees and groupel maps work hand-in-hand in efficiently (and scalably) characterizing the various subsets of a mesh.

MRG trees are associated with (e.g. bound to) the mesh they describe using the DBOPT_MRGTREE_NAME optlist option in the DBPutXxxmesh() calls. MRG trees are used both to describe a multi-mesh object and then again, to describe individual pieces of the multi-mesh.

In addition, once an MRG tree has been defined for a mesh, variables to be associated with the mesh can be defined on only specific subsets of the mesh using the DBOPT_REGION_PNAMES optlist option in the DBPutXxxvar() calls.

Because MRG trees can be used to represent a wide variety of subsetting functionality and because applications have still to gain experience using MRG trees to describe their subsetting applications, the methods defined here are design to be as free-form as possible with few or no limitations on, for example, naming conventions of the various types of subsets. It is simply impossible to know a priori all the different ways in which applications may wish to apply MRG trees to construct subsetting information.

For this reason, where a specific application of MRG trees is desired (to represent materials for example), we document the naming convention an application must use to affect the representation.

1.6.1 DBMakeMrgtree()

- **Summary:** Create and initialize an empty mesh region grouping tree
- **C Signature:**

```
DBmrgtree *DBMakeMrgtree(int mesh_type, int info_bits,
    int max_children, DBoptlist const *opts)
```

- **Fortran Signature:**

```
integer function dbmkmrgtree(mesh_type, info_bits, max_children, optlist_id,
   tree_id)
```

  returns handle to newly created tree in tree_id.

- **Arguments:**

Arg name	Description
mesh_type	The type of mesh object the MRG tree will be associated with. An example would be DB_MULTIMESH, DB_QUADMESH, DB_UCDMESH.
info_bits	UNUSED
max_children	Maximum number of immediate children of the root.
opts	Additional options

- **Returned value:**

  A pointer to a new *DBmrgtree* object on success and NULL on failure

- **Description:**

  This function creates a Mesh Region Grouping (MRG) tree used to define different regions in a mesh.

  An MRG tree is used to describe how a mesh is composed of regions such as materials, parts in an assembly, levels in an adaptive refinement hierarchy, nodesets, slide surfaces, boundary conditions, as well as many other kinds of regions. An example is shown in Figure 0-8 on page.

  Figure 0-8: Example of MRGTree

  In a multi-mesh setting, an MRG tree describing all of the subsets of the mesh is associated with the top-level multimesh object. In addition, separate MRG trees representing the relevant portions of the top-level MRG tree are also associated with each block.

  MRG trees can be used to describe a wide variety of subsets of a mesh. In the paragraphs below, we outline the use of MRG trees to describe a variety of common subsetting scenarios. In some cases, a specific naming convention is required to fully specify the subsetting scenario.

  The paragraphs below describe how to utilize an MRG tree to describe various common kinds of decompositions and subsets.

  Multi-Block Grouping (obsoletes DBOPT_GROUPING options for DBPutMultimesh, and *_visit_domain_groups* convention.

  A multi-block grouping is the assignment of the blocks of a multi-block mesh (e.g. the mesh objects created with DBPutXxxmesh() calls and enumerated by name in a DBPutMultimesh() call) to one of several groups. Each group in the grouping represents several blocks of the multi-block mesh. Historically, support for such a grouping in Silo has been limited in a couple of ways. First, only a single grouping could be defined for a multi-block mesh. Second, the grouping could not be hierarchically defined. MRG trees, however, support both multiple groupings and hierarchical groupings.

In the MRG tree, define a child node of the root named "groupings." All desired groupings shall be placed under this node in the tree.

For each desired grouping, define a groupel map where the number of segments of the map is equal to the number of desired groups. Map segment i will be of groupel type DB_BLOCKCENT and will enumerate the blocks to be assigned to group i. Next, add regions (either an array of regions or one at a time) to the MRG tree, one region for each group and specify the groupel map name and other map parameters to be associated with these regions.

Figure 0-9: Examples of MRG trees for single and multiple groupings.

In the diagram above, for the multiple grouping case, two groupel map objects are defined; one for each grouping. For the 'A' grouping, the groupel map consists of 4 segments (all of which are of groupel type DB_BLOCKCENT) one for each grouping in 'side', 'top', 'bottom' and 'front.' Each segment identifies the blocks of the multi-mesh (at the root of the MRG tree) that are in each of the 4 groups. For the 'B' grouping, the groupel map consists of 2 segments (both of type DB_BLOCKCENT), for each grouping in 'skinny' and 'fat'. Each segment identifies the blocks of the multi-mesh that are in each group.

If, in addition to defining which blocks are in which groups, an application wishes to specify specific nodes and/or zones of the group that comprise each block, additional groupel maps of type DB_NODECENT or DB_ZONECENT are required. However, because such groupel maps are specified in terms of nodes and/or zones, these groupel maps need to be defined on an MRG tree that is associated with an individual mesh block. Nonetheless, the manner of representation is analogous.

Multi-Block Neighbor Connectivity (obsoletes DBPutMultimeshadj):

Multi-block neighbor connectivity information describes the details of how different blocks of a multi-block mesh abut with shared nodes and/or adjacent zones. For a given block, multi-block neighbor connectivity information lists the blocks that share nodes (or have adjacent zones) with the given block and then, for each neighboring block, also lists the specific shared nodes (or adjacent zones).

If the underlying mesh type is structured (e.g. DBPutQuadmesh() calls were used to create the individual mesh blocks), multi-block neighbor connectivity information can be scalably represented entirely at the multi-block level in an MRG tree. Otherwise, it cannot and it must be represented at the individual block level in the MRG tree. This section will describe both scenarios. Note that these scenarios were previously handled with the now deprecated DBPutMultimeshadj() call. That call, however, did not have favorable scalaing behavior for the unstructured case.

The first step in defining multi-block connectivity information is to define a top-level MRG tree node named "neighbors." Underneath this point in the MRG tree, all the information identifying multi-block connectivity will be added.

Next, create a groupel map with number of segments equal to the number of blocks. Segment i of the map will by of type DB_BLOCKCENT and will enumerate the neighboring blocks of block i. Next, in the MRG tree define a child node of the root named "neighborhoods". Underneath this node, define an array of regions, one for each block of the multiblock mesh and associate the groupel map with this array of regions.

For the structured grid case, define a second groupel map with number of segments equal to the number of blocks. Segment i of the map will be of type DB_NODECENT and will enumerate the slabs of nodes block i shares with each of its neighbors in the same order as those neighbors are listed in the previous groupel map. Thus, segment i of the map will be of length equal to the number of neighbors of block i times 6 (2 ijk tuples specifying the lower and upper bounds of the slab of shared nodes).

For the unstructured case, it is necessary to store groupel maps that enumerate shared nodes between shared blocks on MRG trees that are associated with the individual blocks and **not** the multi-block mesh itself. However, the process is otherwise the same.

In the MRG tree to be associated with a given mesh block, create a child of the root named "neighbors." For each neighboring block of the given mesh block, define a groupel map of type DB_NODECENT, enumerating the nodes in the current block that are shared with another block (or of type DB_ZONECENT enumerating the nodes in the

current block that abut another block). Underneath this node in the MRG tree, create a region representing each neighboring block of the given mesh block and associate the appropriate groupel map with each region.

Multi-Block, Structured Adaptive Mesh Refinement:

In a structured AMR setting, each AMR block (typically called a "patch" by the AMR community), is written by a DBPutQuadmesh() call. A DBPutMultimesh() call groups all these blocks together, defining all the individual blocks of mesh that comprise the complete AMR mesh.

An MRG tree, or portion thereof, is used to define which blocks of the multi-block mesh comprise which levels in the AMR hierarchy as well as which blocks are refinements of other blocks.

First, the grouping of blocks into levels is identical to multi-block grouping, described previously. For the specific purpose of grouping blocks into levels, a different top-level node in the MRG needs to be defined named "amr-levels." Below this node in the MRG tree, there should be a set of regions, each region representing a separate refinement level. A groupel map of type DB_BLOCKCENT with number of segments equal to number of levels needs to be defined and associated with each of the regions defined under the "amr-levels' region. The ith segment of the map will enumerate those blocks that belong to the region representing level i. In addition, an MRG variable defining the refinement ratios for each level named "amr-ratios" must be defined on the regions defining the levels of the AMR mesh.

For the specific purpose of identifying which blocks of the multi-block mesh are refinements of a given block, another top-level region node is added to the MRG tree called "amr-refinements". Below the "amr-refinements" region node, an array of regions representing each block in the multi-block mesh should be defined. In addition, define a groupel map with a number of segments equal to the number of blocks. Map segment i will be of groupel type DB_BLOCKCENT and will define all those blocks which are immediate refinements of block i. Since some blocks, with finest resolution do not have any refinements, the map segments defining the refinements for these blocks will be of zero length.

1.6.2 DBAddRegion()

- **Summary:** Add a region to an MRG tree
- **C Signature:**

```
int DBAddRegion(DBmrgtree *tree, char const *reg_name,
    int info_bits, int max_children, char const *maps_name,
    int nsegs, int const *seg_ids, int const *seg_lens,
    int const *seg_types, DBoptlist const *opts)
```

- **Fortran Signature:**

```
integer function dbaddregion(tree_id, reg_name, lregname, info_bits,
   max_children, maps_name, lmaps_name, nsegs, seg_ids, seg_lens,
   seg_types, optlist_id, status)
```

- **Arguments:**

Arg name	Description
tree	The MRG tree object to add a region to.
reg_name	The name of the new region.
info_bits	UNUSED
max_children	Maximum number of immediate children this region will have.
maps_name	[OPT] Name of the groupel map object to associate with this region. Pass NULL if none.
nsegs	[OPT] Number of segments in the groupel map object specified by the maps_name argument that are to be associated with this region. Pass zero if none.
seg_ids	[OPT] Integer array of length nsegs of groupel map segment ids. Pass NULL (0) if none.
seg_lens	[OPT] Integer array of length nsegs of groupel map segment lengths. Pass NULL (0) if none.
seg_types	[OPT] Integer array of length nsegs of groupel map segment element types. Pass NULL (0) if none. These types are the same as the centering options for variables; DB_ZONECENT, DB_NODECENT, DB_EDGECENT, DB_FACECENT and DB_BLOCKCENT (for the blocks of a multimesh)
opts	[OPT] Additional options. Pass NULL (0) if none.

- **Returned value:**

  A positive number on success; -1 on failure

- **Description:**

  Adds a single region node to an MRG tree below the current working region (See *DBSetCwr*).

  If you need to add a large number of similarly purposed region nodes to an MRG tree, consider using the more efficient DBAddRegionArray() function although it does have some limitations with respect to the kinds of groupel maps it can reference.

  A region node in an MRG tree can represent either a specific region, a group of regions or both all of which are determined by actual use by the application.

  Often, a region node is introduced to an MRG tree to provide a separate namespace for regions to be defined below it. For example, to define material decompositions of a mesh, a region named "materials" is introduced as a top-level region node in the MRG tree. Note that in so doing, the region node named "materials" does **not** really represent a distinct region of the mesh. In fact, it represents the union of all material regions of the mesh and serves as a place to define one, or more, material decompositions.

  Because MRG trees are a new feature in Silo, their use in applications is not fully defined and the implementation here is designed to be as free-form as possible, to permit the widest flexibility in representing regions of a mesh. At the same time, in order to convey the semantic meaning of certain kinds of information in an MRG tree, a set of pre-defined region names is described below.

Region Naming Convention	Meaning
"materials"	Top-level region below which material decomposition information is defined. There can be multiple material decompositions, if so desired. Each such decomposition would be rooted at a region named "material_" underneath the "materials" region node.
"groupings"	Top-level region below which multi-block grouping information is defined. There can be multiple groupings, if so desired. Each such grouping would be rooted at a region named "grouping_" underneath the "groupings" region node.
"amrlevels"	Top-level region below which Adaptive Mesh Refinement level groupings are defined.
"amrrefinements"	Top-level region below which Adaptive Mesh Refinment refinement information is defined. This where the information indicating which blocks are refinements of other blocks is defined.
"neighbors"	Top-level region below which multi-block adjacency information is defined.

When a region is being defined in an MRG tree to be associated with a multi-block mesh, often the groupel type of the maps associated with the region are of type `DB_BLOCKCENT`.

1.6.3 DBAddRegionArray()

- **Summary:** Efficiently add multiple, like-kind regions to an MRG tree
- **C Signature:**

```
int DBAddRegionArray(DBmrgtree *tree, int nregn,
    char const * const *regn_names, int info_bits,
    char const *maps_name, int nsegs, int const *seg_ids,
    int const *seg_lens, int const *seg_types,
    DBoptlist const *opts)
```

- **Fortran Signature:**

```
integer function dbaddregiona(tree_id, nregn, regn_names,         lregn_names,
   info_bits, maps_name, lmaps_name, nsegs,         seg_ids, seg_lens,
   seg_types, optlist_id, status)
```

- **Arguments:**

Arg name	Description
tree	The MRG tree object to add the regions to.
nregn	The number of regions to add.
regn_names	This is either an array of nregn pointers to character string names for each region or it is an array of 1 pointer to a character string specifying a printf-style naming scheme for the regions. The existence of a percent character ('%') (used to introduce conversion specifications) anywhere in regn_names[0] will indicate the latter mode.The latter mode is almost always preferable, especially if nergn is large (say more than 100). See below for the format of the printf-style naming string.
info_bits	UNUSED
maps_name	[OPT] Name of the groupel maps object to be associated with these regions. Pass NULL (0) if none.
nsegs	[OPT] The number of groupel map segments to be associated with each region. Note, this is a per-region value. Pass 0 if none.
seg_ids	[OPT] Integer array of length nsegs*nregn groupel map segment ids. The first nsegs ids are associated with the first region. The second nsegs ids are associated with the second region and so fourth. In cases where some regions will have fewer than nsegs groupel map segments associated with them, pass -1 for the corresponding segment ids. Pass NULL (0) if none.
seg_lens	[OPT] Integer array of length nsegs*nregn indicating the lengths of each of the groupel maps. In cases where some regions will have fewer than nsegs groupel map segments associated with them, pass 0 for the corresponding segment lengths. Pass NULL (0) if none.
seg_types	[OPT] Integer array of length nsegs*nregn specifying the groupel types of each segment. In cases where some regions will have fewer than nsegs groupel map segments associated with them, pass 0 for the corresponding segment lengths. Pass NULL (0) if none.
opts	[OPT] Additional options. Pass NULL (0) if none.

- **Returned value:**

  A positive number on success; -1 on failure

- **Description:**

  Use this function instead of DBAddRegion() when you have a large number of similarly purposed regions to add to an MRG tree AND you can deal with the limitations of the groupel maps associated with these regions.

  The key limitation of the groupel map associated with a region created with DBAddRegionArray() array and a groupel map associated with a region created with DBAddRegion() is that every region in the region array must reference nseg map segments (some of which can of course be of zero length).

  Adding a region array is a substantially more efficient way to add regions to an MRG tree than adding them one at a time especially when a printf-style naming convention is used to specify the region names.

  The existence of a percent character ('%') anywhere in regn_names[0] indicates that a printf-style namescheme is to be used. The format of a printf-style namescheme to specify region names is described in the documentation of DBMakeNamescheme() (See *DBMakeNamescheme*)

  Note that the names of regions within an MRG tree are not required to obey the same variable naming conventions as ordinary Silo objects (See *DBVariableNameValid*.) except that MRG region names can in no circumstance contain either a semi-colon character (';') or a new-line character ('\n').

1.6.4 DBSetCwr()

- **Summary:** Set the current working region for an MRG tree
- **C Signature:**

```
int DBSetCwr(DBmrgtree *tree, char const *path)
```

- **Fortran Signature:**

```
integer function dbsetcwr(tree, path, lpath)
```

- **Arguments:**

Arg name	Description
tree	The MRG tree object.
path	The path to set.

- **Returned value:**

  Positive, depth in tree, on success, -1 on failure.

- **Description:**

  Sets the current working region of the MRG tree. The concept of the current working region is completely analogous to the current working directory of a file system.

  Notes:

  Currently, this method is limited to settings up or down the MRG tree just one level. That is, it will work only when the path is the name of a child of the current working region or is "..". This limitation will be relaxed in a future release.

1.6.5 DBGetCwr()

- **Summary:** Get the current working region of an MRG tree
- **C Signature:**

```
char const *GetCwr(DBmrgtree *tree)
```

- **Arguments:**

Arg name	Description
tree	The MRG tree.

- **Returned value:**

  A pointer to a string representing the name of the current working region (not the full path name, just current region name) on success; NULL (0) on failure.

1.6.6 DBPutMrgtree()

- **Summary:** Write a completed MRG tree object to a Silo file
- **C Signature:**

```
int DBPutMrgtree(DBfile *file, const char const *name,
    char const *mesh_name, DBmrgtree const *tree,
    DBoptlist const *opts)
```

- **Fortran Signature:**

```
int dbputmrgtree(dbid, name, lname, mesh_name,
   lmesh_name, tree_id, optlist_id, status)
```

- **Arguments:**

Arg name	Description
file	The Silo file handle
name	The name of the MRG tree object in the file.
mesh_name	The name of the mesh the MRG tree object is associated with.
tree	The MRG tree object to write.
opts	[OPT] Additional options. Pass NULL (0) if none.

- **Returned value:**

  Positive or zero on success, -1 on failure.

- **Description:**

  After using DBPutMrgtree to write the MRG tree to a Silo file, the MRG tree object itself must be freed using DBFreeMrgtree().

1.6.7 DBGetMrgtree()

- **Summary:** Read an MRG tree object from a Silo file
- **C Signature:**

```
DBmrgtree *DBGetMrgtree(DBfile *file, const char *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle
name	The name of the MRG tree object in the file.

- **Returned value:**

  A pointer to a *DBmrgtree* object on success; NULL (0) on failure.

1.6.8 DBFreeMrgtree()

- **Summary:** Free the memory associated by an MRG tree object
- **C Signature:**

```
void DBFreeMrgtree(DBmrgtree *tree)
```

- **Fortran Signature:**

```
integer function dbfreemrgtree(tree_id)
```

- **Arguments:**

Arg name	Description
tree	The MRG tree object to free.

- **Returned value:**

  void

- **Description:**

  Frees all the memory associated with an MRG tree.

1.6.9 DBMakeNamescheme()

- **Summary:** Create a DBnamescheme object for on-demand name generation
- **C Signature:**

```
DBnamescheme *DBMakeNamescheme(const char *ns_str, ...)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
ns_str	The namescheme string as described below.
...	The remaining arguments take various forms. See description below.

- **Description:**

  The remaining arguments after ns_str take one of three forms depending on how the caller wants external array references, if any are present in the format substring of ns_str to be handled.

  In the first form, the format substring of ns_str involves no externally referenced arrays and so there are no additional arguments other than the ns_str string itself.

  In the second form, the caller has all externally referenced arrays needed in the format substring of ns_str already in memory and simply passes their pointers here as the remaining arguments in the same order in which they appear in the format substring of ns_str. The arrays are bound to the returned namescheme object and should not be freed until after the caller is done using the returned namescheme object. In this case, DBFreeNamescheme() does not free these arrays and the caller is required to explicitly free them.

In the third form, the caller makes a request for the Silo library to find in a given file, read and bind to the returned namescheme object all externally referenced arrays in the format substring of `ns_str`. To achieve this, the caller passes a 3-tuple of the form `(void*) 0, (DBfile*) file, (char*) mbobjpath` as the remaining arguments. The initial `(void*)0` is required. The `(DBfile*)file` is the database handle of the Silo file in which all externally referenced arrays exist. The third `(char*)mbobjpath`, which may be `NULL`, is the path within the file, either relative to the file's current working directory, or absolute, at which the multi-block object holding the `ns_str` was found in the file. All necessary externally referenced arrays must exist within the specified file using either relative paths from multi-block object's home directory or the file's current working directory or absolute paths. In this case `DBFreeNamescheme()` also frees memory associated with these arrays.

A namescheme defines a mapping between the non-negative integers (e.g. the natural numbers) and a sequence of strings such that each string to be associated with a given integer (n) can be generated on the fly from printf-style formatting involving simple expressions. Nameschemes are most often used to define names of regions in region arrays or to define names of multi-block objects. The format of a printf-style namescheme is as follows…

The first character of `ns_str` is treated as the *delimiter character definition*. Wherever this delimiter character appears (except as the first character), this will indicate the end of one substring within `ns_str` and the beginning of a next substring. The delimiter character cannot be any of the characters used in the expression language (see below) for defining expressions to generate names of a namescheme. The delimiter character divides `ns_str` into one or more substrings.

The first substring of `ns_strs` (that is the characters from position 1 to the first delimiter character after its definition at index 0) will contain the complete printf-style format string the namescheme will generate. The remaining substrings will contain simple expressions, one for each conversion specifier found in the format substring, which when evaluated will be used as the corresponding argument in an sprintf call to generate the actual name, when and if needed, on demand.

The expression language for building up the arguments to be used along with the printf-style format string is pretty simple.

It supports the '+', '-', '*', '/', '%' (modulo), '|', '&', '^' integer operators and a variant of the question-mark-colon operator, '? : :' which requires an extra, terminating colon.

It supports grouping via '(' and ')' characters.

It supports grouping of characters into arbitrary strings via the single quote character (‘). Any characters appearing between enclosing single quotes are treated as a literal string suitable for an argument to be associated with a %s-type conversion specifier in the format string.

It supports references to external, integer valued arrays introduced via a '#' character appearing before an array's name and external, string valued arrays introduced via a '$' character appearing before an array's name.

Finally, the special operator 'n' appearing in an expression represents a *natural number* within the sequence of names (zero-origin index).

Except for singly quoted strings which evaluate to a literal string suitable for output via a %s type conversion specifier, and $-type external array references which evaluate to an external string, all other expressions are treated as evaluating to integer values suitable for any of the integer conversion specifiers (%[ouxXdi]) which may be used in the format substring.

Here are some examples…

`"|slide_%s|(n%2)?'leader':'follower':"`

The delimiter character is `|`. The format substring is `slide_%s`. The expression substring for the argument to the first (and only in this case) conversion specifier (`%s`) is `(n%2)?'leader':'follower':` When this expression is evaluated for a given region, the region's natural number will be inserted for `n`. The modulo operation with `2` will be applied. If that result is non-zero, the `?::` expression will evaluate to `'leader'`. Otherwise, it will evaluate to `'follower'`. Note there is also a *terminating* colon for the `?::` operator. This naming scheme might be useful for an array of regions representing, alternately, the two halves of a contact

surface. Note also for the `?::` operator, the caller can assume that only the sub-expression corresponding to the whichever half of the operator is satisfied is actually evaluated.

"Hblock_%02dx%02dHn/16Hn%16"
The delimiter character is `H`. The format substring is `block_%02dx%02d`. The expression substring for the argument to the first conversion specifier (`%02d`) is `n/256`. The expression substring for the argument to the second conversion specifier (also `%02d`) is `n%16`. When this expression is evaluated, the region's natural number will be inserted for `n` and the div and mod operators will be evaluated. This naming scheme might be useful for a region array of 256 regions to be named as a 2D array of regions with names like "block_09x11".

"@domain_%03d@n"
The delimiter character is `@`. The format substring is `domain_%03d`. The expression substring for the argument to the one and only conversion specifier is `n`. When this expression is evaluated, the region's natural number is inserted for `n`. This results in names like "domain_000", "domain_001", etc.

"@domain_%03d@n+1"
This is just like the case above except that region names begin with "domain_001" instead of "domain_000". This might be useful to deal with different indexing origins; Fortran vs. C.

"|foo_%03dx%03d|#P[n]|#U[n%4]"
The delimiter character is `|`. The format substring is `foo_%03dx%03d`. The expression substring for the first argument is an external array reference `#P[n]` where the index into the array is just the natural number, `n`. The expression substring for the second argument is another external array reference, `#U[n%4]` where the index is an expression `n%4` on the natural number `n`.

If the caller is handling externally referenced arrays explicitly, because `P` is the first externally referenced array in the format string, a pointer to `P` must be the first to appear in the `...` list of additional args to `DBMakeNamescheme`. Similarly, because `U` appears as the second externally referenced array in the format string, a pointer to `U` must appear second in the `...` as in `DBMakeNamescheme("|foo_%03dx%03d|#P[n]|#U[n%4]", P, U)`

Alternatively, if the caller wants the Silo library to find `P` and `U` in a Silo file, read the arrays from the file and bind them into the namescheme automatically, then `P` and `U` must be simple arrays in the current working directory of the file that is passed in as the 3-tuple `"(int) 0, (DBfile *) dbfile, 0"` in the `...` argument to `DBMakeNamescheme` as in `DBMakeNamescheme("|foo_%03dx%03d|#P[n]|#U[n%4]", 0, dbfile, 0)`.

Use `DBFreeNamescheme()` to free up the space associated with a namescheme. Also note that there are numerous examples of nameschemes in "tests/nameschemes.c" in the Silo source release tarball.

1.6.10 DBGetName()

- **Summary:** Generate a name from a `DBnamescheme` object
- **C Signature:**

```
char const *DBGetName(DBnamescheme *ns, long long natnum)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
natnum	Natural number of the entry in a namescheme to be generated. Must be greater than or equal to zero.

- **Returned value:**

  A string representing the generated name. The returned string should **not** be free'd by the caller. If there are problems with the namescheme, the string could be of length zero (e.g. the first character is a null terminator).

- **Description:**

  Once a namescheme has been created via DBMakeNamescheme, this function can be used to generate names at will from the namescheme. The caller must **not** free the returned string.

  Silo maintains a tiny circular buffer of (32) names constructed and returned by this function so that multiple evaluations in the same expression do not wind up overwriting each other. A call to `DBGetName(0,0)` will free up all memory associated with this tiny circular buffer.

1.6.11 DBGetIndex()

- **Summary:** Reverse engineer a name from a namescheme to obtain field indices
- **C Signature:**

```
long long DBGetIndex(char const *dbns_name_str, int field, int base)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbns_name_str	An arbitrary string but most commonly with one or more substring *fields* consisting entirely of digits (in an arbitrary base numbering system) representing different *fields* generated from various conversion specifiers in a namescheme.
field	Of the various substrings of digits, numbered starting at zero from left to right in dbns_name_str, this argument selects which of the digit substrings is to be processed
base	The numeric base to use to interpret the digit substring field passed to strtoll(). Passing 0 (to let strtoll automatically determine base) is ok but potentially risky.

- **Returned value:**

  A field's converted value on success; LLONG_MAX on failure.

- **Description:**

Nameschemes often generate names of the form `foo_0050.0210.silo`. Sometimes, it is useful to obtain the decimal values of the numbered fields in such a string. Calling `DBGetIndex("foo_0050.0210.silo", 0, 10)` will retrieve the first digit field, "0050", and convert it to a decimal number using a number base of 10. Calling `DBGetIndex("foo_0050.0210.silo", 1, 10)` will retrieve the second digit field, "0210", and convert it to a decimal number using a number base of 10.

Passing a base of 0 is allowed but may also result in unintended outcomes. In the example string, `foo_0050.0210.silo`, calling `DBGetIndex("foo_0050.0210.silo", 0, 0)` will return 40, not 50. This is because a leading 0 is interpreted as an octal number.

In the string `block_030x021.silo` (which could have the interpretation of a block at 2D index [30,21] in a 2D arrangement of blocks), the `0x021` will be interpreted as a digit field in hexadecimal format which may not have been the intention. The solution is to ensure the string has characters separating fields that are not also interpreted as part of an integer constant in the C programming language.

1.6.12 DBPutMrgvar()

- **Summary:** Write variable data to be associated with (some) regions in an MRG tree
- **C Signature:**

```
int DBPutMrgvar(DBfile *file, char const *name,
    char const *mrgt_name,
    int ncomps, char const * const *compnames,
    int nregns, char const * const *reg_pnames,
    int datatype, void const * const *data,
    DBoptlist const *opts)
```

- **Fortran Signature:**

```
integer function dbputmrgv(dbid, name, lname, mrgt_name,
   lmrgt_name, ncomps, compnames, lcompnames, nregns, reg_names,
   lreg_names, datatype, data_ids, optlist_id, status)
```

`character*N compnames` (See *dbset2dstrlen*)

`character*N reg_names` (See *dbset2dstrlen*)

`int* data_ids` (use [dbmkptr](fortran.md#dbmkptr) to get id for each pointer)

- **Arguments:**

Arg name	Description
file	Silo database file handle.
name	Name of this mrgvar object.
tname	name of the mrg tree this variable is associated with.
ncomps	An integer specifying the number of variable components.
compnames	[OPT] Array of ncomps pointers to character strings representing the names of the individual components. Pass NULL(0) if no component names are to be specified.
nregns	The number of regions this variable is being written for.
reg_pnames	Array of nregns pointers to strings representing the pathnames of the regions for which the variable is being written. If nregns>1 and reg_pnames[1]==NULL, it is assumed that reg_pnames[i]=NULL for all i>0 and reg_pnames[0] contains either a printf-style naming convention for all the regions to be named or, if reg_pnames[0] is found to contain no printf-style conversion specifications, it is treated as the pathname of a single region in the MRG tree that is the parent of all the regions for which attributes are being written.
data	Array of ncomps pointers to variable data. The pointer, data[i] points to an array of nregns values of type datatype.
opts	Additional options.

- **Returned value:**

  Zero on success; -1 on failure.

- **Description:**

  Sometimes, it is necessary to associate variable `data` with regions in an MRG tree. This call allows an application to associate variable `data` with a bunch of different regions in one of several ways all of which are determined by the contents of the `reg_pnames` argument.

  Variable `data` can be associated with all of the immediate children of a given region. This is the most common case. In this case, reg_pnames[0] is the `name` of the parent region and reg_pnames[i] is set to `NULL` for all i>0.

  Alternatively, variable `data` can be associated with a bunch of regions whose names conform to a common, printf-style naming scheme. This is typical of regions created with the `DBPutRegionArray()` call. In this case, reg_pnames[0] is the `name` of the parent region and reg_pnames[i] is set to `NULL` for all i>0 and, in addition, reg_pnames[0] is a specially formatted, printf-style string, for naming the regions. See *DBAddRegionArray*. for a discussion of the regn_names argument format.

  Finally, variable `data` can be associated with a bunch of arbitrarily named regions. In this case, each region's `name` must be explicitly specified in the `reg_pnames` array.

  Because MRG trees are a new feature in Silo, their use in applications is not fully defined and the implementation here is designed to be as free-form as possible, to permit the widest flexibility in representing regions of a mesh. At the same time, in order to convey the semantic meaning of certain kinds of information in an MRG tree, a set of pre-defined MRG variables is descirbed below.

Variable Naming Convention	Meaning
"amr-ratios"	An integer variable of 3 components defining the refinement ratios (rx, ry, rz) for an `AMR` mesh. Typically, the refinement ratios can be specified on a level-by-level basis. In this case, this variable should be defined for nregns=<# of levels> on the level regions underneath the "amr-levels" grouping. However, if refinment ratios need to be defined on an individual patch basis instead, this variable should be defined on the individual patch regions under the "amr-refinements" groupings.
"ijk-orientations"	An integer variable of 3 components defined on the individual blocks of a multi-block mesh defining the orientations of the individual blocks in a large, ijk indexing space (Ares convention)
"-extents"	A double precision variable defining the block-by-block extents of a multi-block variable. If =="coords", then it defines the spatial extents of the mesh itself. Note, this convention obsoletes the `DBOPT_XXX_EXTENTS` options on DBPutMultivar/DBPutMultimesh calls.

Don't forget to associate the resulting region variable object(s) with the MRG tree by using the `DBOPT_MRGV_ONAMES` and `DBOPT_MRGV_RNAMES` options in the `DBPutMrgtree()` call.

1.6.13 DBGetMrgvar()

- **Summary:** Retrieve an MRG variable object from a silo file
- **C Signature:**

```
DBmrgvar *DBGetMrgvar(DBfile *file, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`file`	Silo database `file` handle.
`name`	The `name` of the region variable object to retrieve.

- **Returned value:**

  A pointer to a *DBmrgvar* object on success; `NULL` (0) on failure.

1.6.14 DBPutGroupelmap()

- **Summary:** Write a groupel map object to a Silo file
- **C Signature:**

```
int DBPutGroupelmap(DBfile *file, char const *name,
    int num_segs, int const *seg_types, int const *seg_lens,
    int const *seg_ids, int const * const *seg_data,
    void const * const *seg_fracs, int fracs_type,
    DBoptlist const *opts)
```

- **Fortran Signature:**

```
integer function dbputgrplmap(dbid, name, lname, num_segs,
   seg_types, seg_lens, seg_ids, seg_data_ids, seg_fracs_ids, fracs_type,
   optlist_id, status)
```

`integer* seg_data_ids` (use dbmkptr to get id for each pointer) `integer* seg_fracs_ids` (use dbmkptr to get id for each pointer)

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
name	The name of the groupel map object in the file.
nsegs	The number of segments in the map.
seg_types	Integer array of length nsegs indicating the groupel type associated with each segment of the map; one of DB_BLOCKCENT, DB_NODECENT, DB_ZONECENT, DB_EDGECENT, DB_FACECENT.
seg_lens	Integer array of length nsegs indicating the length of each segment
seg_ids	[OPT] Integer array of length nsegs indicating the identifier to associate with each segment. By default, segment identifiers are 0...negs-1. If default identifiers are sufficient, pass NULL (0) here. Otherwise, pass an explicit list of integer identifiers.
seg_data	The groupel map data, itself. An array of nsegs pointers to arrays of integers where array seg_data[i] is of length seg_lens[i].
seg_fracs	[OPT] Array of nsegs pointers to floating point values indicating fractional inclusion for the associated groupels. Pass NULL (0) if fractional inclusions are not required. If, however, fractional inclusions are required but on only some of the segments, pass an array of pointers such that if segment i has no fractional inclusions, seg_fracs[i]=NULL(0). Fractional inclusions are useful for, among other things, defining groupel maps involving mixing materials.
fracs_type	[OPT] data type of the fractional parts of the segments. Ignored if seg_fracs is NULL (0).
opts	Additional options

- **Returned value:**

  Zero on success; -1 on failure.

- **Description:**

  By itself, an MRG tree is not sufficient to fully characterize the decomposition of a mesh into various regions. The MRG tree serves only to identify the regions and their relationships in gross terms. This frees MRG trees from growing linearly (or worse) with problem size.

  All regions in an MRG tree are ultimately defined, in detail, by enumerating a primitive set of Grouping Elements (groupels) that comprise the regions. A groupel map is the object used for this purpose. The problem-sized information needed to fully characterize the regions of a mesh is stored in groupel maps.

  The grouping elements or groupels are the individual pieces of mesh which, when enumerated, define specific regions.

  For a multi-mesh object, the groupels are whole blocks of the mesh. For Silo's other mesh types such as ucd or quad mesh objects, the groupels can be nodes (0d), zones (2d or 3d depending on the mesh dimension), edges (1d) and faces (2d).

  The groupel map concept is best illustrated by example. Here, we will define a groupel map for the material case illustrated in Figure 0-6.

  Figure 0-10: Example of using groupel map for (mixing) materials.

  In the example in the above figure, the groupel map has the behavior of representing the clean and mixed parts of the material decomposition by enumerating in alternating segments of the map, the clean and mixed parts for each successive material.

1.6.15 DBGetGroupelmap()

- **Summary:** Read a groupel map object from a Silo file
- **C Signature:**

```
DBgroupelmap *DBGetGroupelmap(DBfile *file, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
name	The name of the groupel map object to read.

- **Returned value:**

  A pointer to a *DBgroupelmap* object on success. NULL (0) on failure.

1.6.16 DBFreeGroupelmap()

- **Summary:** Free memory associated with a groupel map object
- **C Signature:**

```
void DBFreeGroupelmap(DBgroupelmap *map)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
map	Pointer to a DBgroupel map object.

- **Returned value:**

  void

1.6.17 DBOPT_REGION_PNAMES

- **Summary:** Option list option for defining variables on specific regions of a mesh
- **C Signature:**

  DBOPT_REGION_PNAMES char**

  A null-pointer terminated array of pointers to strings specifying the pathnames of regions in the mrg tree for the associated mesh where the variable is defined. If there is no mrg tree associated with the mesh, the names specified here will be assumed to be material names of the material object associated with the mesh. The last pointer in the array must be NULL and is used to indicate the end of the list of names.

All of Silo's DBPutXxxvar() calls support the DBOPT_REGION_PNAMES option to specify the variable on only some region(s) of the associated mesh. However, the use of the option has implications regarding the ordering of the values in the vars[] arrays passed into the DBPutXxxvar() functions. This section explains the ordering requirements.

Ordinarily, when the DBOPT_REGION_PNAMES option is not being used, the order of the values in the vars arrays passed here is considered to be one-to-one with the order of the nodes (for DB_NODECENT centering) or zones (for DB_ZONECENT centering) of the associated mesh. However, when the DBOPT_REGION_PNAMES option is being used, the order of values in the vars[] is determined by other conventions described below.

If the DBOPT_REGION_PNAMES option references regions in an MRG tree, the ordering is one-to-one with the groupel's identified in the groupel map segment(s) (of the same groupel type as the variable's centering) associated with the region(s); all of the segment(s), in order, of the groupel map of the first region, then all of the segment(s) of the groupel map of the second region, and so on. If the set of groupel map segments for the regions specified include the same groupel multiple times, then the vars[] arrays will wind up needing to include the same value, multiple times.

The preceding ordering convention works because the ordering is explicitly represented by the order in which groupels are identified in the groupel maps. However, if the DBOPT_REGION_PNAMES option references material name(s) in a material object created by a DBPutMaterial() call, then the ordering is not explicitly represented. Instead, it is based on a traversal of the mesh zones restricted to the named material(s). In this case, the ordering convention requires further explanation and is described below.

For DB_ZONECENT variables, as one traverses the zones of a mesh from the first zone to the last, if a zone contains a material listed in DBOPT_REGION_PNAMES (wholly or partially), that zone is considered in the traversal and placed conceptually in an ordered list of traversed zones. In addition, if the zone contains the material only partially, that zone is also placed conceptually in an ordered list of traversed mixed zones. In this case, the values in the vars[] array must be one-to-one with this traversed zones list. Likewise, the values of the mixvars[] array must be one-to-one with the traversed mixed zones list. However, in the special case that the list of materials specified in DBOPT_REGION_PNAMES is of size one (1), an additional optimization is supported.

For the special case that the list of materials defined in DBOPT_REGION_PNAMES is of size one (1), the requirement to specify separate values for zones containing the material only partially in the mixvars[] array is removed. In this case, if the mixlen arg is zero (0) in the cooresponding DBPutXXXvar() call, only the vars[] array, which is one-to-one with (all) traversed zones containing the material either cleanly or partially, will be used. The reason this works is that in the single material case, there is only ever one zonal variable value per zone regardless of whether the zone contains the material cleanly or partially. For DB_NODECENT variables, the situation is complicated by the fact that materials are zone-centric but the variable being defined is node-centered.

So, an additional level of local traversal over a zone's nodes is required. In this case, as one traverses the zones of a mesh from the first zone to the last, if a zone contains a material listed in DBOPT_REGION_PNAMES (wholly or partially), then that zone's nodes are traversed according to the ordering specified in the *node, edge and face ordering for zoo-type UCD zone shape diagram*. On the first encounter of a node, that node is considered in the traversal and placed conceptually in an ordered list of traversed nodes. The values in the vars[] array must be one-to-one with this traversed nodes list. Because we are not aware of any cases of node-centered variables that have mixed material components, there is no analogous traversed mixed nodes list.

For DBOPT_EDGECENT and DBOPT_FACECENT variables, the traversal is handled similarly. That is, the list of zones for the mesh is traversed and for each zone found to contain one of the materials listed in DBOPT_REGION_PNAMES, the zone's edge's (or face's) are traversed in local order specified in the *node, edge and face ordering for zoo-type UCD zone shape diagram*.

For Quad meshes, there is no explicit list of zones (or nodes) comprising the mesh. So, the notion of traversing the zones (or nodes) of a Quad mesh requires further explanation. If the mesh's nodes (or zones) were to be traversed, which would be the first? Which would be the second? Unless the DBOPT_MAJORORDER option was used, the answer is that the traversal is identical to the standard C programming language storage convention for multi-dimensional arrays often called row-major storage order. That is, was we traverse through the list of nodes (or zones) of a Quad mesh, we encounter first node with logical index [0,0,0], then [0,0,1], then

[0,0,2]...[0,1,0]...etc. A traversal of zones would behave similarly. Traversal of edges or faces of a quad mesh would follow the description with *DBPutQuadvar*.

1.7 Object Allocation, Free and IsEmpty Tests

This section describes methods to allocate and initialize many of Silo's objects.

1.7.1 DBAllocXxx()

- **Summary:** Allocate and initialize a Silo structure.
- **C Signature:**

```
DBcompoundarray  *DBAllocCompoundarray (void)
DBcsgmesh        *DBAllocCsgmesh (void)
DBcsgvar         *DBAllocCsgvar (void)
DBcurve          *DBAllocCurve (void)
DBcsgzonelist    *DBAllocCSGZonelist (void)
DBdefvars        *DBAllocDefvars (void)
DBedgelist       *DBAllocEdgelist (void)
DBfacelist       *DBAllocFacelist (void)
DBmaterial       *DBAllocMaterial (void)
DBmatspecies     *DBAllocMatspecies (void)
DBmeshvar        *DBAllocMeshvar (void)
DBmultimat       *DBAllocMultimat (void)
DBmultimatspecies *DBAllocMultimatspecies (void)
DBmultimesh      *DBAllocMultimesh (void)
DBmultimeshadj   *DBAllocMultimeshadj (void)
DBmultivar       *DBAllocMultivar (void)
DBpointmesh      *DBAllocPointmesh (void)
DBquadmesh       *DBAllocQuadmesh (void)
DBquadvar        *DBAllocQuadvar (void)
DBucdmesh        *DBAllocUcdmesh (void)
DBucdvar         *DBAllocUcdvar (void)
DBzonelist       *DBAllocZonelist (void)
DBphzonelist     *DBAllocPHZonelist (void)
DBnamescheme     *DBAllocNamescheme(void);
DBgroupelmap     *DBAllocGroupelmap(int, DBdatatype)
```

- **Fortran Signature:**

```
None
```

- **Returned value:**

  These allocation functions return a pointer to a newly allocated and initialized structure on success and NULL on failure.

- **Description:**

  The allocation functions allocate a new structure of the requested type, and initialize all values to NULL or zero. There are counterpart functions for freeing structures of a given type (see DBFree....

1.7.2 DBFreeXxx()

- **Summary:** Release memory associated with a Silo structure.
- **C Signature:**

```
void DBFreeCompoundarray (DBcompoundarray *x)
void DBFreeCsgmesh (DBcsgmesh *x)
void DBFreeCsgvar (DBcsgvar *x)
void DBFreeCSGZonelist (DBcsgzonelist *x)
void DBFreeCurve(DBcurve *);
void DBFreeDefvars (DBdefvars *x)
void DBFreeEdgelist (DBedgelist *x)
void DBFreeFacelist (DBfacelist *x)
void DBFreeMaterial (DBmaterial *x)
void DBFreeMatspecies (DBmatspecies *x)
void DBFreeMeshvar (DBmeshvar *x)
void DBFreeMultimesh (DBmultimesh *x)
void DBFreeMultimeshadj (DBmultimeshadj *x)
void DBFreeMultivar (DBmultivar *x)
void DBFreeMultimat(DBmultimat *)
void DBFreeMultimatspecies(DBmultimatspecies *)
void DBFreePointmesh (DBpointmesh *x)
void DBFreeQuadmesh (DBquadmesh *x)
void DBFreeQuadvar (DBquadvar *x)
void DBFreeUcdmesh (DBucdmesh *x)
void DBFreeUcdvar (DBucdvar *x)
void DBFreeZonelist (DBzonelist *x)
void DBFreePHZonelist (DBphzonelist *x)
void DBFreeNamescheme(DBnamescheme *)
void DBFreeMrgvar(DBmrgvar *mrgv)
void DBFreeMrgtree(DBmrgtree *tree)
void DBFreeGroupelmap(DBgroupelmap *map)
```

- **Arguments:**

Arg name	Description
x	A pointer to a structure which is to be freed. Its type must correspond to the type in the function name.
Fortran Equivalent:	None

- **Returned value:**

  These free functions return zero on success and -1 on failure.

- **Description:**

  The free functions release the given structure as well as all memory pointed to by these structures. This is the preferred method for releasing these structures. There are counterpart functions for allocating structures of a given type (see DBAlloc…). The functions will not fail if a NULL pointer is passed to them.

1.7.3 DBIsEmpty()

- **Summary:** Query a object returned from Silo for "emptiness"
- **C Signature:**

```
int     DBIsEmptyCurve(DBcurve const *curve)
int     DBIsEmptyPointmesh(DBpointmesh const *msh)
int     DBIsEmptyPointvar(DBpointvar const *var)
int     DBIsEmptyMeshvar(DBmeshvar const *var)
int     DBIsEmptyQuadmesh(DBquadmesh const *msh)
int     DBIsEmptyQuadvar(DBquadvar const *var)
int     DBIsEmptyUcdmesh(DBucdmesh const *msh)
int     DBIsEmptyFacelist(DBfacelist const *fl)
int     DBIsEmptyZonelist(DBzonelist const *zl)
int     DBIsEmptyPHZonelist(DBphzonelist const *zl)
int     DBIsEmptyUcdvar(DBucdvar const *var)
int     DBIsEmptyCsgmesh(DBcsgmesh const *msh)
int     DBIsEmptyCSGZonelist(DBcsgzonelist const *zl)
int     DBIsEmptyCsgvar(DBcsgvar const *var)
int     DBIsEmptyMaterial(DBmaterial const *mat)
int     DBIsEmptyMatspecies(DBmatspecies const *spec)
```

- **Arguments:**

Arg name	Description
x	Pointer to a silo object structure to be queried

- **Description:**

  These functions return non-zero if the object is indeed empty and zero otherwise. When DBSetAllowEmptyObjects() is enabled by a writer, it can produce objects in the file which contain useful metadata but no "problems-sized" data. These methods can be used by a reader to determine if an object read from a file is empty.

1.8 Calculational and Utility

This section of the API manual describes some calculational and utility functions that can be used to compute things such as facelists or catentate an array of strings into a single string for simple output with DBWrite().

There are also functions to compute a *DBmaterial* object from *dense* volume fraction arrays and vice versa.

1.8.1 DBCalcExternalFacelist()

- **Summary:** Calculate an external facelist for a UCD mesh.
- **C Signature:**

```
DBfacelist *DBCalcExternalFacelist (int nodelist[], int nnodes,
    int origin, int shapesize[],
    int shapecnt[], int nshapes, int matlist[],
    int bnd_method)
```

- **Fortran Signature:**

```
integer function dbcalcfl(nodelist, nnodes, origin, shapesize,
   shapecnt, nshapes, matlist, bnd_method, flid)
```

returns the pointer-id of the created object in flid.

- **Arguments:**

Arg name	Description
nodelist	Array of node indices describing mesh zones.
nnodes	Number of nodes in associated mesh.
origin	Origin for indices in the nodelist array. Should be zero or one.
shapesize	Array of length nshapes containing the number of nodes used by each zone shape.
shapecnt	Array of length nshapes containing the number of zones having each shape.
nshapes	Number of zone shapes.
matlist	Array containing material numbers for each zone (else NULL).
bnd_method	Method to use for calculating external faces. See description below.

- **Returned value:**

DBCalcExternalFacelist returns a *DBfacelist* pointer on success and NULL on failure.

- **Description:**

The DBCalcExternalFacelist function calculates an external facelist from the zonelist and zone information describing a UCD mesh. The calculation of the external facelist is controlled by the bnd_method parameter as defined in the table below:

bnd_method	Meaning
0	Do not use material boundaries when computing external faces. The matlist parameter can be replaced with NULL.
1	In addition to true external faces, include faces on material boundaries between zones. Faces get generated for both zones sharing a common face. This setting should not be used with meshes that contain mixed material zones. If this setting is used with meshes with mixed material zones, all faces which border a mixed material zone will be include. The matlist parameter must be provided.

For a description of how the nodes for the allowed shapes are enumerated, see *DBPutUcdmesh*.

1.8.2 DBCalcExternalFacelist2()

- **Summary:** Calculate an external facelist for a UCD mesh containing ghost zones.
- **C Signature:**

```
DBfacelist *DBCalcExternalFacelist2 (int nodelist[], int nnodes,
    int low_offset, int hi_offset, int origin,
    int shapetype[], int shapesize[],
    int shapecnt[], int nshapes, int matlist[], int bnd_method)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
nodelist	Array of node indices describing mesh zones.
nnodes	Number of nodes in associated mesh.
lo_offset	The number of ghost zones at the beginning of the nodelist.
hi_offset	The number of ghost zones at the end of the nodelist.
origin	Origin for indices in the nodelist array. Should be zero or one.
shapetype	Array of length nshapes containing the type of each zone shape. See description below.
shapesize	Array of length nshapes containing the number of noes used by each zone shape.
shapecnt	Array of length nshapes containing the number of zones having each shape.
nshapes	Number of zone shapes.
matlist	Array containing material numbers for each zone (else NULL).
bnd_method	Method to use for calculating external faces. See description below.

- **Returned value:**

  DBCalcExternalFacelist2 returns a *DBfacelist* pointer on success and NULL on failure.

- **Description:**

  The DBCalcExternalFacelist2 function calculates an external facelist from the zonelist and zone information describing a UCD mesh. The calculation of the external facelist is controlled by the bnd_method parameter as defined in the table below:

bnd_method	Meaning
0	Do not use material boundaries when computing external faces. The matlist parameter can be replaced with NULL.
1	In addition to true external faces, include faces on material boundaries between zones. Faces get generated for both zones sharing a common face. This setting should not be used with meshes that contain mixed material zones. If this setting is used with meshes with mixed material zones, all faces which border a mixed material zone will be included. The matlist parameter must be provided.

  The allowed shape types are described in the following table:

Type	Description
DB_ZONETYPE_BEAM	A line segment
DB_ZONETYPE_POLYGON	A polygon where nodes are enumerated to form a polygon
DB_ZONETYPE_TRIANGLE	A triangle
DB_ZONETYPE_QUAD	A quadrilateral
DB_ZONETYPE_POLYHEDRON	A polyhedron with nodes enumerated to form faces and faces are enumerated to form a polyhedron
DB_ZONETYPE_TET	A tetrahedron
DB_ZONETYPE_PYRAMID	A pyramid
DB_ZONETYPE_PRISM	A prism
DB_ZONETYPE_HEX	A hexahedron

  For a description of how the nodes for the allowed shapes are enumerated, see *DBPutUcdmesh*.

1.8.3 DBStringArrayToStringList()

- **Summary:** Utility to catentate a group of strings into a single, semi-colon delimited string.
- **C Signature:**

```
void DBStringArrayToStringList(char const * const *strArray,
    int n, char **strList, int *m)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
strArray	Array of strings to catenate together. Note that it can be ok if some entries in strArray are the empty string, "" or NULL (0).
n	The number of entries in strArray. Passing -1 here indicates that the function should count entries in strArray until reaching the first NULL entry. In this case, embedded NULLs (0s) in strArray are, of course, not allowed.
strList	The returned catenated, semi-colon separated, single, string.
m	The returned length of strList.

- **Description:**

  This is a utility function to facilitate writing of an array of strings to a file. This function will take an array of strings and catenate them together into a single, semi-colon delimited list of strings.

  Some characters are **not** permitted in the input strings. These are '\n', '\0' and ';' characters.

  This function can be used together with DBWrite() to easily write a list of strings to the a Silo database.

1.8.4 DBStringListToStringArray()

- **Summary:** Given a single, semi-colon delimited string, de-catenate it into an array of strings.
- **C Signature:**

```
char **DBStringListToStringArray(char *strList, int n,
    int handleSlashSwap, int skipFirstSemicolon)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
`strList`	A semi-colon separated, single string. Note that this string is modified by the call. If the caller doesn't want this, it will have to make a copy before calling.
`n`	The expected number of individual strings in `strList`. Pass -1 here if you have no aprior knowledge of this number. Knowing the number saves an additional pass over `strList`.
`handleSlashSwap`	a boolean to indicate if slash characters should be swapped as per differences in windows/linux file systems.
`This is specific to Silo's internal handling of strings used in multi-block objects. So, you should pass zero (0) here.`	skipFirstSemicolon
`a boolean to indicate if the first semicolon in the string should be skipped.`	This is specific to Silo's internal usage for legacy compatibility. You should pass zero (0) here.

- **Description:**

  This function performs the reverse of *DBStringArrayToStringList*.

1.8.5 DBFreeStringArray()

- **Summary:** Free an array of strings
- **C Signature:**

```
void DBFreeStringArray(char **strArray, int n)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
`strArray`	the array of strings to be freed (some members can be `NULL`)
`n`	If `n>0`, `n` is the number of `char*` pointers in `strArray`. If `n<0`, `strArray` is treated as `NULL`-terminated. That is, entries in `strArray` are freed until the first `NULL` entry is encountered.

- **Returned value:**

  void

- **Description:**

  This function simplifies freeing of an array of strings constructed from *DBStringListToStringArray*. If `n>0`, if `strArray` contains `NULL` entries, it will handle this condition. If `n<0`, `strArray` is treated as `NULL`-terminated. That is, entries in `strArray` are freed until the first `NULL` entry is encountered.

1.8.6 DBAnnotateUcdmesh()

- **Summary:** Walk a UCD mesh guessing and adding shapetype info
- **C Signature:**

```
int DBAnnotateUcdmesh(DBucdmesh *m)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
m	A pointer to a *DBucdmesh* object

- **Returned value:**

  Returns 1 when one or more zones/shapes were annotated and 0 if not annotation was performed. Returns -1 if an error occurred.

- **Description:**

  Walks a *DBucdmesh* data structure and guesses and adds shapetype info based on `ndims` and `shapesizes` and node counts of the individual shapes. This is useful for taking an old-style *DBZonelist* object which did not include the `shapetype` member populating it based on some simple heuristics.

1.8.7 DBJoinPath()

- **Summary:** Join two strings representing paths into a single path
- **C Signature:**

```
char * DBJoinPath(char const *first, char const *second)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
`first`	The first path relative to which the second path will be joined
`second`	The second path to be combined into the first.

- **Returned value:**

  The joined path string or `NULL` if an error occured. The caller should `free()` the string.

- **Description:**

  The goal of this method is to take an existing string representing a path and a second string representing another path *relative* to the first path and then combine them into a single path.

Input	Result
DBJoinPath("foo","bar")	"foo/bar"
DBJoinPath("foo","./bar")	"foo/bar"
DBJoinPath("foo","../bar")	"bar"
DBJoinPath("foo","/bar")	"/bar"
DBJoinPath("/foo","/bar")	"/bar"
DBJoinPath("/foo","../bar")	"/bar"
DBJoinPath("foo/bar/baz","../../bin")	"foo/bin"
DBJoinPath("foo/bar/baz",".././/..//bin")	"foo/bin"

1.8.8 DBIsDifferentDouble()

- **Summary:** Compare doubles within absolute or relative tolerance
- **C Signature:**

```
int DBIsDifferentDouble(double a, double b, double abstol,
        double reltol, double reltol_eps)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
a	First double to compare
b	Second double to compare
abstol	Tolerance to be used for absolute difference. Set to a negative value to ignore this tolerance.
reltol	Tolerance to be used for relative difference. Set to a negative value to ignore this tolerance.
reltol_eps	Epsilon value to be used in alternative relative tolerance difference. Set to a negative value to ignore this.

- **Returned value:**

  0 if the difference between a and b is below tolerance. Otherwise 1. This cannot fail

- **Description:**

  Determines if a and b are the same or different based on an absolute or relative tolerances passed in. Its easies to see how the function behaves by having a look at the actual code...

```
int DBIsDifferentDouble(double a, double b, double abstol, double reltol, double␣
↪reltol_eps)
{
   double       num, den;

   /* handle possible NaNs first */
   if (isnan(a))
   {
       if (isnan(b)) return 0;
       return 1;
   }
```

(continues on next page)

(continued from previous page)

```
   else if (isnan(b))
   {
       return 1;
   }

   /*
    * First, see if we should use the alternate diff.
    * check |A-B|/(|A|+|B|+EPS) in a way that won't overflow.
    */
   if (reltol_eps >= 0 && reltol > 0)
   {
      if ((a<0 && b>0) || (b<0 && a>0)) {
         num = fabs (a/2 - b/2);
         den = fabs (a/2) + fabs(b/2) + reltol_eps/2;
         reltol /= 2;
      } else {
         num = fabs (a - b);
         den = fabs (a) + fabs(b) + reltol_eps;
      }
      if (0.0==den && num) return 1;
      if (num/den > reltol) return 1;
      return 0;
   }
   else /* use the old Abs|Rel difference test */
   {
      /*
       * Now the |A-B| but make sure it doesn't overflow which can only
       * happen if one is negative and the other is positive.
       */
      if (abstol>0) {
         if ((a<0 && b>0) || (b<0 && a>0)) {
            if (fabs (a/2 - b/2) > abstol/2) return 1;
         } else {
            if (fabs(a-b) > abstol) return 1;
         }
      }

      /*
       * Now check 2|A-B|/|A+B| in a way that won't overflow.
       */
      if (reltol>0) {
         if ((a<0 && b>0) || (b<0 && a>0)) {
            num = fabs (a/2 - b/2);
            den = fabs (a/2 + b/2);
            reltol /= 2;
         } else {
            num = fabs (a - b);
            den = fabs (a/2 + b/2);
         }
         if (0.0==den && num) return 1;
         if (num/den > reltol) return 1;
      }
```

(continues on next page)

(continued from previous page)

```
    if (abstol>0 || reltol>0) return 0;
}

/*
 * Otherwise do a normal exact comparison.
 */
return a!=b;
```

1.8.9 DBIsDifferentLongLong()

- **Summary:** Compare long longs within absolute or relative tolerance
- **C Signature:**

```
int DBIsDifferentLongLong(long long a, long long b, double abstol,
        double reltol, double reltol_eps);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
a	First long long to compare
b	Second long long to compare
abstol	Tolerance to be used for absolute difference. Set to a negative value to ignore this tolerance.
reltol	Tolerance to be used for relative difference. Set to a negative value to ignore this tolerance.
reltol_eps	Epsilon value to be used in alternative relative tolerance difference. Set to a negative value to ignore this.

- **Returned value:**

  0 if the difference between a and b is below tolerance. Otherwise 1. This cannot fail

- **Description:**

  Same as *DBIsDifferentDouble()* except for long long type.

1.8.10 DBCalcDenseArraysFromMaterial()

- **Summary:** Compute dense material volume fraction arrays from a DBmaterial object
- **C Signature:**

```
int DBCalcDenseArraysFromMaterial(DBmaterial const *mat, int datatype,
        int *narrs, void ***vfracs);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
mat	Input *DBmaterial* object
datatype	Desired data type (DB_FLOAT or DB_DOUBLE) of returned vfracs arrays.
narrs	Returned number of vfracs arrays
vfracs	Allocated and returned array of volume fractions for each material.

- **Returned value:**

  0 on successful completion. -1 if an error is encountered.

- **Description:**

  Traverse a Silo *DBmaterial* object and return a collection of *dense*, DB_ZONECENTERED volume fraction arrays, one array of volume fractions for each material. The order of the arrays in the returned vfracs is one-to-one with the order of the matnos member of the *DBmaterial* object. The order of zone-centered volume fractions in any given vfracs[i] array is one-to-one with matlist member of the *DBmaterial* obje ct.

  The representation is *dense* because there is a float (or double) for every zone and every material. Even when a zone is *clean* in a material, the representation stores a 1 for the associated vfracs entry and 0's for all other entries.

1.8.11 DBCalcMaterialFromDenseArrays()

- **Summary:** Build a DBmaterial object from dense volume fraction arrays
- **C Signature:**

```
DBmaterial *DBCalcMaterialFromDenseArrays(int narrs, int ndims, int const *dims,
               int const *matnos, int dtype, DBVCP2_t const vfracs)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg Name	Description
narrs	the number of volume fraction arrays in vfracs
ndims	the number of dimensions in the vfracs arrays
dims	the size in each dimension of the vfracs arrays
matnos	the material numbers
dtype	the datatype of the vfracs arrays (either DB_FLOAT or DB_DOUBLE)
vfracs	the dense volume fraction arrays

- **Returned value:**

  A *DBmaterial* object holding equivalent volume fraction information of the arrays or NULL if an error occurred.

- **Description:**

  Performs the reverse operation of *DBCalcDenseArraysFromMaterial*. Often, the *DBmaterial* representation is a much more efficient storage format and requires far less memory.

1.9 Option Lists or Optlists

Many Silo functions take as a last argument a pointer to an Options List or optlist. This is intended to permit the Silo API to grow and evolve as necessary without requiring substantial changes to the API itself.

In the documentation associated with each function, the list of available options and their meaning is described.

This section of the manual describes only the functions to create and manage options lists.

1.9.1 A Common Coding Gotcha

Tip: Take care regarding memory used in a *DBoptlist* object.

A number of developers have encountered problems using Silo's option lists. Many times, problems arise from the use of *local* scoped variables in a *DBoptlist* object being used outside of that local scope.

The problematic coding looks as follows…

```
AddCycleAndTime(DBoptlist *ol)
{
    int cycle = GetCycle();
    double time = GetTime();

    DBAddOption(ol, DBOPT_CYCLE, &cycle);
    DBAddOption(ol, DBOPT_DTIME, &time);

    return ol;
}
```

The problem with the above code is the *contents* of a *DBoptlist* object are interrogated only at the time the object is *used* in a Silo `DBPutXxxx()` call. This means the optlist's contents need to remain *valid* until the last `DBPutXxxx()` call in which they are used.

In the example here, the `DBAddOption()` calls are adding pointers to variables that are local to `AddCycleAndTime()`. Those memory locations are not valid outside of that function. When the optlist is used in a Silo call such as `DBPutUcdmesh()`, it will result in garbage data being written for `cycle` and `time`.

1.9.2 DBMakeOptlist()

- **Summary:** Allocate an option list.
- **C Signature:**

```
DBoptlist *DBMakeOptlist (int maxopts)
```

- **Fortran Signature:**

```
integer function dbmkoptlist(maxopts, optlist_id)
```

  returns created optlist pointer-id in optlist_id

- **Arguments:**

Arg name	Description
maxopts	Initial maximum number of options expected in the optlist. If this maximum is exceeded, the library will silently re-allocate more space using the golden-rule.

- **Returned value:**

  DBMakeOptlist returns a pointer to an option list on success and NULL on failure.

- **Description:**

  The DBMakeOptlist function allocates memory for an option list and initializes it. Use the function DBAddOption to populate the option list structure, and DBFreeOptlist to free it.

Deprecated since version 4.10: In releases of Silo prior to 4.10, if the caller accidentally added more options to an optlist than it was originally created for, an error would be generated.

New in version 4.10: However, in version 4.10, the library will silently just re-allocate the optlist to accommodate more options.

1.9.3 DBAddOption()

- **Summary:** Add an option to an option list.
- **C Signature:**

```
int DBAddOption (DBoptlist *optlist, int option, void *value)
```

- **Fortran Signature:**

```
integer function dbaddcopt (optlist_id, option, cvalue, lcvalue)
integer function dbaddcaopt (optlist_id, option, nval, cvalue, lcvalue)
integer function dbadddopt (optlist_id, option, dvalue)
integer function dbaddiopt (optlist_id, option, ivalue)
integer function dbaddropt (optlist_id, option, rvalue)

integer ivalue, optlist_id, option, lcvalue, nval
double precision dvalue
real rvalue
character*N cvalue (See [`dbset2dstrlen`](./fortran.md#dbset2dstrlen).)
```

- **Arguments:**

Arg name	Description
optlist	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.
option	Option definition. One of the predefined values described in the table in the notes section of each command which accepts an option list.
value	Pointer to the value associated with the provided option description. The data type is implied by option. The pointer must remain *valid* through the *last call* in which the optlist is used.

- **Returned value:**

  DBAddOption returns a zero on success and -1 on failure.

- **Description:**

  The DBAddOption function adds an option/value pair to an option list. Several of the output functions accept option lists to provide information of an optional nature.

Deprecated since version 4.10: In releases of Silo prior to 4.10, if the caller accidentally added more options to an optlist than it was originally created for, an error would be generated.

New in version 4.10: However, in version 4.10, the library will silently just re-allocate the optlist to accommodate more options.

1.9.4 DBClearOption()

- **Summary:** Remove an option from an option list
- **C Signature:**

```
int DBClearOption(DBoptlist *optlist, int optid)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
optlist	The option list object for which you wish to remove an option
optid	The option id of the option you would like to remove

- **Returned value:**

  DBClearOption returns zero on success and -1 on failure.

- **Description:**

  This function can be used to remove options from an option list. If the option specified by optid exists in the given option list, that option is removed from the list and the total number of options in the list is reduced by one.

  This method can be used together with DBAddOption to modify an existing option in an option list. To modify an existing option in an option list, first call DBClearOption for the option to be modified and then call DBAddOption to re-add it with a new definition.

  There is also a function to query for the value of an option in an option list, DBGetOption.

1.9.5 DBGetOption()

- **Summary:** Retrieve the value set for an option in an option list
- **C Signature:**

```
void *DBGetOption(DBoptlist *optlist, int optid)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
optlist	The optlist to query
optid	The option id to query the value for

- **Returned value:**

  Returns the pointer value set for a given option or NULL if the option is not defined in the given option list.

- **Description:**

  This function can be used to query the contents of an optlist. If the given optlist has an option of the given optid, then this function will return the pointer associated with the given optid. Otherwise, it will return NULL indicating the optlist does not contain an option with the given optid.

1.9.6 DBFreeOptlist()

- **Summary:** Free memory associated with an option list.
- **C Signature:**

```
int DBFreeOptlist (DBoptlist *optlist)
```

- **Fortran Signature:**

```
integer function dbfreeoptlist(optlist_id)
```

- **Arguments:**

Arg name	Description
optlist	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.

- **Returned value:**

  DBFreeOptlist returns a zero on success and -1 on failure.

- **Description:**

  The DBFreeOptlist function releases the memory associated with the given option list.

Tip: The individual option value pointers are not freed.

DBFreeOptlist will not fail if a NULL pointer is passed to it.

1.9.7 DBClearOptlist()

- **Summary:** Clear an optlist.
- **C Signature:**

```
int DBClearOptlist (DBoptlist *optlist)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
optlist	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.

- **Returned value:**

  DBClearOptlist returns zero on success and -1 on failure.

- **Description:**

  The DBClearOptlist function removes all options from the given option list.

1.10 User Defined (Generic) Data and Objects

If you want to create data that other applications (not written by you or someone working closely with you) can read and understand, these are **not** the right functions to use. That is because the data that these functions create is not self-describing and inherently non-shareable.

However, if you need to write data that only you (or someone working closely with you) will read such as for restart purposes, the functions described here may be helpful. The functions described here allow users to read and write arbitrary arrays of raw data as well as to create and write or read user-defined Silo objects.

1.10.1 DBWrite()

- **Summary:** Write a simple variable.
- **C Signature:**

```
int DBWrite (DBfile *dbfile, char const *varname, void const *var,
    int const *dims, int ndims, int datatype)
```

- **Fortran Signature:**

```

```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the simple variable.
var	Array defining the values associated with the variable.
dims	Array of length ndims which describes the dimensionality of the variable. Each value in the dims array indicates the number of elements contained in the variable along that dimension.
ndims	Number of dimensions.
datatype	Datatype of the variable. One of the predefined Silo data types.

- **Returned value:**

  Returns zero on success and -1 on failure.

- **Description:**

  The DBWrite function writes a simple variable into a Silo file. It is not associated with any other Silo object.

1.10.2 DBWriteSlice()

- **Summary:** Write a (hyper)slab of a simple variable
- **C Signature:**

```
int DBWriteSlice (DBfile *dbfile, char const *varname,
    void const *var, int datatype, int const *offset,
    int cost *length, int const *stride, int const *dims,
    int ndims)
```

- **Fortran Signature:**

```
integer function dbwriteslice(dbid, varname, lvarname, var,
   datatype, offset, length, stride, dims, ndims)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the simple variable.
var	Array defining the values associated with the slab.
datatype	Datatype of the variable. One of the predefined Silo data types.
offset	Array of length ndims of offsets in each dimension of the variable. This is the 0-origin position from which to begin writing the slice.
length	Array of length ndims of lengths of data in each dimension to write to the variable. All lengths must be positive.
stride	Array of length ndims of stride steps in each dimension. If no striding is desired, zeroes should be passed in this array.
dims	Array of length ndims which describes the dimensionality of the entire variable. Each value in the dims array indicates the number of elements contained in the entire variable along that dimension.
ndims	Number of dimensions.

- **Returned value:**

zero on success and -1 on failure.

- **Description:**

  The `DBWriteSlice` function writes a slab of data to a simple variable from the data provided in the `var` pointer. Any hyperslab of data may be written.

  The size of the entire variable (after all slabs have been written) must be known when the `DBWriteSlice` function is called. The data in the `var` parameter is written into the entire variable using the location specified in the offset, length, and `stride` parameters. The data that makes up the entire variable may be written with one or more calls to `DBWriteSlice`.

  The minimum `length` value is 1 and the minimum `stride` value is one.

  A one-dimensional array slice:

  Figure 0-11: Array slice

1.10.3 DBReadVar()

- **Summary:** Read a simple Silo variable.
- **C Signature:**

```
int DBReadVar (DBfile *dbfile, char const *varname, void *result)
```

- **Fortran Signature:**

```
integer function dbrdvar(dbid, varname, lvarname, ptr)
```

- **Arguments:**

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`result`	Pointer to memory into which the variable should be read. It is up to the application to provide sufficient space in which to read the variable.

- **Returned value:**

  zero on success and -1 on failure.

- **Description:**

  The `DBReadVar` function reads a simple variable into the given space.

  See *DBGetVar* for a memory-allocating version of this function.

1.10.4 DBReadVarSlice()

- **Summary:** Read a (hyper)slab of data from a simple variable.
- **C Signature:**

```
int DBReadVarSlice (DBfile *dbfile, char const *varname,
    int const *offset, int const *length, int const *stride,
    int ndims, void *result)
```

- **Fortran Signature:**

```
integer function dbrdvarslice(dbid, varname, lvarname, offset,
   length, stride, ndims, ptr)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the simple variable.
offset	Array of length ndims of offsets in each dimension of the variable. This is the 0-origin position from which to begin reading the slice.
length	Array of length ndims of lengths of data in each dimension to read from the variable. All lengths must be positive.
stride	Array of length ndims of stride steps in each dimension. If no striding is desired, zeroes should be passed in this array.
ndims	Number of dimensions in the variable.
result	Pointer to location where the slice is to be written. It is up to the application to provide sufficient space in which to read the variable.

- **Returned value:**

  zero on success and -1 on failure.

- **Description:**

  The DBReadVarSlice function reads a slab of data from a simple variable into a location provided in the result pointer. Any hyperslab of data may be read.

  Note that the minimum length value is 1 and the minimum stride value is one.

  A one-dimensional array slice:

  Figure 0-12: Array slice

1.10.5 DBGetVar()

- **Summary:** Allocate space for, and return, a simple variable.
- **C Signature:**

```
void *DBGetVar (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Name of the variable

- **Returned value:**

  A pointer to newly allocated and populated memory on success and NULL on failure.

- **Description:**

  The DBGetVar function allocates space for a simple variable, reads the variable from the Silo database, and returns a pointer to the new space. If an error occurs, NULL is returned. It is up to the application to cast the returned pointer to the correct data type.

  See *DBReadVar* for non-memory allocating versions of this function.

1.10.6 DBInqVarExists()

- **Summary:** Queries variable existence
- **C Signature:**

```
int DBInqVarExists (DBfile *dbfile, char const *name);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Object name.

- **Returned value:**

  DBInqVarExists returns non-zero if the object exists in the file. Zero otherwise.

- **Description:**

  The DBInqVarExists function is used to check for existence of an object in the given file.

  If an object was written to a file, but the file has yet to be DBClose'd, the results of this function querying that variable are undefined.

1.10.7 DBInqVarType()

- **Summary:** Return the type of the given object
- **C Signature:**

```
DBObjectType DBInqVarType (DBfile *dbfile, char const *name);
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Object name.

- **Returned value:**

Returns the *DBObjectType* corresponding to the given object.

- **Description:**

  The DBInqVarType function returns the *DBObjectType* of the given object. The value returned is described in the following table:

Object Type	Returned Value
Invalid or object not found	DB_INVALID_OBJECT
Quadmesh	DB_QUADMESH
Quadvar	DB_QUADVAR
UCD mesh	DB_UCDMESH
UCD variable	DB_UCDVAR
CSG mesh	DB_CSGMESH
CSG variable	DB_CSGVAR
Multiblock mesh	DB_MULTIMESH
Multiblock variable	DB_MULTIVAR
Multiblock material	DB_MULTIMAT
Multiblock material species	DB_MULTIMATSPECIES
Material	DB_MATERIAL
Material species	DB_MATSPECIES
Facelist	DB_FACELIST
Zonelist	DB_ZONELIST
Polyhedral-Zonelist	DB_PHZONELIST
CSG-Zonelist	DB_CSGZONELIST
Edgelist	DB_EDGELIST
Curve	DB_CURVE
Pointmesh	DB_POINTMESH
Pointvar	DB_POINTVAR
Defvars	DB_DEFVARS
Compound array	DB_ARRAY
Directory	DB_DIR
Other variable (one written out using DBWrite.)	DB_VARIABLE
User-defined	DB_USERDEF

1.10.8 DBGetVarByteLength()

- **Summary:** Return the memory byte length of a simple variable.
- **C Signature:**

```
int DBGetVarByteLength(DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Variable name.

- **Returned value:**

  The length of the given simple variable in bytes on success and -1 on failure.

- **Description:**

  The DBGetVarByteLength function returns the *memory* length of the requested simple variable, in bytes. This is useful for determining how much memory to allocate before reading a simple variable with DBReadVar. Note that this would not be a concern if one used the DBGetVar function, which allocates space itself.

1.10.9 DBGetVarByteLengthInFile()

- **Summary:** Get the *file* length of a simple variable
- **C Signature:**

```
DBGetVarByteLengthInFile(DBfile *file, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	The silo database file handle
name	Name of a simple variable in the file

- **Returned value:**

  Length of variable in file on success; -1 on failure.

- **Description:**

  Sometimes, the lengh of a variable in a file may be different from its length in memory. This is especially true if type conversion is performed on the variable when it is being read or when compression is applied. This function returns the number of bytes the variable takes up in the file.

1.10.10 DBGetVarDims()

- **Summary:** Get dimension information of a variable in a Silo file
- **C Signature:**

```
int DBGetVarDims(DBfile *file, const char const *name, int
    maxdims, int *dims)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
name	The name of the Silo object to obtain dimension information for.
maxdims	The maximum size of dims.
dims	An array of maxdims integer values to be populated with the dimension information returned by this call.

- **Returned value:**

  The number of dimensions on success; -1 on failure

- **Description:**

  This function will populate the dims array up to a maximum of maxdims values with dimension information of the specified Silo variable (object) name. The number of dimensions is returned as the function's return value.

1.10.11 DBGetVarLength()

- **Summary:** Return the number of elements in a simple variable.
- **C Signature:**

```
int DBGetVarLength (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
integer function dbinqlen(dbid, varname, lvarname, len)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Variable name.

- **Returned value:**

  The number of *elements* in the given simple variable on success and -1 on failure.

- **Description:**

  The DBGetVarLength function returns the length of the requested simple variable, in number of elements. For example a 16 byte array containing 4 floats has 4 elements.

1.10.12 DBGetVarType()

- **Summary:** Return the Silo datatype of a simple variable.
- **C Signature:**

```
int DBGetVarType (DBfile *dbfile, char const *varname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
varname	Variable name.

- **Returned value:**

  The Silo *DBdatatype* of the given simple variable on success and -1 on failure.

- **Description:**

  The DBGetVarType function returns the Silo *DBdatatype* of the requested simple variable. This works only for simple Silo variables (those written using *DBWrite* or *DBWriteSlice*). To query the type of other variables, use *DBInqVarType* instead.

1.10.13 DBPutCompoundarray()

- **Summary:** Write a Compound Array object into a Silo file.
- **C Signature:**

```
int DBPutCompoundarray (DBfile *dbfile, char const *name,
    char const * const elemnames[], int const *elemlengths,
    int nelems, void const *values, int nvalues, int datatype,
    DBoptlist const *optlist);
```

- **Fortran Signature:**

```
integer function dbputca(dbid, name, lname, elemnames,
   lelemnames, elemlengths, nelems, values, datatype, optlist_id,
   status)
```

  character*N elemnames (See *dbset2dstrlen*.)

- **Arguments:**

Arg name	Description
dbfile	Database file pointer
name	Name of the compound array structure.
elemnames	Array of length nelems containing pointers to the names of the elements.
elemlengths	Array of length nelems containing the lengths of the elements.
nelems	Number of simple array elements.
values	Array whose length is determined by nelems and elemlengths containing the values of the simple array elements.
nvalues	Total length of the values array.
datatype	Data type of the values array. One of the predefined Silo types.
optlist	Pointer to an option list structure containing additional information to be included in the compound array object written into the Silo file. Use NULL is there are no options.

- **Returned value:**

  DBPutCompoundarray returns zero on success and -1 on failure.

- **Description:**

  The DBPutCompoundarray function writes a compound array object into a Silo file. A compound array is an array whose elements are simple arrays all of which are the same *DBdatatype*.

  Often, an application will partition a block of memory into named pieces, but write the block to a database as a single entity. Fortran common blocks are used in this way. The compound array object is an abstraction of this partitioned memory block.

1.10.14 DBInqCompoundarray()

- **Summary:** Inquire Compound Array attributes.
- **C Signature:**

```
int DBInqCompoundarray (DBfile *dbfile, char const *name,
    char ***elemnames, int *elemlengths,
    int *nelems, int *nvalues, int *datatype)
```

- **Fortran Signature:**

```
integer function dbinqca(dbid, name, lname, maxwidth,
   nelems, nvalues, datatype)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
name	Name of the compound array.
elemnames	Returned array of length nelems containing pointers to the names of the array elements.
elemlengths	Returned array of length nelems containing the lengths of the array elements.
nelems	Returned number of array elements.
nvalues	Returned number of total values in the compound array.
datatype	Datatype of the data values. One of the predefined Silo data types.

- **Returned value:**

Zero on success and -1 on failure.

- **Description:**

  The DBInqCompoundarray function returns information about the compound array. It does not return the data values themselves; use *DBGetCompoundarray* instead.

1.10.15 DBGetCompoundarray()

- **Summary:** Read a compound array from a Silo database.
- **C Signature:**

```
DBcompoundarray *DBGetCompoundarray (DBfile *dbfile,
    char const *arrayname)
```

- **Fortran Signature:**

```
integer function dbgetca(dbid, name, lname, lelemnames,
   elemnames, elemlengths, nelems, values, nvalues, datatype)
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
arrayname	Name of the compound array.

- **Returned value:**

  A pointer to a *DBcompoundarray* structure on success and NULL on failure.

- **Description:**

  The DBGetCompoundarray function allocates a *DBcompoundarray* structure, reads a compound array from the Silo database, and returns a pointer to that structure. If an error occurs, NULL is returned.

1.10.16 DBMakeObject()

- **Summary:** Allocate an object of the specified length and initialize it.
- **C Signature:**

```
DBobject *DBMakeObject (char const *objname, int objtype,
    int maxcomps)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
objname	Name of the object.
objtype	Type of object. One of the predefined *DBObjectType* types.
maxcomps	Initial maximum number of components needed for this object. If this number is exceeded, the library will silently re-allocate more space using the golden rule.

- **Returned value:**

  DBMakeObject returns a pointer to the newly allocated and initialized Silo object on success and NULL on failure.

- **Description:**

  The DBMakeObject function allocates space for an object of maxcomps components.

  In releases of the Silo library prior to 4.10, if a DBobject ever had more components added to it than the maxcomps it was created with, an error would be generated and the operation to add a component would fail. However, starting in version 4.10, the maxcomps argument is used only for the initial object creation. If a caller attempts to add more than this number of components to an object, Silo will simply re-allocate the object to accomodate the additional components.

  Data producers may use this method to either modify an existing Silo object type or create an empty, new user-defined object using the type DB_USERDEF. Modified Silo objects will be recognized by Silo as long as they are not modified in ways that remove *essential* data members. Data producers may *add* data members to Silo objects and those objects will still behave as those Silo objects. However, obtaining any user-defined members of such an object may require *reading* the object via the *DBGetObject* method instead of the formal Silo method for the object (e.g. DBGetUcdmesh for a DB_UCDMESH type object).

1.10.17 DBFreeObject()

- **Summary:** Free memory associated with an object.
- **C Signature:**

```
int DBFreeObject (DBobject *object)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to the object to be freed. This object is created with the DBMakeObject function.

- **Returned value:**

  zero on success and -1 on failure.

- **Description:**

  The DBFreeObject function releases the memory associated with the given object. The data associated with the object's components is not released.

  DBFreeObject will not fail if a NULL pointer is passed to it.

1.10.18 DBChangeObject()

- **Summary:** Overwrite an existing object in a Silo file with a new object
- **C Signature:**

```
int DBChangeObject(DBfile *file, DBobject *obj)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
obj	The new DBobject object (which knows its name) to write to the file.

- **Returned value:**

  Zero on succes; -1 on failure

- **Description:**

  DBChangeObject writes a new DBobject object to a file, replacing the object in the file with the same name. Changing (e.g. overwriting) existing objects in Silo files is frought with peril. See *DBSetAllowOverwrites* for more information.

1.10.19 DBClearObject()

- **Summary:** Clear an object.
- **C Signature:**

```
int DBClearObject (DBobject *object)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to the object to be cleared. This object is created with the DBMakeObject function.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBClearObject function clears an existing object. The number of components associated with the object is set to zero.

1.10.20 DBAddDblComponent()

- **Summary:** Add a double precision floating point component to an object.
- **C Signature:**

```
int DBAddDblComponent (DBobject *object, char const *compname,
    double d)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to an object. This object is created with the DBMakeObject function.
compname	The component name.
d	The value of the double precision floating point component.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBAddDblComponent function adds a component of double precision floating point data to an existing object.

1.10.21 DBAddFltComponent()

- **Summary:** Add a floating point component to an object.
- **C Signature:**

```
int DBAddFltComponent (DBobject *object, char const *compname,
    double f)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to an object. This object is created with the DBMakeObject function.
compname	The component name.
f	The value of the floating point component.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBAddFltComponent function adds a component of floating point data to an existing object.

1.10.22 DBAddIntComponent()

- **Summary:** Add an integer component to an object.
- **C Signature:**

```
int DBAddIntComponent (DBobject *object, char const *compname,
    int i)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to an object. This object is created with the DBMakeObject function.
compname	The component name.
i	The value of the integer component.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBAddIntComponent function adds a component of integer data to an existing object.

1.10.23 DBAddStrComponent()

- **Summary:** Add a string component to an object.
- **C Signature:**

```
int DBAddStrComponent (DBobject *object, char const *compname,
    char const *s)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to the object. This object is created with the DBMakeObject function.
compname	The component name.
s	The value of the string component. Silo copies the contents of the string.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBAddStrComponent function adds a component of string data to an existing object.

1.10.24 DBAddVarComponent()

- **Summary:** Add a variable component to an object.
- **C Signature:**

```
int DBAddVarComponent (DBobject *object, char const *compname,
    char const *vardata)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
object	Pointer to the object. This object is created with the DBMakeObject function.
compname	Component name.
vardata	Name of the variable object associated with the component (see Description).

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBAddVarComponent function adds a component of the variable type to an existing object.

  The variable name in vardata is stored verbatim into the object. No translation or typing is done on the variable as it is added to the object.

1.10.25 DBWriteComponent()

- **Summary:** Add a variable component to an object and write the associated data.
- **C Signature:**

```
int DBWriteComponent (DBfile *dbfile, DBobject *object,
    char const *compname, char const *prefix,
    char const *datatype, void const *var, int nd,
    long const *count)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
object	Pointer to the object.
compname	Component name.
prefix	Path name prefix of the object.
datatype	Data type of the component's data. One of: "short", "integer", "long", "float", "double", "char".
var	Pointer to the component's data.
nd	Number of dimensions of the component.
count	An array of length nd containing the length of the component in each of its dimensions.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBWriteComponent function adds a component to an existing object and also writes the component's data to a Silo file.

1.10.26 DBWriteObject()

- **Summary:** Write an object into a Silo file.
- **C Signature:**

```
int DBWriteObject (DBfile *dbfile, DBobject const *object,
    int freemem)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
object	Object created with DBMakeObject and populated with DBAddFltComponent, DBAddIntComponent, DBAddStrComponent, and DBAddVarComponent.
freemem	If non-zero, then the object will be freed after writing.

- **Returned value:**

  Zero on success and -1 on failure.

- **Description:**

  The DBWriteObject function writes an object into a Silo file. This method may be used to write any of Silo's known, high-level *Objects*. This method is more often used to write user-defined objects. They are used when the basic Silo structures are not sufficient.

1.10.27 DBGetObject()

- **Summary:** Read an object from a Silo file as a generic object
- **C Signature:**

```
DBobject *DBGetObject(DBfile *file, char const *objname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
file	The Silo database file handle.
objname	The name of the object to get.

- **Returned value:**

  On success, a pointer to a *DBobject* struct containing the object's data. NULL on failure.

- **Description:**

  Each of the objects Silo supports has corresponding methods to both write them to a Silo database file via DBPutXxx and get them from a file via DBGetXxx.

  However, Silo objects can also be accessed as generic objects through the generic object interface. This is recommended only for objects that were written with DBWriteObject() method.

1.10.28 DBGetComponent()

- **Summary:** Allocate space for, and return, an object component.
- **C Signature:**

```
void *DBGetComponent (DBfile *dbfile, char const *objname,
    char const *compname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
objname	Object name.
compname	Component name.

- **Returned value:**

  A pointer to newly allocated space containing the component value on success, and NULL on failure.

- **Description:**

  The DBGetComponent function allocates space for one object component, reads the component, and returns a pointer to that space. If either the object or component does not exist, NULL is returned. It is up to the application to cast the returned pointer to the appropriate type.

1.10.29 DBGetComponentType()

- **Summary:** Return the type of an object component.
- **C Signature:**

```
int DBGetComponentType (DBfile *dbfile, char const *objname,
    char const *compname)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
dbfile	Database file pointer.
objname	Object name.
compname	Component name.

- **Returned value:**

  The values that are returned depend on the component's type and how the component was written into the object. The component types and their corresponding return values are listed in the table below.

Component Type	Return value
int	DB_INT
float	DB_FLOAT
double	DB_DOUBLE
char*	DB_CHAR
variable	DB_VARIABLE
everything else	DB_NOTYPE

- **Description:**

  The DBGetComponentType function reads the component's type and returns it. If either the object or component does not exist, DB_NOTYPE is returned. This function allows the application to process the component without having to know its type in advance.

1.11 JSON Interface to Silo Objects

Warning: JSON support in Silo is experimental. Silo must be configured with `--enable-json` to enable JSON support.

The interface may be dramatically re-worked, eliminated or replaced with something like Conduit. In addition, applications using Silo's JSON interface will have to us the json-c library which is also built when Silo is configured, `-I<silo-install>/json/include -L<silo-install>/json/lib -ljson`.

JSON stands for JavaScript Object Notation. The json-c library is a C implementation of JSON.

Silo's JSON interface consists of two parts. The first part is just the json-c library interface which includes methods such as `json_object_new_int()` which creates a new integer valued JSON object and `json_object_to_json_string()` which returns an ascii string representation of a JSON object as well as many other methods. This interface is documented with the json-c library and is not documented here.

The second part is some extensions to the json-c library we have defined for the purposes of providing a higher performance JSON interface for Silo objects. This includes the definition of a new JSON object type; a pointer to an *external* array. This is called an `extptr` object and is actually a specific assemblage of the following 4 JSON sub-objects…

Member name	JSON-C type	Remarks
`"datatype"`	`json_type_int`	Value is one of Silo's *DBdatatype* indicating type of data pointed to by `ptr`
`"ndims"`	`json_type_int`	Value is number of dimensions of array data pointed to by `ptr`
`"dims"`	`json_type_array`	An array of `json_type_int` values indicating size in each dimension of data pointed to by `ptr`
`"ptr"`	`json_type_string`	An ASCII hexadecimal representation of the `void*` pointer holding the array of data.

An example of the JSON representation of the `extptr` object for the data for a `DB_FLOAT`, zone-centered, variable on a 3D quad mesh of 10 x 20 x 31 zones is…

```
{
  "datatype": 19,
  "ndims": 3,
  "dims": [10, 20, 31],
  "string": "0xFFFFFFFFAC76211B"
}
```

The `extpr` object is used for all Silo data representing problem-sized array data. For example, it is used to hold coordinate data for a mesh object, or variable data for a variable object or nodelist data for a zonelist object.

Another extension of JSON we have defined for Silo is a binary format for serialized JSON objects and methods to serialize and unserialize a JSON object to a binary buffer. Although JSON implementations other than json-c also define a binary format (see for example, BSON) we have defined one here as an extension to json-c. Silo's binary format can be used, for example, by a parallel application to conveniently send Silo objects between processors by serializing to a binary buffer at the sender and then unserializing at the receiver.

Any application wishing to use the JSON Silo interface must include the `silo_json.h` header file.

In this section we describe only those methods we have defined beyond those that come with the json-c library. The functions in this part of the library are

1.11.1 `json-c` Extensions

- **Summary:** Extensions to json-c library to support Silo
- **C Signature:**

```
/* Create/delete extptr object */
json_object* json_object_new_extptr(void *p, int ndims,
int const *dims, int datatype);
void json_object_extptr_delete(json_object *jso);

/* Inspect various members of an extptr object */
int json_object_is_extptr(json_object *obj);
int json_object_get_extptr_datatype(json_object *obj);
int json_object_get_extptr_ndims(json_object *obj);
int json_object_get_extptr_dims_idx(json_object *obj, int idx);
void* json_object_get_extptr_ptr(json_object *obj);

/* binary serialization */
int json_object_to_binary_buf(json_object *obj, int flags,
void **buf, int *len);
json_object* json_object_from_binary_buf(void *buf, int len);

/* Read/Write raw binary data to a file */
int json_object_to_binary_file(char const *filename,
json_object *obj);
json_object* json_object_from_binary_file(char const *filename);

/* Fix extptr members that were ascii-fied via standard json
string serialization */
void json_object_reconstitute_extptrs(json_object *o);
```

- **Fortran Signature:**

```
None
```

- **Description:**

  As described in the introduction to this Silo API section, Silo defines a new JSON object type called an `extptr` object. It is a pointer to an external array of data. Because the json-c library Silo uses permits us to override the delete method for a JSON object, if you use the standard json-c method of deleting a JSON object, `json_object_put()`, it will have the effect of deleting any external arrays referenced by `extptr` objects.

  Note that the binary serialization defined here can be UN-serialized only by this (Silo) implementation of JSON. If you serialize to a standard JSON string using the json-c library's `json_object_to_json_string()` the resulting serialization can be correctly interpreted by *any* JSON implementation. However, in so doing, all `extptr` objects (which are unique to Silo) are converted to the standard JSON array type. All performance advantages of `extptr` objects are lost. They can, however, be re-constituted after UN-serializing a standard JSON string by the `json_object_reconstitute_extprs()` method.

1.11.2 DBWriteJsonObject()

- **Summary:** Write a JSON object to a Silo file
- **C Signature:**

```
DBWriteJsonObject(DBfile *db, json_object *jobj)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
db	Silo database file handle
jobj	JSON object pointer

- **Description:**

  This call takes a JSON object pointer and writes the object to a Silo file.

  If the object is constructed so as to match one of Silo's standard objects (any Silo object ordinarily written with a DBPutXXX() call), then the JSON object will be written to the file such that any Silo reader calling the matching DBGetXXX() method will successfully read the object. In other words, it is possible to use this method to write first-class Silo objects to a file such as a ucd-mesh or a quad-var, etc. All that is required is that the JSON object be constructed in such a way that it holds all the metadata members Silo requires/uses for that specific object. See documentation for the companion *DBGetJsonObject()*.

  Note that because there is no char const *name argument to this method, the JSON object itself must indicate the name of the object. This is done by defining a string valued member with key "silo_name".

1.11.3 DBGetJsonObject()

- **Summary:** Get an object from a Silo file as a JSON object
- **C Signature:**

```
json_object *DBGetJsonObject(DBfile *db, char const *name)
```

- **Fortran Signature:**

```
None
```

- **Arguments:**

Arg name	Description
db	Silo database file handle
name	Name of object to read

- **Description:**

  This method will read an object from a Silo file and return it as a JSON object. It can read *any* Silo object from a Silo file including objects written to the file using DBPutXXX().

  Note, however, that any problem-sized data associate with the object is returned as extptr sub-objects. See introduction to this API section for a description of extptr objects.

1.12 Previously Undocumented Use Conventions

Silo is a relatively old library. It was originally developed in the early 1990's. Over the years, a number of use conventions have emerged and taken root and are now firmly entrenched in a variety of applications using Silo.

This section of the API manual simply tries to enumerate all these conventions and their meanings. In a few cases, a long-standing use convention has been subsumed by the recent introduction of formalized Silo objects or options to implement the convention. These cases are documented and the user is encouraged to use the formal Silo approach.

Since everything documented in this section of the Silo API is a convention on the use of Silo, where one would ordinarily see a function call prototype, instead example call(s) to the Silo that implement the convention are described.

1.12.1 _visit_defvars

- **Summary:** convention for derived variable definitions
- **C Signature:**

```
int n;
char defs[1024];
sprintf(defs, "foo scalar x+y;bar vector {x,y,z};gorfo scalar sqrt(x)");
n = strlen(defs);
DBWrite(dbfile, "_visit_defvars", defs, &n, 1, DB_CHAR);
```

- **Description:**

  Do not use this convention. Instead See *DBPutDefvars*.

  `_visit_defvars` is an array of characters. The contents of this array is a semi-colon separated list of derived variable expressions of the form

```
<name of derived variable> <space> <name of type> <space> <definition>
```

  If an array of characters by this name exists in a Silo file, its contents will be used to populate the post-processor's derived variables. For VisIt, this would mean VisIt's expression system.

  This was also known as the `_meshtv_defvars` convention too.

  This named array of characters can be written at any subdirectory in the Silo file.

1.12.2 _visit_searchpath

- **Summary:** directory order to search when opening a Silo file
- **C Signature:**

```
int n;
char dirs[1024];
sprintf(dirs, "nodesets;slides;");
n = strlen(dirs);
DBWrite(dbfile, "_visit_searchpath", dirs, &n, 1, DB_CHAR);
```

- **Description:**

  When opening a Silo file, an application is free to traverse directories in whatever order it wishes. The `_visit_searchpath` convention is used by the data producer to control how downstream, post-processing tools traverse a Silo file's directory hierarchy.

_visit_searchpath is an array of characters representing a semi-colon separated list of directory names. If a character array of this name is found at any directory in a Silo file, the directories it lists (which are considered to be relative to the directory in which this array is found unless the directory names begin with a slash '/') and only those directories are searched in the order they are specified in the list.

1.12.3 _visit_domain_groups

- **Summary:** method for grouping blocks in a multi-block mesh
- **C Signature:**

```
int domToGroupMap[16];
int j;
for (j = 0; j < 16; j++)
    domToGroupMap[j] = j%4;
DBWrite(dbfile, "_visit_domain_groups", domToGroupMap, &j, 1, DB_INT);
```

- **Description:**

  Do not use this convention. Instead use Mesh Region Grouping (MRG) trees. See *DBMakeMrgtree*.

  _visit_domain_groups is an array of integers equal in size to the number of blocks in an associated multi-block mesh object specifying, for each block, a group the block is a member of. In the example code above, there are 16 blocks assigned to 4 groups.

1.12.4 AlphabetizeVariables

- **Summary:** flag to tell post-processor to alphabetize variable lists
- **C Signature:**

```
int doAlpha = 1;
int n = 1;
DBWrite(dbfile, "AlphabetizeVariables", &doAlpha, &n, 1, DB_INT);
```

- **Description:**

  The AlphabetizeVariables convention is a simple integer value which, if non-zero, indicates that the post-processor should alphabetize its variable lists. In VisIt, this would mean that various menus in the GUI, for example, are constructed such that variable names placed near the top of the menus come alphabetically before variable names near the bottom of the menus. Otherwise, variable names are presented in the order they are encountered in the database which is often the order they were written to the database by the data producer.

1.12.5 ConnectivityIsTimeVarying

- **Summary:** flag telling post-processor if connectivity of meshes in the Silo file is time varying or not
- **C Signature:**

```
int isTimeVarying = 1;
int n = 1;
DBWrite(dbfile, "ConnectivityIsTimeVarying", &isTimeVarying, &n, 1, DB_INT);
```

- **Description:**

  The `ConnectivityIsTimeVarying` convention is a simple integer flag which, if non-zero, indicates to post-processing tools that the connectivity for the mesh(s) in the database varies with time. This has important performance implications and should only be specified if indeed it is necessary as, for instance, in post-processors that assume connectivity is **not** time varying. This is an assumption made by VisIt and the `ConnectivityIsTimeVarying` convention is a way to tell VisIt to **not** make this assumption.

1.12.6 MultivarToMultimeshMap_vars

- **Summary:** list of multivars to be associated with multimeshes
- **C Signature:**

```
int len;
char tmpStr[256];
sprintf(tmpStr, "d;p;u;v;w;hist;mat1");
len = strlen(tmpStr);
DBWrite(dbfile, "MultivarToMultimeshMap_vars", tmpStr, &len, 1, DB_CHAR);
```

- **Description:**

  Do not use this convention. Instead use the `DBOPT_MMESH_NAME` optlist option for a *DBPutMultivar()* call to associate a multimesh with a multivar.

  The `MultivarToMultimeshMap_vars` use convention goes hand-in-hand with the *MultivarToMultimeshMap_meshes* use convention. The `_vars` portion is an array of characters defining a semi-colon separated list of multivar object names to be associated with multi-mesh names. The `_mesh` portion is an array of characters defining a semi-colon separated list of associated multimesh object names. This convention was introduced to deal with a shortcoming in Silo where multivar objects did not know the multimesh object they were associated with. This has since been corrected by the `DBOPT_MMESH_NAME` optlist option for a *DBPutMultivar()* call.

1.12.7 MultivarToMultimeshMap_meshes

- **Summary:** list of multimeshes to be associated with multivars
- **C Signature:**

```
int len;
char tmpStr[256];
sprintf(tmpStr, "mesh1;mesh1;mesh1;mesh1;mesh1;mesh1;mesh1");
len = strlen(tmpStr);
DBWrite(dbfile, "MultivarToMultimeshMap_meshes", tmpStr, &len, 1, DB_CHAR);
```

- **Description:**

  See *MultivarToMultimeshMap_vars* on page .

1.13 Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within a single application interacting with a Silo file. Note that when Silo was originally written, the vision was that only visualization/post-processing tools would ever attempt to read the contents of Silo files. Therefore, the Fortran interface has never included all the companion functions to read objects. That said, it is possible to write simple fortran callable wrappers to the C functions much like the write interface already implemented. Have a look in the source file silo_f.c for examples.

1.13.1 dbmkptr()

- **Summary:** create a pointer-id from a pointer
- **C Signature:**

```
integer function dbmkptr(void p)
```

- **Arguments:**

Arg name	Description
p	pointer for which a pointer-id is needed

- **Returned value:**

  the integer pointer id to associate with the pointer

- **Description:**

  In cases where the C interface returns to the application a pointer to an abstract Silo object, in the Fortran interface an integer pointer-id is created and returned instead. In addition, in cases where the C interface would accept an array of pointers, such as in *DBPutCsgvar()*, the Fortran interface accepts an array of pointer-ids. This function is used to create a pointer-id from a pointer.

  A table of pointers is maintained internally in the Fortran wrapper library. The pointer-id is simply the index into this table where the associated object's pointer actually is. The caller can free up space in this table using *dbrmptr()*

1.13.2 dbrmptr()

- **Summary:** remove an old and no longer needed pointer-id
- **Fortran Signature:**

```
integer function dbrmptr(ptr_id)
```

- **Arguments:**

Arg name	Description
ptr_id	the pointer-id to remove

- **Returned value:**

  always 0

1.13.3 dbset2dstrlen()

- **Summary:** Set the size of a 'row' for pointers to 'arrays' of strings
- **Fortran Signature:**

```
integer function dbset2dstrlen(int len)
integer len
```

- **Arguments:**

Arg name	Description
len	The length to set

- **Returned value:**

  Returns the previously set value.

- **Description:**

  A number of functions in the Fortran interface take a char* argument that is really treated internally in the Fortran interface as a 2D array of characters. Calling this function allows the caller to specify the length of the rows in this 2D array of characters. If necessary, this setting can be varied from call to call.

  The default value is 32 characters.

1.13.4 dbget2dstrlen()

- **Summary:** Get the size of a 'row' for pointers to 'arrays' of character strings
- **Fortran Signature:**

```
integer function dbget2dstrlen()
```

- **Arguments:**

  None

- **Returned value:**

  The current setting for the 2D string length.

1.13.5 DBFortranAllocPointer()

- **Summary:** Facilitates accessing C objects through Fortran
- **Fortran Signature:**

```
int DBFortranAllocPointer (void *pointer)
```

- **Arguments:**

Arg name	Description
pointer	A pointer to a Silo object for which a Fortran identifier is needed

- **Returned value:**

  DBFortranAllocPointer returns an integer that Fortran code can use to reference the given Silo object.

- **Description:**

  The `DBFortranAllocPointer` function allows programs written in both C and Fortran to access the same data structures. Many of the routines in the Fortran interface to Silo use an "object id", an integer which refers to a Silo object. DBFortanAllocPointer converts a `pointer` to a Silo object into an integer that Fortran code can use. In some ways, this function is the inverse of `DBFortranAccessPointer`.

  The integer that `DBFortranAllocPointer` returns is used to index a table of Silo object pointers. When done with the integer, the entry in the table may be freed for use later through the use of `DBFortranRemovePointer`.

  See *DBFortranAccessPointer* and *DBFortranRemovePointer* for more information about how to use Silo objects in code that uses C and Fortran together.

  For example, if you have a `DBfile*` pointer for a Silo database file and wish to pass this object to some Fortran function(s), the coding pattern would look like the following…

```
DBfile *db = DBOpen("foo.silo", DB_UNKNOWN, DB_APPEND);
/*
.
. C/C++ code operates on db `pointer`
.
*/

/* create entry in Fortran wrappers for this Silo Object */
int dbid = DBFortranAllocPointer(db);

/* pass dbid to any Fortran code to interact with file */

/* free up Fortran wrapper resources for this Silo object */

DBFortranRemovePointer(dbid);
```

1.13.6 DBFortranAccessPointer()

- **Summary:** Access Silo objects created through the Fortran Silo interface.
- **C Signature:**

```
void *DBFortranAccessPointer (int value)
```

- **Arguments:**

Arg name	Description
`value`	The `value` returned by a Silo Fortran function, referencing a Silo object.

- **Returned value:**

  DBFortranAccessPointer returns a pointer to a Silo object (which must be cast to the appropriate type) on success, and `NULL` on failure.

- **Description:**

  The `DBFortranAccessPointer` function allows programs written in both C and Fortran to access the same data structures. Many of the routines in the Fortran interface to Silo return an "object id", an integer which refers to

a Silo object. DBFortranAccessPointer converts this integer into a C pointer so that the sections of code written in C can access the Silo object directly.

See *DBFortranAllocPointer* and *DBFortranRemovePointer* for more information about how to use Silo objects in code that uses C and Fortran together.

1.13.7 DBFortranRemovePointer()

- **Summary:** Removes a pointer from the Fortran-C index table
- **C Signature:**

```
void DBFortranRemovePointer (int value)
```

- **Arguments:**

Arg name	Description
value	An integer returned by DBFortranAllocPointer

- **Returned value:**

  Nothing

- **Description:**

  The DBFortranRemovePointer function frees up the storage associated with Silo object pointers as allocated by DBFortranAllocPointer.

  Code that uses both C and Fortran may make use of DBFortranAllocPointer to allocate space in a translation table so that the same Silo object may be referenced by both languages. DBFortranAccessPointer provides access to this Silo object from the C side. Once the Fortran side of the code is done referencing the object, the space in the translation table may be freed by calling DBFortranRemovePointer. See *DBFortranAccessPointer* and *DBFortranAllocPointer* for more information about how to use Silo objects in code that uses C and Fortran together.

1.13.8 dbwrtfl()

- **Summary:** Write a facelist object referenced by its object_id to a silo file
- **C Signature:**

```
dbwrtfl(dbid, name, lname, object_id, status)
```

- **Arguments:**

Arg name	Description
dbid	The identifier for the Silo database to write the object to.
name	The name to be assigned to the object in the file.
lname	The length of the name argument.
object_id	The identifier for the facelist object, obtained via dbcalcfl.
status	Return value indicating success or failure of the operation; 0 on success, -1 on failure.

- **Returned value:**

  Nothing

- **Description:**

  This function is designed to go hand-in-hand with `dbcalcfl`, the function used to calculate an external facelist. When `dbcalcfl` is called, an object identifier is returned in `object_id` for the newly created facelist. This call can then be used to write that facelist object to a Silo database.

1.14 Python Interface

Tip: Silo's python interface is designed to compile and work with either Python 2 or Python 3

Silo's configure logic looks for `python`, not `python3`. If you need it to look instead for `python3`, this works...

```
env PYTHON=python3 PYTHON_CPPFLAGS=-I<path-to-python-headers> ./configure --enable-
↪pythonmodule
```

where `path-to-python-headers` can often be obtained by...

```
% python3
Python 3.8.9 (default, May 17 2022, 12:55:41)
[Clang 13.1.6 (clang-1316.0.21.2.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import sysconfig
>>> sysconfig.get_config_var("INCLUDEPY")
'/Applications/Xcode.app/Contents/Developer/Library/Frameworks/Python3.framework/
↪Versions/3.8/Headers'
```

Warning: Configuring with `--disable-shared` or `--enable-shared=no` will disable the python module

It is probably easiest to understand the Python interface to Silo by examining some examples and tests. In the source code distribution, you can find some examples in tools/python and tests in tests directories. Here, we briefly describe Silo's Python interface.

The Python interface will be in the `lib` dir of the Silo installation, named `Silo.so`. To use it, Python needs to be told where to find it. You can do this a couple of ways; through the `PYTHONPATH` environment variable or by explicitly adding the Silo installation `lib` dir to Python's path using `sys.path.append()`.

For example, if Silo is installed to /foo/bar, this works...

```
% env PYTHONPATH=/foo/bar/lib python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple `LLVM` 7.0.0] on darwin
Type "help", "copyright", "credits" or "license" for more info.
>>> import Silo
```

Or, if you prefer to use `sys.path.append`...

```
% python
Python 2.7.10 (default, Oct 23 2015, 19:19:21)
[GCC 4.2.1 Compatible Apple `LLVM` 7.0.0] on darwin
Type "help", "copyright", "credits" or "license" for more info.
```

(continues on next page)

(continued from previous page)

```
>>> import sys
>>> sys.path.append("/foo/bar/lib")
>>> import Silo
```

1.14.1 Silo.Open()

- **Summary:** Open a Silo file (See *DBOpen*)
- **C Signature:**

```
DBfile Silo.Open(filename, flags);
```

- **Arguments:**

Arg name	Description
`filename`	Name of the Silo file to open
`flags`	Pass either `Silo.DB_READ` if you will only read objects from the file or `Silo.DB_APPEND` if you need to also write data to the file.

- **Description:**

  Returns a `DBfile` object as a Python object

1.14.2 Silo.Create()

- **Summary:** Create a new silo file (See *DBCreate*)
- **C Signature:**

```
DBfile Silo.Create(filename, info, driver, clobber)
```

- **Arguments:**

Arg name	Description
`filename`	[required string] name of the file to create
`info`	[required string] comment to be stored in the file
`driver`	[optional int] which `driver` to use. Pass either `Silo.DB_PDB` or `Silo.DB_HDF5`. Note that advanced `driver` features are not available through the Python interface. Default is `Silo.DB_PDB`.
`clobber`	[optional int] indicate whether any existing file should be clobbered. Pass either `Silo.DB_CLOBBER` or `Silo.DB_NOCLOBBER`. Default is `Silo.DB_CLOBBER`.

- **Description:**

  Returns a `DBfile` object as a Python object

1.14.3 <DBfile>.GetToc()

- **Summary:** Get the table of contents
- **C Signature:**

```
DBtoc <DBfile>.GetToc()
```

- **Description:**

  Returns a `DBToc` object as a Python object. This probably should really be a Python dictionary object but it is not presently. There are no methods defined for a `DBToc` object but if you print it, you can get the list of objects in the current working directory in the file.

1.14.4 <DBfile>.GetVarInfo()

- **Summary:** Get metadata and bulk data of any object (See *DBGetObject*)
- **C Signature:**

```
dict <DBfile>.GetVarInfo(name, flag)
```

- **Arguments:**

Arg name	Description
`name`	[required string] `name` of object to read
`flag`	[optional int] `flag` to indicate if object bulk/raw data should be included. Pass 0 to **not** also read object bulk/raw data. Pass non-zero to also read object bulk/raw data. Default is 0.

- **Description:**

  Returns a Python dictionary object for a Silo high level object (e.g. not a primitive array). This method cannot be used to read the contents of a primitive array. It can be used for any object the Silo C interface's `DBGetObject()` would also be used. If object bulk data is not also read, then the dictionary members for those sub-objects will contain a string holding the path of either a sub-object or a primitive array. Note that on the HDF5 driver, if friendly HDF5 names were not used to create the file, then the string paths for these sub-objects are often cryptic references to primitive arrays in the hidden `/.silo` directory of the Silo file.

  This method is poorly named. A better name is probably `GetObject`.

1.14.5 <DBfile>.GetVar()

- **Summary:** Get a primitive array (See *DBReadVar*)
- **C Signature:**

```
tuple <DBfile>.GetVar(name)
```

- **Arguments:**

Arg name	Description
`name`	[required string] `name` of primitive array to read

- **Description:**

  This method returns a primitive array as a Python tuple

1.14.6 <DBfile>.SetDir()

- **Summary:** Set current working directory of the Silo file (See *DBSetDir*)
- **C Signature:**

```
NoneType <DBfile>.SetDir(name)
```

- **Arguments:**

Arg name	Description
name	[required string] name of directory to set

- **Description:**

  Sets the current working directory of the Silo file

1.14.7 <DBfile>.Close()

- **Summary:** Close the Silo file
- **C Signature:**

```
NoneType <DBfile>.Close()
```

- **Description:**

  Close the Silo file

1.14.8 <DBfile>.WriteObject()

- **Summary:** Write a Python dictionary as a Silo object (See *DBWriteObject*)
- **C Signature:**

```
NoneType <DBfile>.WriteObject(name, obj_dict)
```

- **Arguments:**

Arg name	Description
name	[required string] name of the new object to write
obj_dict	[required dict] Python dictionary containing object data

- **Description:**

  This method will write any Python dictionary object to a Silo file as a Silo object. Here's the rub. Readers employing Silo's high level interface (e.g. DBGetUcdmesh, DBGetQuadvar, etc.) will be able recognize an object so written if and only if the dict object's structure *matches* a known high-level Silo object.

  So, you can use this method to write objects that can be read later via Silo's high-level object methods such DBGetUcdmesh and DBGetMaterial, etc. as long as the Python dictionary's members match what Silo expects.

Often, the easiest way to interrogate how a given Python dict object should be structured to match a Silo object is to find an example object in some file and read it into Python with `GetVarInfo()`.

It is fine to create a dict object with additional members too. For example, if you create a dict object that is intended to be a Silo material object, you can add additional members to it and readers will still be able to read it via `DBGetMaterial`. Of course, such readers will not be aware of any additional members so handled.

It is also fine to create wholly new kinds of Silo objects for which there are no corresponding high-level interface methods such as `GetUcdmesh` or `GetQuadvar` in the C language interface. Such an object can be read by the generic object, `DBGetObject()` C language interface method.

1.14.9 <DBfile>.Write()

- **Summary:** Write primitive array data to a Silo file (see DBWrite)
- **C Signature:**

```
NoneType <DBfile>.Write(name, data)
```

- **Arguments:**

Arg name	Description
`name`	[required string] `name` of the primitive array
`data`	[required tuple] the `data` to write

- **Description:**

This method will write a primitve array to a Silo file. However, it presently handles only one dimensional tuples. Furthermore, the tuples must be consistent in type (e.g. all floats or all ints).

1.14.10 <DBfile>.MkDir()

- **Summary:** Make a directory in a Silo file
- **C Signature:**

```
NoneType <DBfile>.MkDir(name)
```

- **Arguments:**

Arg name	Description
`name`	[required string] `name` of the directory to create

- **Description:**

Creates a new directory in a Silo file

1.15 Deprecated Functions

The following functions have been deprecated from Silo, some as early as version 4.6. Attempts to call these methods in later versions may still succeed. However, deprecation warnings will be generated on stderr (See *DBSetDeprecateWarnings*.). There is no guarantee that these methods will exist in later versions of Silo.

- DBGetComponentNames
- DBGetAtt (completely removed in version 4.10)
- DBListDir (completely removed in version 4.10)
- DBReadAtt (completely removed in version 4.10)
- DBGetQuadvar1 (completely removed in version 4.10)
- DBcontinue (completely removed in version 4.10)
- DBPause (completely removed in version 4.10)
- DBPutZonelist (use [DBPutZonelist2}(objects.md#dbputzonelist2) instead)
- DBPutUcdsubmesh (use *Mesh Region Grouping* trees instead)
- DBErrFunc (use *DBErrfunc* instead)
- DBSetDataReadMask (use *DBSetDataReadMask2* instead)
- DBGetDataReadMask (use *DBGetDataReadMask2* instead)

1.16 Silo Library Header File

We include the contents of the Silo header file here including a description of all DBxxx object structs that are returned in DBGetXXX() calls as well as all other constant and symbols defined by the library.

1.16.1 DBtoc

```
typedef struct DBtoc_ {

    char         **curve_names;
    int            ncurve;

    char         **multimesh_names;
    int            nmultimesh;

    char         **multimeshadj_names;
    int            nmultimeshadj;

    char         **multivar_names;
    int            nmultivar;

    char         **multimat_names;
    int            nmultimat;
```

(continues on next page)

(continued from previous page)

```
    char         **multimatspecies_names;
    int            nmultimatspecies;

    char         **csgmesh_names;
    int            ncsgmesh;

    char         **csgvar_names;
    int            ncsgvar;

    char         **defvars_names;
    int            ndefvars;

    char         **qmesh_names;
    int            nqmesh;

    char         **qvar_names;
    int            nqvar;

    char         **ucdmesh_names;
    int            nucdmesh;

    char         **ucdvar_names;
    int            nucdvar;

    char         **ptmesh_names;
    int            nptmesh;

    char         **ptvar_names;
    int            nptvar;

    char         **mat_names;
    int            nmat;

    char         **matspecies_names;
    int            nmatspecies;

    char         **var_names;
    int            nvar;

    char         **obj_names;
    int            nobj;

    char         **dir_names;
    int            ndir;

    char         **array_names;
    int            narray;

    char         **mrgtree_names;
    int            nmrgtree;

    char         **groupelmap_names;
```

(continues on next page)

(continued from previous page)

```
    int             ngroupelmap;

    char          **mrgvar_names;
    int             nmrgvar;

    char          **symlink_target_names;
    int             nsymlink;
    char          **symlink_names; /* copies of other members; never free'd */

} DBtoc;
```

1.16.2 DBcurve

```
typedef struct DBcurve_ {
/*----------- X vs. Y (Curve) Data -----------*/
    int             id;          /* Identifier for this object */
    int             datatype;    /* Datatype for x and y (float, double) */
    int             origin;      /* '0' or '1' */
    char           *title;       /* Title for curve */
    char           *xvarname;    /* Name of domain (x) variable */
    char           *yvarname;    /* Name of range  (y) variable */
    char           *xlabel;      /* Label for x-axis */
    char           *ylabel;      /* Label for y-axis */
    char           *xunits;      /* Units for domain */
    char           *yunits;      /* Units for range  */
    void           *x;           /* Domain values for curve */
    void           *y;           /* Range  values for curve */
    int             npts;        /* Number of points in curve */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char           *reference;   /* Label to reference object */
    int             coord_sys;   /* To indicate other coordinate systems */
    double          missing_value; /* Value to indicate var data is invalid/missing */
} DBcurve;
```

1.16.3 DBdefvars

```
typedef struct DBdefvars_ {
    int             ndefs;       /* number of definitions */
    char          **names;       /* [ndefs] derived variable names */
    int            *types;       /* [ndefs] derived variable types */
    char          **defns;       /* [ndefs] derived variable definitions */
    int           *guihides;     /* [ndefs] flags to hide from
                                    post-processor's GUI */
} DBdefvars;
```

1.16.4 DBpointmesh

```
typedef struct DBpointmesh_ {
/*------------ Point Mesh ------------*/
    int             id;          /* Identifier for this object */
    int             block_no;    /* Block number for this mesh */
    int             group_no;    /* Block group number for this mesh */
    char           *name;        /* Name associated with this mesh */
    int             cycle;       /* Problem cycle number */
    char           *units[3];    /* Units for each axis */
    char           *labels[3];   /* Labels for each axis */
    char           *title;       /* Title for curve */

    void           *coords[3];   /* Coordinate values */
    float           time;        /* Problem time */
    double          dtime;       /* Problem time, double data type */
   /*
    * The following two fields really only contain 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetPointmesh() which
    * can cause three doubles to be stored there instead of three floats.
    */
    float           min_extents[6];  /* Min mesh extents [ndims] */
    float           max_extents[6];  /* Max mesh extents [ndims] */

    int             datatype;    /* Datatype for coords (float, double) */
    int             ndims;       /* Number of computational dimensions */
    int             nels;        /* Number of elements in mesh */
    int             origin;      /* '0' or '1' */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    void           *gnodeno;     /* global node ids */
    char           *mrgtree_name; /* optional name of assoc. mrgtree object */
    int             gnznodtype;  /* datatype for global node/zone ids */
    char           *ghost_node_labels;
    char          **alt_nodenum_vars;
} DBpointmesh;
```

1.16.5 DBmultimesh

```
typedef struct DBmultimesh_ {
/*------------ Multi-Block Mesh ------------*/
    int             id;          /* Identifier for this object */
    int             nblocks;     /* Number of blocks in mesh */
    int             ngroups;     /* Number of block groups in mesh */
    int            *meshids;     /* Array of mesh-ids which comprise mesh */
    char          **meshnames;   /* Array of mesh-names for meshids */
    int            *meshtypes;   /* Array of mesh-type indicators [nblocks] */
    int            *dirids;      /* Array of directory ID's which contain blk */
    int             blockorigin; /* Origin (0 or 1) of block numbers */
    int             grouporigin; /* Origin (0 or 1) of group numbers */
    int             extentssize; /* size of each extent tuple */
    double         *extents;     /* min/max extents of coords of each block */
```

(continues on next page)

(continued from previous page)

```
    int           *zonecounts;  /* array of zone counts for each block */
    int           *has_external_zones;  /* external flags for each block */
    int            guihide;     /* Flag to hide from post-processor's GUI */
    int            lgroupings;  /* size of groupings array */
    int            *groupings;  /* Array of mesh-ids, group-ids, and counts */
    char          **groupnames; /* Array of group-names for groupings  */
    char          *mrgtree_name;/* optional name of assoc. mrgtree object */
    int            tv_connectivity;
    int            disjoint_mode;
    int            topo_dim;    /* Topological dimension; max of all blocks. */
    char          *file_ns;     /* namescheme for files (in lieu of meshnames) */
    char          *block_ns;    /* namescheme for block objects (in lieu of meshnames) */
    int            block_type;  /* constant block type for all blocks (in lieu of␣
↪meshtypes) */
    int           *empty_list;  /* list of empty block #'s (option for namescheme) */
    int            empty_cnt;   /* size of empty list */
    int            repr_block_idx; /* index of a 'representative' block */
    char         **alt_nodenum_vars;
    char         **alt_zonenum_vars;
    char          *meshnames_alloc; /* original alloc of meshnames as string list */
} DBmultimesh;
```

1.16.6 DBmultimeshadj

```
typedef struct DBmultimeshadj_ {
/*----------- Multi-Block Mesh Adjacency -----------*/
    int            nblocks;     /* Number of blocks in mesh */
    int            blockorigin; /* Origin (0 or 1) of block numbers */
    int           *meshtypes;   /* Array of mesh-type indicators [nblocks] */
    int           *nneighbors;  /* Array [nblocks] neighbor counts */

    int           lneighbors;
    int           *neighbors;   /* Array [lneighbors] neighbor block numbers */
    int           *back;        /* Array [lneighbors] neighbor block back */

    int            totlnodelists;
    int           *lnodelists;  /* Array [lneighbors] of node counts shared */
    int          **nodelists;   /* Array [lneighbors] nodelists shared */

    int            totlzonelists;
    int           *lzonelists;  /* Array [lneighbors] of zone counts adjacent */
    int          **zonelists;   /* Array [lneighbors] zonelists adjacent */
} DBmultimeshadj;
```

1.16.7 DBmultivar

```
typedef struct DBmultivar_ {
/*------------ Multi-Block Variable ------------*/
    int            id;          /* Identifier for this object  */
    int            nvars;       /* Number of variables   */
    int            ngroups;     /* Number of block groups in mesh */
    char         **varnames;    /* Variable names   */
    int           *vartypes;    /* variable types   */
    int            blockorigin; /* Origin (0 or 1) of block numbers */
    int            grouporigin; /* Origin (0 or 1) of group numbers */
    int            extentssize; /* size of each extent tuple */
    double        *extents;     /* min/max extents of each block */
    int            guihide;     /* Flag to hide from post-processor's GUI */
    char         **region_pnames;
    char          *mmesh_name;
    int            tensor_rank;    /* DB_VARTYPE_XXX */
    int            conserved;   /* indicates if the variable should be conserved
                                   under various operations such as interp. */
    int            extensive;   /* indicates if the variable reprsents an extensiv
                                   physical property (as opposed to intensive) */
    char          *file_ns;     /* namescheme for files (in lieu of meshnames) */
    char          *block_ns;    /* namescheme for block objects (in lieu of meshnames) */
    int            block_type;  /* constant block type for all blocks (in lieu of␣
↪meshtypes) */
    int           *empty_list;  /* list of empty block #'s (option for namescheme) */
    int            empty_cnt;   /* size of empty list */
    int            repr_block_idx; /* index of a 'representative' block */
    double         missing_value; /* Value to indicate var data is invalid/missing */
    char          *varnames_alloc; /* original alloc of varnames as string list */
} DBmultivar;
```

1.16.8 DBmultimat

```
typedef struct DBmultimat_ {
    int            id;          /* Identifier for this object  */
    int            nmats;       /* Number of materials   */
    int            ngroups;     /* Number of block groups in mesh */
    char         **matnames;    /* names of constiuent DBmaterial objects */
    int            blockorigin; /* Origin (0 or 1) of block numbers */
    int            grouporigin; /* Origin (0 or 1) of group numbers */
    int           *mixlens;     /* array of mixlen values in each mat */
    int           *matcounts;   /* counts of unique materials in each block */
    int           *matlists;    /* list of materials in each block */
    int            guihide;     /* Flag to hide from post-processor's GUI */
    int            nmatnos;     /* global number of materials over all pieces */
    int           *matnos;      /* global list of material numbers */
    char         **matcolors;   /* optional colors for materials */
    char         **material_names; /* optional names of the materials */
    int            allowmat0;   /* Flag to allow material "0" */
    char          *mmesh_name;
```

(continues on next page)

(continued from previous page)

```
    char           *file_ns;     /* namescheme for files (in lieu of meshnames) */
    char           *block_ns;    /* namescheme for block objects (in lieu of meshnames) */
    int            *empty_list;  /* list of empty block #'s (option for namescheme) */
    int             empty_cnt;   /* size of empty list */
    int             repr_block_idx; /* index of a 'representative' block */
    char           *matnames_alloc; /* original alloc of matnames as string list */
} DBmultimat;
```

1.16.9 DBmultimatspecies

```
typedef struct DBmultimatspecies_ {
    int             id;          /* Identifier for this object  */
    int             nspec;       /* Number of species   */
    int             ngroups;     /* Number of block groups in mesh */
    char          **specnames;   /* Species object names   */
    int             blockorigin; /* Origin (0 or 1) of block numbers */
    int             grouporigin; /* Origin (0 or 1) of group numbers */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    int             nmat;        /* equiv. to nmatnos of a DBmultimat */
    int            *nmatspec;    /* equiv. to matnos of a DBmultimat */
    char          **species_names; /* optional names of the species */
    char          **speccolors;  /* optional colors for species */
    char           *file_ns;     /* namescheme for files (in lieu of meshnames) */
    char           *block_ns;    /* namescheme for block objects (in lieu of meshnames) */
    int            *empty_list;  /* list of empty block #'s (option for namescheme) */
    int             empty_cnt;   /* size of empty list */
    int             repr_block_idx; /* index of a 'representative' block */
    char           *specnames_alloc; /* original alloc of matnames as string list */
} DBmultimatspecies;
```

1.16.10 DBzonelist

```
typedef struct DBzonelist_ {
    int             ndims;       /* Number of dimensions (2,3) */
    int             nzones;      /* Number of zones in list */
    int             nshapes;     /* Number of zone shapes */
    int            *shapecnt;    /* [nshapes] occurences of each shape */
    int            *shapesize;   /* [nshapes] Number of nodes per shape */
    int            *shapetype;   /* [nshapes] Type of shape */
    int            *nodelist;    /* Sequent lst of nodes which comprise zones */
    int             lnodelist;   /* Number of nodes in nodelist */
    int             origin;      /* '0' or '1' */
    int             min_index;   /* Index of first real zone */
    int             max_index;   /* Index of last real zone */

/*--------- Optional zone attributes ---------*/
    int            *zoneno;      /* [nzones] zone number of each zone */
    void           *gzoneno;     /* [nzones] global zone number of each zone */
    int             gnznodtype;  /* datatype for global node/zone ids */
```

(continues on next page)

(continued from previous page)

```
    char          *ghost_zone_labels;
    char         **alt_zonenum_vars;
} DBzonelist;
```

1.16.11 DBphzonelist

```
typedef struct DBphzonelist_ {
    int            nfaces;      /* Number of faces in facelist (aka "facetable") */
    int           *nodecnt;     /* Count of nodes in each face */
    int            lnodelist;   /* Length of nodelist used to construct faces */
    int           *nodelist;    /* List of nodes used in all faces */
    char          *extface;     /* boolean flag indicating if a face is external */
    int            nzones;      /* Number of zones in this zonelist */
    int           *facecnt;     /* Count of faces in each zone */
    int            lfacelist;   /* Length of facelist used to construct zones */
    int           *facelist;    /* List of faces used in all zones */
    int            origin;      /* '0' or '1' */
    int            lo_offset;   /* Index of first non-ghost zone */
    int            hi_offset;   /* Index of last non-ghost zone */

/*--------- Optional zone attributes ---------*/
    int           *zoneno;      /* [nzones] zone number of each zone */
    void          *gzoneno;     /* [nzones] global zone number of each zone */
    int            gnznodtype;  /* datatype for global node/zone ids */
    char          *ghost_zone_labels;
    char         **alt_zonenum_vars;
} DBphzonelist;
```

1.16.12 DBfacelist

```
typedef struct DBfacelist_ {
/*----------- Required components ------------*/
    int            ndims;       /* Number of dimensions (2,3) */
    int            nfaces;      /* Number of faces in list */
    int            origin;      /* '0' or '1' */
    int           *nodelist;    /* Sequent list of nodes comprise faces */
    int            lnodelist;   /* Number of nodes in nodelist */

/*----------- 3D components -------------------*/
    int            nshapes;     /* Number of face shapes */
    int           *shapecnt;    /* [nshapes] Num of occurences of each shape */
    int           *shapesize;   /* [nshapes] Number of nodes per shape */

/*----------- Optional type component----------*/
    int            ntypes;      /* Number of face types */
    int           *typelist;    /* [ntypes] Type ID for each type */
    int           *types;       /* [nfaces] Type info for each face */

/*--------- Optional node attributes ----------*/
```

(continues on next page)

(continued from previous page)

```
    int           *nodeno;      /* [lnodelist] node number of each node */

/*----------- Optional zone-reference component---------*/
    int           *zoneno;      /* [nfaces] Zone number for each face */
} DBfacelist;
```

1.16.13 DBedgelist

```
typedef struct DBedgelist_ {
    int            ndims;       /* Number of dimensions (2,3) */
    int            nedges;      /* Number of edges */
    int           *edge_beg;    /* [nedges] */
    int           *edge_end;    /* [nedges] */
    int            origin;      /* '0' or '1' */
} DBedgelist;
```

1.16.14 DBquadmesh

```
typedef struct DBquadmesh_ {
/*----------- Quad Mesh -----------*/
    int            id;          /* Identifier for this object */
    int            block_no;    /* Block number for this mesh */
    int            group_no;    /* Block group number for this mesh */
    char          *name;        /* Name associated with mesh */
    int            cycle;       /* Problem cycle number */
    int            coord_sys;   /* Cartesian, cylindrical, spherical */
    int            major_order; /* 1 indicates row-major for multi-d arrays */
    int            stride[3];   /* Offsets to adjacent elements  */
    int            coordtype;   /* Coord array type: collinear,
                                 * non-collinear */
    int            facetype;    /* Zone face type: rect, curv */
    int            planar;      /* Sentinel: zones represent area or volume? */

    void          *coords[3];   /* Mesh node coordinate ptrs [ndims] */
    int            datatype;    /* Type of coordinate arrays (double,float) */
    float          time;        /* Problem time */
    double         dtime;       /* Problem time, double data type */
   /*
    * The following two fields really only contain 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetQuadmesh() which
    * can cause three doubles to be stored there instead of three floats.
    */
    float          min_extents[6];  /* Min mesh extents [ndims] */
    float          max_extents[6];  /* Max mesh extents [ndims] */

    char          *labels[3];   /* Label associated with each dimension */
    char          *units[3];    /* Units for variable, e.g, 'mm/ms' */
    int            ndims;       /* Number of computational dimensions */
    int            nspace;      /* Number of physical dimensions */
```

(continues on next page)

(continued from previous page)

```
    int             nnodes;      /* Total number of nodes */

    int             dims[3];     /* Number of nodes per dimension */
    int             origin;      /* '0' or '1' */
    int             min_index[3];   /* Index in each dimension of 1st
                                     * non-phoney */
    int             max_index[3];   /* Index in each dimension of last
                                     * non-phoney */
    int             base_index[3];  /* Lowest real i,j,k value for this block */
    int             start_index[3]; /* i,j,k values corresponding to original
                                     * mesh */
    int             size_index[3];  /* Number of nodes per dimension for
                                     * original mesh */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char           *mrgtree_name; /* optional name of assoc. mrgtree object */
    char           *ghost_node_labels;
    char           *ghost_zone_labels;
    char          **alt_nodenum_vars;
    char          **alt_zonenum_vars;
} DBquadmesh;
```

1.16.15 DBucdmesh

```
typedef struct DBucdmesh_ {
/*------------ Unstructured Cell Data (UCD) Mesh ------------*/
    int             id;          /* Identifier for this object */
    int             block_no;    /* Block number for this mesh */
    int             group_no;    /* Block group number for this mesh */
    char           *name;        /* Name associated with mesh */
    int             cycle;       /* Problem cycle number */
    int             coord_sys;   /* Coordinate system */
    int             topo_dim;    /* Topological dimension. */
    char           *units[3];    /* Units for variable, e.g, 'mm/ms' */
    char           *labels[3];   /* Label associated with each dimension */

    void           *coords[3];   /* Mesh node coordinates */
    int             datatype;    /* Type of coordinate arrays (double,float) */
    float           time;        /* Problem time */
    double          dtime;       /* Problem time, double data type */
   /*
    * The following two fields really only contain 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetUcdmesh() which
    * can cause three doubles to be stored there instead of three floats.
    */
    float           min_extents[6];  /* Min mesh extents [ndims] */
    float           max_extents[6];  /* Max mesh extents [ndims] */

    int             ndims;       /* Number of computational dimensions */
    int             nnodes;      /* Total number of nodes */
    int             origin;      /* '0' or '1' */
```

(continues on next page)

(continued from previous page)

```
    DBfacelist    *faces;       /* Data structure describing mesh faces */
    DBzonelist    *zones;       /* Data structure describing mesh zones */
    DBedgelist    *edges;       /* Data struct describing mesh edges
                                 * (option) */

/*--------- Optional node attributes ---------*/
    void          *gnodeno;     /* [nnodes] global node number of each node */

/*--------- Optional zone attributes ---------*/
    int           *nodeno;      /* [nnodes] node number of each node */

/*--------- Optional polyhedral zonelist ---------*/
    DBphzonelist  *phzones;     /* Data structure describing mesh zones */

    int            guihide;     /* Flag to hide from post-processor's GUI */
    char          *mrgtree_name; /* optional name of assoc. mrgtree object */
    int            tv_connectivity;
    int            disjoint_mode;
    int            gnznodtype;  /* datatype for global node/zone ids */
    char          *ghost_node_labels;
    char         **alt_nodenum_vars;
} DBucdmesh;
```

1.16.16 DBquadvar

```
typedef struct DBquadvar_ {
/*----------- Quad Variable -----------*/
    int            id;          /* Identifier for this object */
    char          *name;        /* Name of variable */
    char          *units;       /* Units for variable, e.g, 'mm/ms' */
    char          *label;       /* Label (perhaps for editing purposes) */
    int            cycle;       /* Problem cycle number */
    int            meshid;      /* Identifier for associated mesh (Deprecated Sep2005) */

    void         **vals;        /* Array of pointers to data arrays */
    int            datatype;    /* Type of data pointed to by 'val' */
    int            nels;        /* Number of elements in each array */
    int            nvals;       /* Number of arrays pointed to by 'vals' */
    int            ndims;       /* Rank of variable */
    int            dims[3];     /* Number of elements in each dimension */

    int            major_order; /* 1 indicates row-major for multi-d arrays */
    int            stride[3];   /* Offsets to adjacent elements  */
    int            min_index[3];  /* Index in each dimension of 1st
                                   * non-phoney */
    int            max_index[3];  /* Index in each dimension of last
                                   * non-phoney */
    int            origin;      /* '0' or '1' */
    float          time;        /* Problem time */
```

(continues on next page)

(continued from previous page)

```
    double         dtime;       /* Problem time, double data type */
   /*
    * The following field really only contains 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetQuadvar() which
    * can cause three doubles to be stored there instead of three floats.
    */
    float          align[6];    /* Centering and alignment per dimension */

    void         **mixvals;     /* nvals ptrs to data arrays for mixed zones */
    int            mixlen;      /* Num of elmts in each mixed zone data
                                 * array */

    int            use_specmf;  /* Flag indicating whether to apply species
                                 * mass fractions to the variable. */

    int            ascii_labels;/* Treat variable values as ASCII values
                                   by rounding to the nearest integer in
                                   the range [0, 255] */
    char           *meshname;   /* Name of associated mesh */
    int            guihide;     /* Flag to hide from post-processor's GUI */
    char         **region_pnames;
    int            conserved;   /* indicates if the variable should be conserved
                                   under various operations such as interp. */
    int            extensive;   /* indicates if the variable reprsents an extensiv
                                   physical property (as opposed to intensive) */
    int            centering;   /* explicit centering knowledge; should agree
                                   with alignment. */
    double         missing_value; /* Value to indicate var data is invalid/missing */
} DBquadvar;
```

1.16.17 DBucdvar

```
typedef struct DBucdvar_ {
/*------------ Unstructured Cell Data (UCD) Variable ------------*/
    int            id;          /* Identifier for this object */
    char          *name;        /* Name of variable */
    int            cycle;       /* Problem cycle number */
    char          *units;       /* Units for variable, e.g, 'mm/ms' */
    char          *label;       /* Label (perhaps for editing purposes) */
    float          time;        /* Problem time */
    double         dtime;       /* Problem time, double data type */
    int            meshid;      /* Identifier for associated mesh (Deprecated Sep2005) */

    void         **vals;        /* Array of pointers to data arrays */
    int            datatype;    /* Type of data pointed to by 'vals' */
    int            nels;        /* Number of elements in each array */
    int            nvals;       /* Number of arrays pointed to by 'vals' */
    int            ndims;       /* Rank of variable */
    int            origin;      /* '0' or '1' */
```

(continues on next page)

(continued from previous page)

```
    int             centering;   /* Centering within mesh (nodal or zonal) */
    void          **mixvals;     /* nvals ptrs to data arrays for mixed zones */
    int             mixlen;      /* Num of elmts in each mixed zone data
                                  * array */

    int             use_specmf;  /* Flag indicating whether to apply species
                                  * mass fractions to the variable. */
    int             ascii_labels;/* Treat variable values as ASCII values
                                    by rounding to the nearest integer in
                                    the range [0, 255] */
    char           *meshname;    /* Name of associated mesh */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char          **region_pnames;
    int             conserved;   /* indicates if the variable should be conserved
                                    under various operations such as interp. */
    int             extensive;   /* indicates if the variable reprsents an extensiv
                                    physical property (as opposed to intensive) */
    double          missing_value; /* Value to indicate var data is invalid/missing */
} DBucdvar;
```

1.16.18 DBmeshvar

```
typedef struct DBmeshvar_ {
/*------------ Generic Mesh-Data Variable ------------*/
    int             id;          /* Identifier for this object */
    char           *name;        /* Name of variable */
    char           *units;       /* Units for variable, e.g, 'mm/ms' */
    char           *label;       /* Label (perhaps for editing purposes) */
    int             cycle;       /* Problem cycle number */
    int             meshid;      /* Identifier for associated mesh (Deprecated Sep2005) */

    void          **vals;        /* Array of pointers to data arrays */
    int             datatype;    /* Type of data pointed to by 'val' */
    int             nels;        /* Number of elements in each array */
    int             nvals;       /* Number of arrays pointed to by 'vals' */
    int             nspace;      /* Spatial rank of variable */
    int             ndims;       /* Rank of 'vals' array(s) (computatnl rank) */

    int             origin;      /* '0' or '1' */
    int             centering;   /* Centering within mesh (nodal,zonal,other) */
    float           time;        /* Problem time */
    double          dtime;       /* Problem time, double data type */
   /*
    * The following field really only contains 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetPointvar() which
    * can cause three doubles to be stored there instead of three floats.
    */
    float           align[6];    /* Alignmnt per dimension if
                                  * centering==other */
```

(continues on next page)

(continued from previous page)

```
    /* Stuff for multi-dimensional arrays (ndims > 1) */
    int             dims[3];     /* Number of elements in each dimension */
    int             major_order; /* 1 indicates row-major for multi-d arrays */
    int             stride[3];   /* Offsets to adjacent elements  */
   /*
    * The following two fields really only contain 3 elements.  However, silo
    * contains a bug in PJ_ReadVariable() as called by DBGetUcdmesh() which
    * can cause three doubles to be stored there instead of three floats.
    */
    int             min_index[6];  /* Index in each dimension of 1st
                                    * non-phoney */
    int             max_index[6];  /* Index in each dimension of last
                                     non-phoney */

    int             ascii_labels;/* Treat variable values as ASCII values
                                    by rounding to the nearest integer in
                                    the range [0, 255] */
    char           *meshname;      /* Name of associated mesh */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char          **region_pnames;
    int             conserved;   /* indicates if the variable should be conserved
                                    under various operations such as interp. */
    int             extensive;   /* indicates if the variable reprsents an extensiv
                                    physical property (as opposed to intensive) */
    double          missing_value; /* Value to indicate var data is invalid/missing */
} DBmeshvar;
```

1.16.19 DBmaterial

```
typedef struct DBmaterial_ {
/*------------ Material Information ------------*/
    int             id;          /* Identifier */
    char           *name;        /* Name of this material information block */
    int             ndims;       /* Rank of 'matlist' variable */
    int             origin;      /* '0' or '1' */
    int             dims[3];     /* Number of elements in each dimension */
    int             major_order; /* 1 indicates row-major for multi-d arrays */
    int             stride[3];   /* Offsets to adjacent elements in matlist */

    int             nmat;        /* Number of materials */
    int            *matnos;      /* Array [nmat] of valid material numbers */
    char          **matnames;    /* Array of material names   */
    int            *matlist;     /* Array[nzone] w/ mat. number or mix index */
    int             mixlen;      /* Length of mixed data arrays (mix_xxx) */
    int             datatype;    /* Type of volume-fractions (double,float) */
    void           *mix_vf;      /* Array [mixlen] of volume fractions */
    int            *mix_next;    /* Array [mixlen] of mixed data indeces */
    int            *mix_mat;     /* Array [mixlen] of material numbers */
    int            *mix_zone;    /* Array [mixlen] of back pointers to mesh */
```

(continues on next page)

(continued from previous page)

```
    char         **matcolors;   /* Array of material colors */
    char          *meshname;    /* Name of associated mesh */
    int            allowmat0;   /* Flag to allow material "0" */
    int            guihide;     /* Flag to hide from post-processor's GUI */
} DBmaterial;
```

1.16.20 DBmatspecies

```
typedef struct DBmatspecies_ {
/*------------ Species Information ------------*/
    int            id;          /* Identifier */
    char          *name;        /* Name of this matspecies information block */
    char          *matname;     /* Name of material object with which the
                                 * material species object is associated. */
    int            nmat;        /* Number of materials */
    int           *nmatspec;    /* Array of lngth nmat of the num of material
                                 * species associated with each material. */
    int            ndims;       /* Rank of 'speclist' variable */
    int            dims[3];     /* Number of elements in each dimension of the
                                 * 'speclist' variable. */
    int            major_order; /* 1 indicates row-major for multi-d arrays */
    int            stride[3];   /* Offsts to adjacent elmts in 'speclist'  */

    int            nspecies_mf; /* Total number of species mass fractions. */
    void          *species_mf;  /* Array of length nspecies_mf of mass
                                 * frations of the material species. */
    int           *speclist;    /* Zone array of dimensions described by ndims
                                 * and dims.  Each element of the array is an
                                 * index into one of the species mass fraction
                                 * arrays.  A positive value is the index in
                                 * the species_mf array of the mass fractions
                                 * of the clean zone's material species:
                                 * species_mf[speclist[i]] is the mass fraction
                                 * of the first species of material matlist[i]
                                 * in zone i. A negative value means that the
                                 * zone is a mixed zone and that the array
                                 * mix_speclist contains the index to the
                                 * species mas fractions: -speclist[i] is the
                                 * index in the 'mix_speclist' array for zone
                                 * i. */
    int            mixlen;      /* Length of 'mix_speclist' array. */
    int           *mix_speclist;  /* Array of lgth mixlen of 1-orig indices
                                   * into the 'species_mf' array.
                                   * species_mf[mix_speclist[j]] is the index
                                   * in array species_mf' of the first of the
                                   * mass fractions for material
                                   * mix_mat[j]. */

    int            datatype;    /* Datatype of mass fraction data. */
    int            guihide;     /* Flag to hide from post-processor's GUI */
```

(continues on next page)

(continued from previous page)

```
    char         **specnames;   /* Array of species names; length is sum of nmatspec   */
    char         **speccolors;  /* Array of species colors; length is sum of nmatspec */
} DBmatspecies;
```

1.16.21 DBcsgzonelist

```
typedef struct DBcsgzonelist_ {
/*----------- CSG Zonelist -----------*/
    int            nregs;       /* Number of regions in regionlist */
    int            origin;      /* '0' or '1' */

    int           *typeflags;   /* [nregs] type info about each region */
    int           *leftids;     /* [nregs] left operand region refs */
    int           *rightids;    /* [nregs] right operand region refs */
    void          *xform;       /* [lxforms] transformation coefficients */
    int            lxform;      /* length of xforms array */
    int            datatype;    /* type of data in xforms array */

    int            nzones;      /* number of zones */
    int           *zonelist;    /* [nzones] region ids (complete regions) */
    int            min_index;   /* Index of first real zone */
    int            max_index;   /* Index of last real zone */

/*--------- Optional zone attributes ---------*/
    char         **regnames;    /* [nregs] names of each region */
    char         **zonenames;   /* [nzones] names of each zone */
    char         **alt_zonenum_vars;
} DBcsgzonelist;
```

1.16.22 DBcsgmesh

```
typedef struct DBcsgmesh_ {
/*----------- CSG Mesh -----------*/
    int            block_no;    /* Block number for this mesh */
    int            group_no;    /* Block group number for this mesh */
    char          *name;        /* Name associated with mesh */
    int            cycle;       /* Problem cycle number */
    char          *units[3];    /* Units for variable, e.g, 'mm/ms' */
    char          *labels[3];   /* Label associated with each dimension */

    int            nbounds;     /* Total number of boundaries */
    int           *typeflags;   /* nbounds boundary type info flags */
    int           *bndids;      /* optional, nbounds explicit ids */

    void          *coeffs;      /* coefficients in the representation of
                                   each boundary */
    int            lcoeffs;     /* length of coeffs array */
    int           *coeffidx;    /* array of nbounds offsets into coeffs
                                   for each boundary's coefficients */
```

(continues on next page)

(continued from previous page)

```
    int             datatype;    /* data type of coeffs data */

    float           time;        /* Problem time */
    double          dtime;       /* Problem time, double data type */
    double          min_extents[3];  /* Min mesh extents [ndims] */
    double          max_extents[3];  /* Max mesh extents [ndims] */

    int             ndims;       /* Number of spatial & topological dimensions */
    int             origin;      /* '0' or '1' */

    DBcsgzonelist *zones;        /* Data structure describing mesh zones */

/*--------- Optional boundary attributes ---------*/
    char          **bndnames;     /* [nbounds] boundary names */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char           *mrgtree_name; /* optional name of assoc. mrgtree object */
    int             tv_connectivity;
    int             disjoint_mode;
    char          **alt_nodenum_vars;
} DBcsgmesh;
```

1.16.23 DBcsgvar

```
typedef struct DBcsgvar_ {
/*----------- CSG Variable -----------*/
    char           *name;        /* Name of variable */
    int             cycle;       /* Problem cycle number */
    char           *units;       /* Units for variable, e.g, 'mm/ms' */
    char           *label;       /* Label (perhaps for editing purposes) */
    float           time;        /* Problem time */
    double          dtime;       /* Problem time, double data type */

    void          **vals;        /* Array of pointers to data arrays */
    int             datatype;    /* Type of data pointed to by 'vals' */
    int             nels;        /* Number of elements in each array */
    int             nvals;       /* Number of arrays pointed to by 'vals' */

    int             centering;   /* Centering within mesh (nodal or zonal) */

    int             use_specmf;  /* Flag indicating whether to apply species
                                  * mass fractions to the variable. */
    int             ascii_labels;/* Treat variable values as ASCII values
                                    by rounding to the nearest integer in
                                    the range [0, 255] */
    char           *meshname;    /* Name of associated mesh */
    int             guihide;     /* Flag to hide from post-processor's GUI */
    char          **region_pnames;
    int             conserved;   /* indicates if the variable should be conserved
                                    under various operations such as interp. */
    int             extensive;   /* indicates if the variable reprsents an extensiv
```

(continues on next page)

(continued from previous page)

```
                                    physical property (as opposed to intensive) */
    double         missing_value; /* Value to indicate var data is invalid/missing */
} DBcsgvar;
```

1.16.24 DBmrgtree

```
typedef struct _DBmrgtree {
    char *name;
    char *src_mesh_name;
    int src_mesh_type;
    int type_info_bits;
    int num_nodes;
    DBmrgtnode *root;
    DBmrgtnode *cwr;

    char **mrgvar_onames;
    char **mrgvar_rnames;
} DBmrgtree;
```

1.16.25 DBmrgvar

```
typedef struct _DBmrgvar {
    char *name;
    char *mrgt_name;
    int ncomps;
    char **compnames;
    int nregns;
    char **reg_pnames;
    int datatype;
    void **data;
} DBmrgvar ;
```

1.16.26 DBgroupelmap

```
typedef struct _DBgroupelmap {
    char *name;
    int num_segments;
    int *groupel_types;
    int *segment_lengths;
    int *segment_ids;
    int **segment_data;
    void **segment_fracs;
    int fracs_data_type;
} DBgroupelmap;
```

1.16.27 DBcompoundarray

```
typedef struct DBcompoundarray_ {
    int            id;          /*identifier of the compound array */
    char          *name;        /*name of te compound array  */
    char         **elemnames;   /*names of the simple array elements */
    int           *elemlengths; /*lengths of the simple arrays  */
    int            nelems;      /*number of simple arrays  */
    void          *values;      /*simple array values   */
    int            nvalues;     /*sum reduction of `elemlengths' vector */
    int            datatype;    /*simple array element data type */
} DBcompoundarray;
```

1.16.28 DBoptlist

```
typedef struct DBoptlist_ {

    int           *options;     /* Vector of option identifiers */
    void         **values;      /* Vector of pointers to option values */
    int            numopts;     /* Number of options defined */
    int            maxopts;     /* Total length of option/value arrays */

} DBoptlist;
```

1.16.29 DBobject

```
typedef struct DBobject_ {

    char          *name;
    char          *type;        /* Type of group/object */
    char         **comp_names;  /* Array of component names */
    char         **pdb_names;   /* Array of internal (PDB) variable names */
    int            ncomponents; /* Number of components */
    int            maxcomponents;  /* Max number of components */

    /* fields below are a hack for HDF5 driver to handle
       customization of 'standard' objects */
    char           h5_vals[DB_MAX_H5_OBJ_VALS*3*sizeof(double)];
    int            h5_offs[DB_MAX_H5_OBJ_VALS];
    int            h5_sizes[DB_MAX_H5_OBJ_VALS];
    int            h5_types[DB_MAX_H5_OBJ_VALS];
    char          *h5_names[DB_MAX_H5_OBJ_VALS];
} DBobject;
```

1.16.30 DBdatatype

```
/* Data types */
typedef enum {
    DB_INT=16,
    DB_SHORT=17,
    DB_LONG=18,
    DB_FLOAT=19,
    DB_DOUBLE=20,
    DB_CHAR=21,
    DB_LONG_LONG=22,
    DB_NOTYPE=25            /*unknown type */
} DBdatatype;
```

1.16.31 DBObjectType

```
/* Objects that can be stored in a data file */
typedef enum {
    DB_INVALID_OBJECT= -1,       /*causes enum to be signed, do not remove,
                                   space before minus sign necessary for lint*/
    DB_QUADRECT = DB_QUAD_RECT,
    DB_QUADCURV = DB_QUAD_CURV,
    DB_QUADMESH=500,
    DB_QUADVAR=501,
    DB_UCDMESH=510,
    DB_UCDVAR=511,
    DB_MULTIMESH=520,
    DB_MULTIVAR=521,
    DB_MULTIMAT=522,
    DB_MULTIMATSPECIES=523,
    DB_MULTIBLOCKMESH=DB_MULTIMESH,
    DB_MULTIBLOCKVAR=DB_MULTIVAR,
    DB_MULTIMESHADJ=524,
    DB_MATERIAL=530,
    DB_MATSPECIES=531,
    DB_FACELIST=550,
    DB_ZONELIST=551,
    DB_EDGELIST=552,
    DB_PHZONELIST=553,
    DB_CSGZONELIST=554,
    DB_CSGMESH=555,
    DB_CSGVAR=556,
    DB_CURVE=560,
    DB_DEFVARS=565,
    DB_POINTMESH=570,
    DB_POINTVAR=571,
    DB_ARRAY=580,
    DB_DIR=600,
    DB_SYMLINK=601,
    DB_VARIABLE=610,
    DB_MRGTREE=611,
```

(continues on next page)

(continued from previous page)

```
    DB_GROUPELMAP=612,
    DB_MRGVAR=613,
    DB_USERDEF=700
} DBObjectType;
```

1.16.32 Open/Create flags

```
/* Flags for DBCreate */
#define         DB_CLOBBER      0
#define         DB_NOCLOBBER    1

/* Flags for DBOpen */
#define         DB_READ         1
#define         DB_APPEND       2

/* Compatibility flags for DBOpen|DBCreate
   These can get OR'd into above flags and
   occupy the 2nd most significant nibble. */
#define DB_COMPAT_OVER_PERF     0x00000010
#define DB_PERF_OVER_COMPAT     0x00000020

/* Target machine for DBCreate */
#define         DB_LOCAL        0
#define         DB_SUN3         10
#define         DB_SUN4         11
#define         DB_SGI          12
#define         DB_RS6000       13
#define         DB_CRAY         14
#define         DB_INTEL        15
```

1.16.33 Optlist options

```
/* Options */
#define DBOPT_FIRST             260
#define DBOPT_ALIGN             260
#define DBOPT_COORDSYS          262
#define DBOPT_CYCLE             263
#define DBOPT_FACETYPE          264
#define DBOPT_HI_OFFSET         265
#define DBOPT_LO_OFFSET         266
#define DBOPT_LABEL             267
#define DBOPT_XLABEL            268
#define DBOPT_YLABEL            269
#define DBOPT_ZLABEL            270
#define DBOPT_MAJORORDER        271
#define DBOPT_NSPACE            272
#define DBOPT_ORIGIN            273
#define DBOPT_PLANAR            274
```

(continues on next page)

(continued from previous page)

```
#define DBOPT_TIME              275
#define DBOPT_UNITS             276
#define DBOPT_XUNITS            277
#define DBOPT_YUNITS            278
#define DBOPT_ZUNITS            279
#define DBOPT_DTIME             280
#define DBOPT_USESPECMF         281
#define DBOPT_XVARNAME          282
#define DBOPT_YVARNAME          283
#define DBOPT_ZVARNAME          284
#define DBOPT_ASCII_LABEL       285
#define DBOPT_MATNOS            286
#define DBOPT_NMATNOS           287
#define DBOPT_MATNAME           288
#define DBOPT_NMAT              289
#define DBOPT_NMATSPEC          290
#define DBOPT_BASEINDEX         291 /* quad meshes for node and zone */
#define DBOPT_ZONENUM           292 /* ucd meshes for zone */
#define DBOPT_NODENUM           293 /* ucd/point meshes for node */
#define DBOPT_BLOCKORIGIN       294
#define DBOPT_GROUPNUM          295
#define DBOPT_GROUPORIGIN       296
#define DBOPT_NGROUPS           297
#define DBOPT_MATNAMES          298
#define DBOPT_EXTENTS_SIZE      299
#define DBOPT_EXTENTS           300
#define DBOPT_MATCOUNTS         301
#define DBOPT_MATLISTS          302
#define DBOPT_MIXLENS           303
#define DBOPT_ZONECOUNTS        304
#define DBOPT_HAS_EXTERNAL_ZONES 305
#define DBOPT_PHZONELIST        306
#define DBOPT_MATCOLORS         307
#define DBOPT_BNDNAMES          308
#define DBOPT_REGNAMES          309
#define DBOPT_ZONENAMES         310
#define DBOPT_HIDE_FROM_GUI     311
#define DBOPT_TOPO_DIM          312 /* TOPOlogical DIMension */
#define DBOPT_REFERENCE         313 /* reference object */
#define DBOPT_GROUPINGS_SIZE    314 /* size of grouping array */
#define DBOPT_GROUPINGS         315 /* groupings array */
#define DBOPT_GROUPINGNAMES     316 /* array of size determined by
                                       number of groups of names of groups. */
#define DBOPT_ALLOWMAT0         317 /* Turn off material numer "0" warnings*/
#define DBOPT_MRGTREE_NAME      318
#define DBOPT_REGION_PNAMES     319
#define DBOPT_TENSOR_RANK       320
#define DBOPT_MMESH_NAME        321
#define DBOPT_TV_CONNECTIVITY   322
#define DBOPT_DISJOINT_MODE     323
#define DBOPT_MRGV_ONAMES       324
#define DBOPT_MRGV_RNAMES       325
```

(continues on next page)

(continued from previous page)

```
#define DBOPT_SPECNAMES          326
#define DBOPT_SPECCOLORS         327
#define DBOPT_LLONGNZNUM         328
#define DBOPT_CONSERVED          329
#define DBOPT_EXTENSIVE          330
#define DBOPT_MB_FILE_NS         331
#define DBOPT_MB_BLOCK_NS        332
#define DBOPT_MB_BLOCK_TYPE      333
#define DBOPT_MB_EMPTY_LIST      334
#define DBOPT_MB_EMPTY_COUNT     335
#define DBOPT_MB_REPR_BLOCK_IDX 336
#define DBOPT_MISSING_VALUE      337
#define DBOPT_ALT_ZONENUM_VARS   338
#define DBOPT_ALT_NODENUM_VARS   339
#define DBOPT_GHOST_NODE_LABELS 340
#define DBOPT_GHOST_ZONE_LABELS 341
#define DBOPT_LAST               499
```

1.16.34 VFD options

```
/* Options relating to virtual file drivers */
#define DBOPT_H5_FIRST              500
#define DBOPT_H5_VFD                500
#define DBOPT_H5_RAW_FILE_OPTS      501
#define DBOPT_H5_RAW_EXTENSION      502
#define DBOPT_H5_META_FILE_OPTS     503
#define DBOPT_H5_META_EXTENSION     504
#define DBOPT_H5_CORE_ALLOC_INC     505
#define DBOPT_H5_CORE_NO_BACK_STORE 506
#define DBOPT_H5_META_BLOCK_SIZE    507
#define DBOPT_H5_SMALL_RAW_SIZE     508
#define DBOPT_H5_ALIGN_MIN          509
#define DBOPT_H5_ALIGN_VAL          510
#define DBOPT_H5_DIRECT_MEM_ALIGN   511
#define DBOPT_H5_DIRECT_BLOCK_SIZE  512
#define DBOPT_H5_DIRECT_BUF_SIZE    513
#define DBOPT_H5_LOG_NAME           514
#define DBOPT_H5_LOG_BUF_SIZE       515
#define DBOPT_H5_MPIO_COMM          516
#define DBOPT_H5_MPIO_INFO          517
#define DBOPT_H5_MPIP_NO_GPFS_HINTS 518
#define DBOPT_H5_SIEVE_BUF_SIZE     519
#define DBOPT_H5_CACHE_NELMTS       520
#define DBOPT_H5_CACHE_NBYTES       521
#define DBOPT_H5_CACHE_POLICY       522
#define DBOPT_H5_FAM_SIZE           523
#define DBOPT_H5_FAM_FILE_OPTS      524
#define DBOPT_H5_USER_DRIVER_ID     525
#define DBOPT_H5_USER_DRIVER_INFO   526
#define DBOPT_H5_SILO_BLOCK_SIZE    527
```

(continues on next page)

(continued from previous page)

```
#define DBOPT_H5_SILO_BLOCK_COUNT   528
#define DBOPT_H5_SILO_LOG_STATS     529
#define DBOPT_H5_SILO_USE_DIRECT    530
#define DBOPT_H5_FIC_SIZE           531
#define DBOPT_H5_FIC_BUF            532
#define DBOPT_H5_FCPL_HID_T         533
#define DBOPT_H5_FAPL_HID_T         534
#define DBOPT_H5_LAST               599
```

1.16.35 Error handling options

```
/* Error trapping method */
#define         DB_TOP          0 /*default--API traps  */
#define         DB_NONE         1 /*no errors trapped  */
#define         DB_ALL          2 /*all levels trap (traceback) */
#define         DB_ABORT        3 /*abort() is called  */
#define         DB_SUSPEND      4 /*suspend error reporting temporarily */
#define         DB_RESUME       5 /*resume normal error reporting */
#define         DB_ALL_AND_DRVR 6 /*DB_ALL + driver error reporting */
```

1.16.36 Error return codes

```
/* Errors */
#define     E_NOERROR   0       /*No error   */
#define     E_BADFTYPE  1       /*Bad file type   */
#define     E_NOTIMP    2       /*Callback not implemented */
#define     E_NOFILE    3       /*No data file specified    */
#define     E_INTERNAL  5       /*Internal error        */
#define     E_NOMEM     6       /*Not enough memory     */
#define     E_BADARGS   7       /*Bad argument to function  */
#define     E_CALLFAIL  8       /*Low-level function failure    */
#define     E_NOTFOUND  9       /*Object not found      */
#define     E_TAURSTATE 10      /*Taurus: database state error  */
#define     E_MSERVER   11      /*SDX: too many connections */
#define     E_PROTO     12      /*SDX: protocol error       */
#define     E_NOTDIR    13      /*Not a directory       */
#define     E_MAXOPEN   14      /*Too many open files  */
#define     E_NOTFILTER 15      /*Filter(s) not found  */
#define     E_MAXFILTERS    16  /*Too many filters  */
#define     E_FEXIST    17      /*File already exists  */
#define     E_FILEISDIR 18      /*File is actually a directory */
#define     E_FILENOREAD    19  /*File lacks read permission. */
#define     E_SYSTEMERR 20      /*System level error occured. */
#define     E_FILENOWRITE 21    /*File lacks write permission. */
#define     E_INVALIDNAME 22    /* Variable name is invalid */
#define     E_NOOVERWRITE 23    /*Overwrite not permitted */
#define     E_CHECKSUM  24      /*Checksum failed */
#define     E_COMPRESSION  25   /*Compression failed */
#define     E_GRABBED   26      /*Low level driver enabled */
```

(continues on next page)

(continued from previous page)

```
#define     E_NOTREG    27      /*The dbfile pointer is not resitered. */
#define     E_CONCURRENT 28     /*File is opened multiply and not all read-only. */
#define     E_DRVRCANTOPEN 29   /*Driver cannot open the file. */
#define     E_BADOPTCLASS 30    /*Optlist contains options for wrong class */
#define     E_NOTENABLEDINBUILD 31 /*feature not enabled in this build */
#define     E_MAXFILEOPTSETS 32 /*Too many file options sets */
#define     E_NOHDF5 33         /*HDF5 driver not available */
#define     E_EMPTYOBJECT 34    /*Empty object not currently permitted*/
#define     E_OBJBUFFULL  35    /*No more temp. buffer space for object */
#define     E_NOSILOHDF5 36     /*Not HDF5 silo produced by silo */
#define     E_NERRORS   50
```

1.16.37 Major order options

```
/* Definitions for MAJOR_ORDER */
#define  DB_ROWMAJOR            0
#define  DB_COLMAJOR            1
```

1.16.38 Variable centering options

```
/* Definitions for CENTERING */
#define  DB_NOTCENT             0
#define  DB_NODECENT            110
#define  DB_ZONECENT            111
#define  DB_FACECENT            112
#define  DB_BNDCENT             113 /* for CSG meshes only */
#define  DB_EDGECENT            114
#define  DB_BLOCKCENT           115 /* for 'block-centered' data on multimeshs */
```

1.16.39 Coordinate system options

```
/* Definitions for COORD_SYSTEM */
#define  DB_CARTESIAN           120
#define  DB_CYLINDRICAL         121 /* x,r;  y,theta;  z,height; 2D variant is eqiv. to␣
↪poloar */
#define  DB_SPHERICAL           122 /* x,r;  y,theta;  z,phi;  2D variant is equiv. to␣
↪polar */
#define  DB_NUMERICAL           123
#define  DB_OTHER               124
```

1.16.40 Derived variable types

```
/* Definitions for derived variable types */
#define DB_VARTYPE_SCALAR               200
#define DB_VARTYPE_VECTOR               201
#define DB_VARTYPE_TENSOR               202
#define DB_VARTYPE_SYMTENSOR            203
#define DB_VARTYPE_ARRAY                204
#define DB_VARTYPE_MATERIAL             205
#define DB_VARTYPE_SPECIES              206
#define DB_VARTYPE_LABEL                207
```

1.16.41 CSG options

```
/* Definitions for CSG boundary types
   Designed so low-order 16 bits are unused.

   The last few characters of the symbol are intended
   to indicate the representational form of the surface type

   G = generalized form  (n values, depends on surface type)
   P = point (3 values, x,y,z in 3D, 2 values in 2D x,y)
   N = normal (3 values, Nx,Ny,Nz in 3D, 2 values in 2D Nx,Ny)
   R = radius (1 value)
   A = angle (1 value in degrees)
   L = length (1 value)
   X = x-intercept (1 value)
   Y = y-intercept (1 value)
   Z = z-intercept (1 value)
   K = arbitrary integer
   F = planar face defined by point-normal pair (6 values)
   */
#define DBCSG_QUADRIC_G         0x01000000
#define DBCSG_SPHERE_PR         0x02010000
#define DBCSG_ELLIPSOID_PRRR    0x02020000
#define DBCSG_PLANE_G           0x03000000
#define DBCSG_PLANE_X           0x03010000
#define DBCSG_PLANE_Y           0x03020000
#define DBCSG_PLANE_Z           0x03030000
#define DBCSG_PLANE_PN          0x03040000
#define DBCSG_PLANE_PPP         0x03050000
#define DBCSG_CYLINDER_PNLR     0x04000000
#define DBCSG_CYLINDER_PPR      0x04010000
#define DBCSG_BOX_XYZXYZ        0x05000000
#define DBCSG_CONE_PNLA         0x06000000
#define DBCSG_CONE_PPA          0x06010000
#define DBCSG_POLYHEDRON_KF     0x07000000
#define DBCSG_HEX_6F            0x07010000
#define DBCSG_TET_4F            0x07020000
#define DBCSG_PYRAMID_5F        0x07030000
#define DBCSG_PRISM_5F          0x07040000
```

(continues on next page)

(continued from previous page)

```
/* Definitions for 2D CSG boundary types */
#define DBCSG_QUADRATIC_G       0x08000000
#define DBCSG_CIRCLE_PR         0x09000000
#define DBCSG_ELLIPSE_PRR       0x09010000
#define DBCSG_LINE_G            0x0A000000
#define DBCSG_LINE_X            0x0A010000
#define DBCSG_LINE_Y            0x0A020000
#define DBCSG_LINE_PN           0x0A030000
#define DBCSG_LINE_PP           0x0A040000
#define DBCSG_BOX_XYXY          0x0B000000
#define DBCSG_ANGLE_PNLA        0x0C000000
#define DBCSG_ANGLE_PPA         0x0C010000
#define DBCSG_POLYGON_KP        0x0D000000
#define DBCSG_TRI_3P            0x0D010000
#define DBCSG_QUAD_4P           0x0D020000

/* Definitions for CSG Region operators */
#define DBCSG_INNER             0x7F000000
#define DBCSG_OUTER             0x7F010000
#define DBCSG_ON                0x7F020000
#define DBCSG_UNION             0x7F030000
#define DBCSG_INTERSECT         0x7F040000
#define DBCSG_DIFF              0x7F050000
#define DBCSG_COMPLIMENT        0x7F060000
#define DBCSG_XFORM             0x7F070000
#define DBCSG_SWEEP             0x7F080000
```

CHAPTER

TWO

CONTACTS

For inquiries, users are encouraged to use Silo's GitHub Discussions space. GitHub accounts are free.