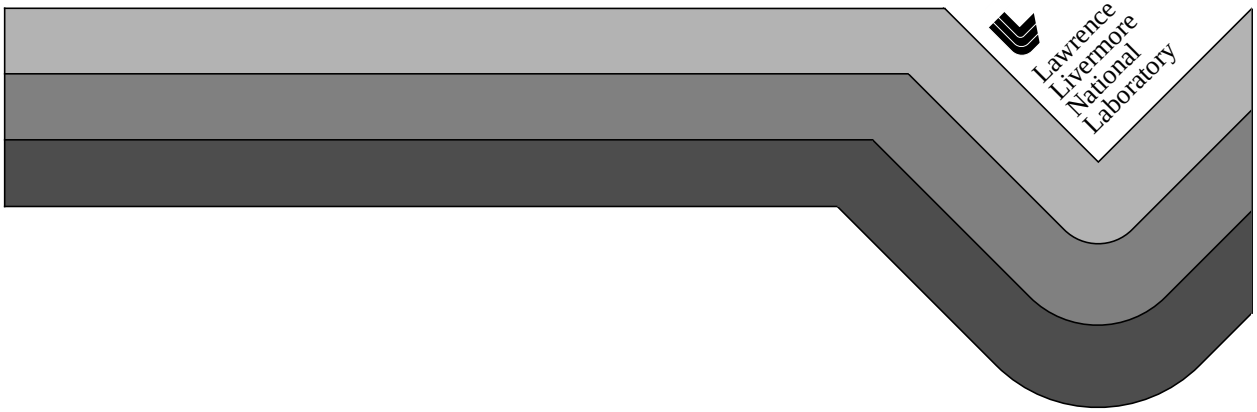
Silo User's Guide

Revision: November 2009
Version: 4.7.1 of the Silo Library
Document Release Number LLNL-SM-421083



This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory in part under Contract W-7405-Eng-48 and in part under Contract DE-AC52-07NA27344.

Chapter 1 Introduction to Silo

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, disk 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[1] 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 these meshes. In addition, Silo supports a wide array of other useful objects to address various scientific computing applications' needs.

Although the Silo library is a serial library, it has key features which enable it to be applied quite effectively and scalably in parallel.

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)[2] and PDB (Portable Data Base, a binary database file format developed at LLNL by Stewart Brown) drivers. However, the reader should take care not to infer

1. VisIt can be obtained from http://www.llnl.gov/visit

from this that Silo can read *any* HDF5 file. It cannot. For the most part, Silo is able to read only files that it has also written.

1.2. 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 visualization, data differencing, browsing 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, typically 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 support 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 the "objects" of computer science. That is arrays, structs and linked lists (e.g. data structures). The other focused on the "objects" of computational modeling. That is structured and unstructured meshes with piecewise-constant and piecewise-linear fields. Examples of the former are CDF, HDF (HDF4 and HDF5) and PDBLib. Silo is an example of the latter type of I/O library. At the same time, Silo makes use of the former.

2. The National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign (UIUC). The HDF5 software can be obtained from http://hdf5.ncsa.uiuc.edu/HDF5/release/obtain5.html.

1.3. Silo Architecture

Silo has several drivers. Some are read-only and some are read-write. These are illustrated in Figure 1-1:

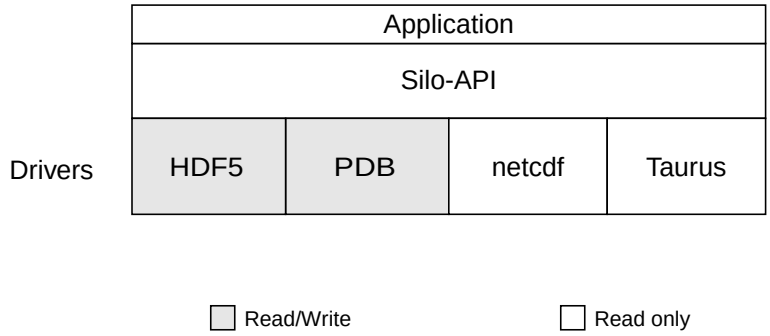


Figure 1-1: Model of Silo Architecture.

Silo supports both read and write on the PDB (Portable Database) formatted files and HDF5 drivers. However, Silo 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.

1.3.1. 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.

1.3.2. 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.

1.4. 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.

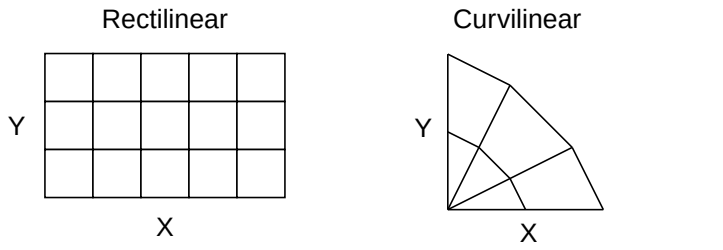
Block	This is the fundamental building block of a computational mesh. It defines the nodal coordinates of one contiguous section of a mesh (also known as a mesh-block).
Mesh	A computational mesh, composed of one or more mesh-blocks. A mesh can be composed of mesh-blocks of different types (quad, UCD) as well as of different shapes.
Variable	Data which are associated in some way with a computational mesh. Variables usually represent values of some physics quantity (e.g., pressure). Values are usually located either at the mesh nodes or at zone centers.
Material	A physical material being modeled in a computer simulation.
Node	A mathematical point. The fundamental building-block of a mesh or zone.
Zone	An area or volume of which meshes are comprised. Zones are polygons or polyhedra with nodes as vertices (see “UCD 2-D and 3-D Cell Shapes” on page 1-6.)

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.

1.5.3. 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 (Fig. 1-2).




```
X = {0.0,1.0,2.0,3.0,
     4.0,5.0}
Y = {0.0,1.0,2.0,3.0}
```

```
X = {0.0,1.0,2.0,
     0.0,0.8,1.6,
     0.0,0.4,0.8,
     0.0,0.0,0.0}
Y = {0.0,0.0,0.0,
     0.0,0.4,0.8,
     0.0,0.8,1.6,
     0.0,1.0,2.0}
```

Figure 1-2: Examples of quadrilateral meshes.

1.5.4. UCD-Based Meshes and Related Data

An unstructured (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*.

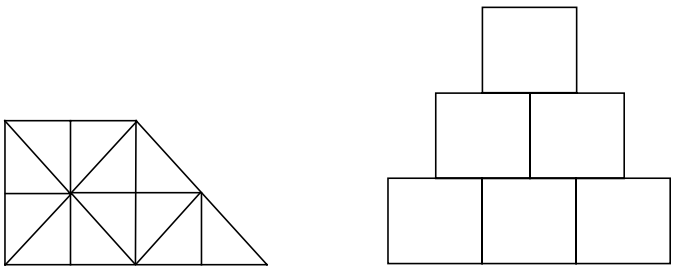


Figure 1-3: Sample 2-D UCD Meshes

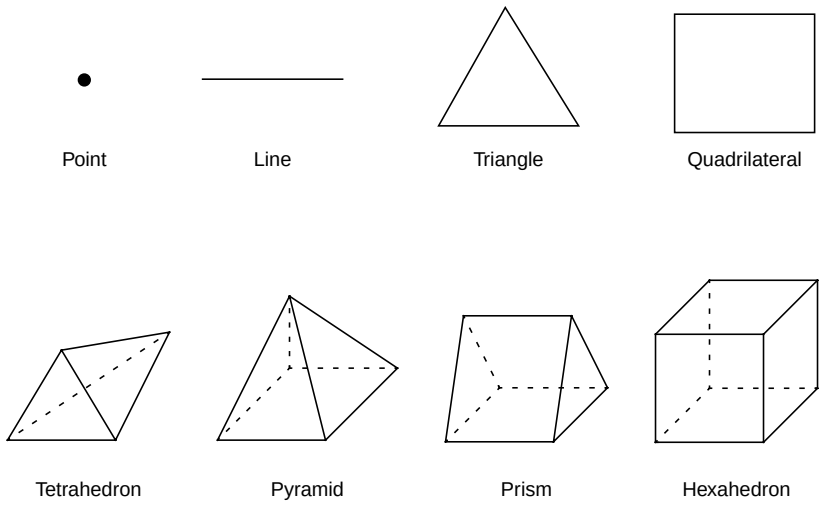


Figure 1-4: UCD 2-D and 3-D Cell Shapes

1.5.5. 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.

1.5.6. 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.

1.5.7. 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.

1.6. 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 quadrilateral mesh. At a minimum, this must include the dimension and coordinate data, but typically also includes the mesh's coordinate system, labelling and unit information, minimum and maximum extents, and valid index ranges.
Quadvar	A variable associated with a quadrilateral mesh. At a minimum, this must include 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 mesh[1]. At a minimum, this must include 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.
Ucdvar	A variable associated with a UCD mesh. This at a minimum must include 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 point mesh. At a minimum, this must include dimension and coordinate data.
Csgmesh	A constructive solid geometry (CSG) mesh.
Csgvar	A variable defined on a CSG mesh (always zone centered).
Defvar	Defined variable representing an arithmetic expression involving other variables.
Groupel Map	Used in concert with an MRG tree to define subsetted regions of meshes.
Multimat	A set of materials. 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.
Multimesh	A set of meshes. This object contains the names of and types of the meshes in the set.
Multivar	Mesh variable data associated with a multimesh.

1. Unstructured cell data (UCD) is a term commonly used to denote an arbitrarily connected mesh. Such a mesh is composed of vectors of coordinate values along with an index array which identifies the nodes associated with each zone and/or face. Zones may contain any number of nodes for 2-D meshes, and either four, five, six, or eight nodes for 3-D meshes.

Material	Material information. This includes the number of materials present, a list of valid material identifiers, and a zonal-length array which contains the material identifiers for each zone.
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.
MRG Tree	Mesh Region Grouping tree used to define various subset regions of any of Silo's mesh types.
Zonelist	Zone-oriented connectivity information for a UCD mesh. This object contains a sequential list of nodes which identifies the zones in the mesh, and arrays which describe the shape(s) of the zones in the mesh.
PHZonelist	Arbitrary, polyhedral extension of a zonelist.
Facelist	Face-oriented connectivity information for a UCD mesh. This object contains a sequential list of nodes which identifies the faces in the 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.
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.
Variable	Array 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 a mesh.

1.6.8. 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 computing applications. Some of the more useful ones are briefly summarized here.

Compound Array	A compound array is an abstraction of a Fortran common block. It is also somewhat like a C struct. It is a list of similarly typed by differently named and sized (usually small in size) items that one often treats as a group (particularly for I/O purposes).
Directory	A silo file can be organized into *directories* in much the same way as a UNIX™ filesystem.
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.
User Defined Object	A generic, user-defined object or arbitrary nature.

1.7. 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.

1.7.9. Limitations of Fortran Interface

First, it 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.

1.7.10. 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
```

l<strname>	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.
l<strname>s	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.
<object>_id	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* is 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.
data_ids	Wherever the C interface accepts an array of void* (e.g. a void** argument), the Fortran interface accepts an array of integer *pointer_id*s. 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.
status	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.

1.8. 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 between 8 and 64 depending on the number of concurrent I/O channels the application has available.

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. For many parallel, HPC platforms, this number is typically between 8 and 64.

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 affectionately called *Poor Man's Parallel I/O* (PMPIO).

There is a separate header file, pmpio.h, with a set of convenience methods to support PMPIO-based parallel I/O with Silo. See "Multi-Block Objects, Parallelism and Poor-Man's Parallel I/O" on page 128 and See "PMPIO_Init" on page 149 for more information.

Chapter 2 C and Fortran Functions

2.1. C Interface Overview

This chapter documents the C and Fortran interface to the Silo library. The C header file is “silo.h” and the Fortran header file is “silo.inc”

2.1.1. Optional Arguments

Many Silo functions have optional arguments. By optional, it is meant that a dummy value can be supplied instead of an actual value. An argument to a C function which the user does not want to provide, and which is documented as being optional, should be replaced with a NULL (as defined in the file `silo.h`).

2.1.2. Using the Silo Option Parameter

Many of the functions take as one of their arguments a list of option-name/option-value pairs. In this way additional information can be passed to a function without having to change the function's interface. The following sequence of function declarations outlines the procedure for creating and populating such a list:

```
DBoptlist *DBMakeOptlist (int maxopts) /* Create a list with
                                          maximum list length */

int DBAddOption (          /* Add an option to the list: */
          DBoptlist *optlist,   /* the list, */
          int option_id,        /* the option, */
          void *option_value    /* the option's value */
          )
```

2.1.3. C Calling Sequence

The functions in the Silo output package should be called in a particular order.

2.1.3.1. Write Sequence

Start by creating a Silo file, with DBCreate(), create any necessary directories, then call the remaining routines as needed for writing out the mesh, material data, and any physics variables associated with the mesh.

Schematically, your program should look something like this:

```
DBCreate

DBMkdir
DBSetDir
      DBPutQuadmesh
      DBPutQuadvar1
      DBPutQuadvar1
       . . .
DBSetDir

DBMkdir
DBSetDir
      DBPutZonelist
      DBPutFacelist
      DBPutUcdmesh
      DBPutMaterial
      DBPutUcdvar1
       . . .
DBSetDir
DBClose
```

2.1.3.2. Example of C Calling Sequence for writing

The following C code is an example of the creation of a Silo file with just one directory (the root):

```
#include <silo.h>
#include <string.h>

int main()
{
    DBfile        *file = NULL;     /* The Silo file pointer */
    char          *coordnames[2];   /* Names of the coordinates */
    float          nodex[4];        /* The coordinate arrays */
    float          nodey[4];
    float         *coordinates[2];  /* The array of coordinate
                                       arrays */
    int            dimensions[2];   /* The number of nodes in
                                      each dimension */

    /* Create the Silo file */
```

```
    file = DBCreate("sample.silo", DB_CLOBBER, DB_LOCAL, NULL,
                    DB_PDB);

    /* Name the coordinate axes 'X' and 'Y' */
    coordnames[0] = strdup("X");
    coordnames[1] = strdup("Y");

    /* Give the x coordinates of the mesh */
    nodex[0] = -1.1;
    nodex[1] = -0.1;
    nodex[2] =  1.3;
    nodex[3] =  1.7;

    /* Give the y coordinates of the mesh */
    nodey[0] = -2.4;
    nodey[1] = -1.2;
    nodey[2] =  0.4;
    nodey[3] =  0.8;

    /* How many nodes in each direction? */
    dimensions[0] = 4;
    dimensions[1] = 4;

    /* Assign coordinates to coordinates array */
    coordinates[0] = nodex;
    coordinates[1] = nodey;

    /* Write out the mesh to the file */
    DBPutQuadmesh(file, "mesh1", coordnames, coordinates,
                  dimensions, 2, DB_FLOAT, DB_COLLINEAR, NULL);

    /* Close the Silo file */
    DBClose(file);

    return (0);
}
```

2.1.3.3. Read Sequence

Start by opening the Silo file with DBOpen(), then change to the required directory, and then read the mesh, material, and variables. Schematically, your program should look something like this:

```
DBOpen

DBSetDir
      DBGetQuadmesh
      DBGetQuadvar1
      DBGetQuadvar1
       . . .
```

```
DBSetDir
     DBGetUcdmesh
     DBGetUcdvar1
     DBGetMaterial
      . . .

DBClose
```

2.2. Fortran Interface

Currently, C-callable functions exist for all routines, but Fortran-callable functions exist for only a portion of the routines. The Fortran header file is “silo.inc”.

2.2.4. Optional Arguments

The functions described below have optional arguments. By optional, it is meant that a dummy value can be supplied instead of an actual value. An argument to a Fortran function, which the user does not want to provide, and which is documented as optional, should be replaced with the parameter DB_F77NULL, which is defined in the file `silo.inc`.

2.2.5. Using the Silo Option Parameter

Many of the functions take as one of their arguments a list of option-name/option-value pairs. In this way, additional information can be passed to a function without having to change the function’s interface. The following sequence of function declarations outlines the procedure for creating and populating such a list:

```
integer function dbmkoptlist(    ! Create a list:
          maxopts,               ! maximum list length
          optlist_id             ! list identifier
          )

integer function dbaddiopt (      ! Add an integer option
                                  ! to the list:
          optlist_id,             ! the list
          option_id,              ! the option
          int_value               ! the option’s integer
                                  ! value
          )
```

There also are functions for adding real and character option values to a list.

2.2.6. Fortran Calling Sequence

The functions in the Silo output package should be called in a particular order. Start by creating a Silo file, with `dbcreate()`, create any necessary directories, then call the remaining routines as needed for writing out the mesh, material data, and any physics variables associated with the mesh.

Schematically, your program should look something like this:

```
dbcreate

dbmkdir
dbsetdir
     dbputqm
     dbputqv1
     dbputqv1
     dbputqv1
      . . .
dbsetdir

dbmkdir
dbsetdir
     dbputzl
     dbputfl
     dbputum
     dbputmat
     dbputuv1
 . . .
dbsetdir

dbclose
```

2.3. Reading Silo Files

Silo functions that return Silo objects from an open file return a C struct data structure defining the object. The most reliable source of information on the C structure returned from each call is the silo header file, silo.h. For reference, the header file for this version of Silo is attached as an appendix to this manual.

1 API Section Error Handling and Other Global Library Behavior

The functions described in this section of the Silo Application Programming Interface (API) manual, are those that effect behavior of the library, globally, for any file(s) that are or will be open. 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.

The functions described here are...

DBErrFunc—Get name of error-generating function

Synopsis:

```
char *DBErrFunc (void)
```

Fortran Equivalent:

```
None
```

Returns:

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

Description:

The DBErrFunc 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.

DBErrno—Get internal error number.

Synopsis:

```
int DBErrno (void)
```

Fortran Equivalent:

```
integer function dberrno()
```

Returns:

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

Description:

The DBErrno function is used to find the number of the last Silo error message. It is implemented as a macro. The error numbers are not guaranteed to remain the same between different release versions of Silo.

DBErrString—Get error message.

Synopsis:

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

Fortran Equivalent:

```
None
```

Returns:

DBErrString returns a `char*` 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.

DBShowErrors—Set the error reporting mode.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

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

Returns:

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`

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_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.

DBVariableNameValid—check if character string represents a valid Silo variable name

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`s`	The character string to check

Returns:

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.

DBVersion—Get the version of the Silo library.

Synopsis:

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

Fortran Equivalent:

```
None
```

Returns:

DBVersion returns the version as a character string.

Description:

The DBVersion function determines what version of the Silo library is being used and returns that version in string form.

DBVersionGE—Greater than or equal comparison for version of the Silo library

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`Maj`	Integer, major version number
`Min`	Integer, minor version number
`Pat`	Integer, patch version number

Returns:

One (1) if the library's version number 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.

DBFileVersion—Version of the Silo library used to create the specified file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile` Database file handle

Returns:

A character string representation of the version number of the Silo library that was used to create the Silo file.

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. The former returns the version of the Silo library that was used when DBCreate() was called on the specified file. The latter 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 not 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.

DBFileVersionGE—Greater than or equal comparison for version of the Silo library a given file was created with

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file handle
`Maj`	Integer major version number
`Min`	Integer minor version number
`Pat`	Integer patch version number

Returns:

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.

DBSetAllowOverwrites—Set flag permitting or denying overwrites of Silo objects

Synopsis:

```
int DBSetAllowOverwrites(int allow)
```

Fortran Equivalent:

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

Arguments:

`allow`	Integer value indicating if Silo library should allow overwrites to Silo objects. A value of 0 indicates that overwrites should NOT be allowed, a value of non-zero indicates that overwrites SHOULD be allowed.

Returns:

Returns the previous value set for allowing overwrites.

Description:

By default, the Silo library does not do any work to determine if the caller is somehow using the library in such a way that Silo objects are being overwritten. In fact, if a given file is open by multiple processes, it is impossible for Silo to detect this condition and either prevent it or even issue a message indicating it is happening.

When DBSetAllowOverwrites is passed a non-zero value, all succeeding calls that modify a Silo file first check to make sure the object(s) being written do not already exist in the file. If they do, the operation will abort and an error message will be generated.

Some Silo calls such as DBWriteSlice permit repeated calls to write different portions of the same object. Overwrites are always allowed for these types of objects regardless of the setting passed here.

The default setting is to allow overwrites and not do any costly work to detect when they are occuring.

DBGetAllowOverwrites—Get current setting for the allow overwrites flag

Synopsis:

```
int DBGetAllowOverwrites(void)
```

Fortran Equivalent:

```
integer function dbgetovrwrt()
```

Returns:

Returns the current setting for the allow overwrites flag

Description:

See DBSetAllowOverwrites for a description of the meaning of this flag

DBForceSingle—Force all floating point data read in read methods to be single precision

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

Zero on success. -1 on failure

Description:

This setting is global to the whole library. After a call to DBForceSingle() with a non-zero force value, all functions that read floating point data from a Silo database will convert any double-precision data they encounter to single precision (and set the associated datatype members of the DBxxx objects to DB_FLOAT). Calling DBForceSingle() with a force value of zero, will return the library to ‘normal’ behavior. That is, double-precision data will be read and returned in double-precision.

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. Write methods are NOT effected.

DBSetDataReadMask—Set the data read mask

Synopsis:

```
long DBSetDataReadMask (long mask)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBSetDataReadMask returns the previous data read mask.

Description:

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

Most Silo objects have a metadata portion and a data portion. The data portion is that part of the object that consists of pointers to long arrays of data. These arrays are "problem sized".

Setting the data read mask allows for a DBGet* call to only return part of the data. With the data read mask set to DBAll, the DBGet* 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 data model 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 together, setting all of the bit values to 1.
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 information.
DBMatMatnos	The lists of material numbers in material objects are read by the DBGetMaterial call.
DBMatMatnames	The arrays of material names in material objects are read by the DBGetMaterial call.

Mask bit	Meaning
DBMatMatlist	The lists of the correspondence between zones and material numbers in material objects are read by the DBGetMaterial call.
DBMatMixList	The lists of mixed material information in material objects are read by the DBGetMaterial call.
DBCurveArrays	The data values of curves are read by the DBGetCurve call.
DBPMCoords	The coordinate values of pointmeshes are read by the DBGetPointmesh call.
DBPVData	The data values of pointvars are read by the DBGetPointvar call.
DBQMCoords	The coordinate values of quadmeshes are read by the DBGetQuadmesh call.
DBQVData	The data values of quadvars are read by the DBGetQuadvar call.
DBUMCoords	The coordinate values of UCD meshes are read by the DBGetUcdmesh call.
DBUMFacelist	The facelists of UCD meshes are read by the DBGetUcdmesh call.
DBUMZonelist	The zonelists of UCD meshes are read by the DBGetUcdmesh call.
DBUVData	The data values of UCD variables are read by the DBGetUcdvar call.
DBFacelistInfo	The nodelists and shape information in facelists are read by the DBGetFacelist call.
DBZonelistInfo	The nodelist and shape information in zonelists are read by the DBGetZonelist call.
DBUMGlobNodeNo	The global node numbers of UCD meshes are read by the DBGetUcdmesh call
DBZonelistGlobZoneNo	The global zone numbers of UCD meshes are read by the DBGetUcdmesh call
DBMatMatcolors	The material colors of material objects are read by the DBGetMaterial and DBGetMultimat calls
DBMMADJNodelists	The multimesh adjacency nodelist is read in DBGetMultimeshadj()
DBMMADJZonelists	The multimesh adjacency zonelists is read in DBGetMultimeshadj()
DBCSGMBoundaryInfo	The boundary list is read by the DBGetCsgmesh call
DBCSGMZonelist	The zonelist is read by the DBGetCsgmesh call
DBCSGMBoundaryNames	The boundary names are read by the DBGetCsgmesh call
DBCSGVData	The data values of CSG variables are read by the DBGetCsgvar call
DBCSGZonelistZoneNames	The zone names are read by the DBGetCSGZonelist call
DBCSGZonelistRegNames	The region names are read by the DBGetCSGZonelist call

Use the DBGetDataReadMask 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.

DBGetDataReadMask—Get the current data read mask

Synopsis:

```
long DBGetDataReadMask (void)
```

Fortran Equivalent:

```
None
```

Returns:

DBGetDataReadMask returns the current data read mask.

Description:

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

See the documentation on DBSetDataReadMask for a complete description.

`DBSetEnableChecksums`—Set flag controlling checksum checks

Synopsis:

```
int DBSetEnableChecksums(int enable)
```

Fortran Equivalent:

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

Arguments:

`enable`	Integer value controlling checksum behavior of the Silo library. See description for a complete explanation.

Returns:

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 them only on reads and only if checksums were actually computed and stored with the data when it was written.

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” on page 2-11). 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 checksumed. In most applications, this represents more than 99% of all the data the client writes. However, it is important to note that when checksuming is enabled, NOT ALL data written by Silo is checksumed. Various bits of metadata is not checksumed.

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.

DBGetEnableChecksums—Get current state of flag controlling checksumming

Synopsis:

```
int DBGetEnableChecksums(void)
```

Fortran Equivalent:

```
integer function dbgetcksums()
```

Returns:

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.

DBSetCompression—Set compression options for succeeding writes of Silo data

Synopsis:n

```
int DBSetCompression(char *options)
```

Fortran Equivalent:

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

Arguments:

`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.

Returns:

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 without regard to the type of data and mesh being written. Others depend on the type of data and sometimes even the type of mesh.

Compression parameters 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 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.

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, selects entropy coding method, this is best suited for data that has been processed, working best for small numbers.MASK=NN, 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 (DBPutXXX, DBWrite, etc.) 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 is licensed for non-commercial use only while the decoder (e.g. decompression) is unlimited. Read more about SZIP licensing at http://www.hdfgroup.org/doc_resource/SZIP/index.html. 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 com-

pression 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.

DBGetCompression—Get current compression parameters

Synopsis:

```
char *DBGetCompression()
```

Fortran Equivalent:

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

Arguments:

```
None
```

Returns:

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

Description:

Obtain the current compression parameters.

DBSetFriendlyHDF5Names—Set flag to indicate Silo should create friendly names for HDF5 datasets

Synopsis:

```
int DBSetFriendlyHDF5Names(int enable)
```

Fortran Equivalent:

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

Arguments:

`enable`	Flag to indicate if friendly names should be turned on (non-zero value) or off (zero).

Returns:

Old setting for this flag

Description:

The HDF5 driver uses HDF5 in a way that makes 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 is not a problem for Silo but is a problem for these and other HDF5 tools.

DBSetFriendlyHDF5Names() is a way to address this issue so that the data in an HDF5 file written by Silo looks more "natural.". Calling DBSetFriendlyHDF5Names() with a non-zero value will result in additional HDF5 metadata being added to the file (in the form of links) with better names (and locations) for Silo objects' datasets.

Passing a value of 2 for `enable` here causes the HDF5 driver to create *hard* links. All other non-zero values cause the HDF5 driver to create *soft* links.

Finally note that creation of links does increase the file size somewhat. 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.

Notes:

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

DBGetFriendlyHDF5Names—Get setting for friendly HDF5 names flag

Synopsis:

```
int DBGetFriendlyHDF5Names()
```

Fortran Equivalent:

```
integer function dbgethdfnms()
```

Arguments:

None

Returns:

The current setting for the HDF5 friendly names flag.

Description:

See DBSetFriendlyHDF5Names().

DBSetDeprecateWarnings—Set maximum number of deprecate warnings Silo will issue for any one function, option or convention

Synopsis:

```
int DBSetDeprecateWarnings(int max_count)
```

Fortran Equivalent:

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

Arguments:

`max_count` Maximum number of warnings Silo will issue for any single API function.

Returns:

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 became 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.

DBGetDeprecateWarnings—Get maximum number of deprecated function warnings Silo will issue

Synopsis:

```
int DBGetDeprecateWarnings()
```

Fortran Equivalent:

```
integer function dbgetdepwarn()
```

Arguments:

None

Returns:

The current maximum number of deprecate warnings

Description:

DB_VERSION_GE—Compile time macro to test silo version number

Synopsis:

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

Arguments:

`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.

Returns:

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 4.5.1 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
```

2 API Section Files and File Structure

If you are looking for information regarding how to use Silo from a parallel application, please See “Multi-Block Objects, Parallelism and Poor-Man’s Parallel I/O” on page 131.

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*. 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 filesystem. 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, Parallelism and Poor-Man’s Parallel I/O” on page 131.), VisIt will use directories in a Silo file to create *submenus* within its Graphical User Interface (GUI). For example, if VisIt opens a Silo file with two directories called “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.

The functions described in this section of the interface are...

DBCreate—Create a Silo output file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`pathname`	Path name of file to create. This can be either an absolute or relative path.
`mode`	Creation mode. One of the predefined Silo modes: DB_CLOBBER or DB_NOCLOBBER.
`target`	Destination file format. One of the predefined types: DB_LOCAL, DB_SUN3, DB_SUN4, DB_SGI, DB_RS6000, or DB_CRAY.
`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, send NULL for this argument.
`filetype`	Destination file type. Specify one of ether DB_PDB, DB_HDF5, DB_HDF5_SEC2, DB_HDF5_STDIO, DB_HDF5_CORE, DB_HDF5_MPIO, or DB_HDF5_MPIPOSIX.

Returns:

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

Description:

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

Notes:

Silo supports two underlying "drivers" for storing named arrays 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). In turn, the HDF5 library also supports a number of system interfaces for doing the actual disk I/O; section 2 routines (e.g create/open/read/write/close), stdio routines (e.g. fcreate/fopen/fread/fwrite/fclose) are the most common. In HDF5 parlance, these are called Virtual File Drivers (VFDs).

Because section 2 routines are unbuffered, that 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.

The HDF5 driver for Silo also supports several of HDF5's more exotic VFDs. These are the "core" VFD which creates the entire file in memory and then writes it to disk (with minimal I/O requests) upon close as well as a couple of interfaces specialized for parallel file systems. Although Silo itself DOES NOT support true parallel I/O (e.g. multiple processors writing to the same file, concurrently), Silo can take advantage of any performance capabilities available in the underlying I/O systems calls in HDF5's parallel VFDs. These are the MPI-IO VFD which uses MPI-IO's I/O routines and the MPI-POSIX.

For the DB_HDF5_CORE filetype, it is necessary for the caller to specify the allocation increment to use each time HDF5 needs to grow the "file" in memory. This is specified in terms of kilobytes (1024 bytes) as the high-order 21 bits of the filetype argument. So, for example, to specify that HDF5 allocate space for the "file" in memory in 1 Megabyte increments, the caller would construct the filetype argument as `((1024<<11)|DB_HDF5_CORE)`. The 1024 is because we want 1024 Kilobytes (e.g. 1 Megabyte) increments. The 11 bit shift is to put the value in the high order 21 bit portion of the filetype argument.

Both PDB and HDF5 support the concept of targeting output files. That is, a Sun IEEE file can be created on the Cray, and vice versa. If creating files on a mainframe or other powerful computer, it is best to target the file for the machine where the file will be processed. Because of the extra time required to do the floating point conversions, however, one may wish to bypass the targeting function by providing DB_LOCAL as the target.

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.

DBOpen—Open an existing Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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: DB_PDB, DB_HDF5, DB_HDF5_SEC2, DB_HDF5_STDIO, DB_HDF5_MPIO, DB_HDF5_MPIPOSIX, DB_TAURUS, or DB_UNKNOWN.
`mode`	The mode of the file to open. One of the values DB_READ or DB_APPEND.

Returns:

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` is DB_UNKNOWN, Silo will guess at the file type by iterating through the known types attempting to open the file. This iteration does incur a small performance penalty. Thus, 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 from specified filename.

Indeed, in order to use a specific VFD (see “DBCreate” on page 2-35) in HDF5, it is necessary to pass the specific DB_HDF5_XXX argument in this call. If the caller wishes to support both HDF5 and PDB files and doesn’t always know ahead of time which file type will be opened, the caller can always iterate over the file types just as the DB_UNKNOWN functionality currently does.

The reader will notice that one of HDF5’s VFDs, DB_HDF5_CORE, is not supported in this call. This is because HDF5 does NOT currently support bringing a whole file into memory from disk. It supports only the creation of new files with the core VFD.

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

DBClose—Close a Silo database.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`dbfile` Database file pointer.

Returns:

DBClose returns zero on success and -1 on failure.

Description:

The DBClose function closes a Silo database.

DBGetToc—Get the table of contents of a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile` Database file pointer.

Returns:

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBMkDir—Create a new directory in a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

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

Returns:

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`".

DBSetDir—Set the current directory within the Silo database.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

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

Returns:

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.

DBGetDir—Get the name of the current directory.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBGetDir returns zero on success and -1 on failure.

Description:

The DBGetDir function returns the name of the current directory.

DBCpDir—Copy a directory hierarchy from one Silo file to another.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

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.

DBGrabDriver—Obtain the low-level driver file handle

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`file` The Silo database file handle.

Returns:

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 file id. 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 the file are prevented until the 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 re-produce the problem on a similar file that has NOT been grabbed to rule out the possible corruption from grabbing.

DBUngrabDriver—Ungrab the low-level file driver

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`file`	The Silo database file handle.
`drvr_hndl`	The low-level driver handle.

Returns:

The driver type on success, DB_UNKNOWN on failure.

Description:

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

DBGetDriverType—Get the type of driver for the specified file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`file` A Silo database file handle.

Returns:

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

DBGetDriverTypeFromPath—Guess the driver type used by a file with the given pathname

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`path` Path to a file on the filesystem

Returns:

DB_UNKNOWN on failure to determine type. Otherwise, the driver type (e.g. DB_PDB, DB_HDF5)

Notes:

As currently implemented, it is not possible for this method to return a driver type the library has not been compiled with.

DBInqFile—Inquire if `filename` is a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`filename` Name of file.

Returns:

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.

_silolibinfo—character array written by Silo to root directory indicating the Silo library version number used to generate the file

Synopsis:

```
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.

_hdf5libinfo—character array written by Silo to root directory indicating the HDF5 library version number used to generate the file

Synopsis:

```
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.

`_was_grabbed`—single integer written by Silo to root directory whenever a Silo file has been grabbed.

Synopsis:

```
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.

3 API Section Meshes, Variables and Materials

If you are interested in learning how to deal with these objects in parallel, See “Multi-Block Objects, Parallelism and Poor-Man’s Parallel I/O” on page 131.

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.

The functions described in this section of the manual include...

DBPutCurve—Write a curve object into a Silo file

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

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.

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_LABEL	int	Problem cycle value.	0
DBOPT_XLABEL	char *	Label for the x-axis	NULL
DBOPT_YLABEL	char *	Label for the y-axis	NULL

Option Name	Value Data Type	Option Meaning	Default Value
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

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.

DBGetCurve—Read a curve from a Silo database.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`dbfile`	Database file pointer.
`curvename`	Name of the curve to read.

Returns:

DBCurve 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutPointmesh—Write a point mesh object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
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:

`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.

Returns:

DBPutPointmesh 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 mesh. This includes the coordinate arrays, the number of dimensions (1,2,3,...) and the number of points.

Notes:

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.

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_XLABEL	char *	Character string defining the label associated with the X dimension.	NULL

Option Name	Value Data Type	Option Meaning	Default Value
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_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_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
The following optlist options have been deprecated. Instead use MRG trees			
DBOPT_GROUPNUM	int	The group number to which this point-mesh belongs.	-1 (not in a group)

DBGetPointmesh—Read a point mesh from a Silo database.

Synopsis:

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

Arguments:

`dbfile`	Database file pointer.
`meshname`	Name of the mesh.

Returns:

DBGetPointmesh 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutPointvar—Write a vector/tensor point variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
None. See DBPutPointvar1
```

Arguments:

`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.

Returns:

DBPutPointvar 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.

Notes:

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.

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
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_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” on page 188.	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

DBPutPointvar1—Write a scalar point variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

DBPutPointvar1 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.

Notes:

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.

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_CYCLE	int	Problem cycle value.	0
DBOPT_NSPACE	int	Number of spatial dimensions used by this mesh.	ndims
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_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_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

DBGetPointvar—Read a point variable from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the variable.

Returns:

DBGetPointvar 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutQuadmesh—Write a quad mesh object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
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:

`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.
`ndims`	Number of dimensions.
`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.

Returns:

DBPutQuadmesh 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.)

Notes:

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.

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	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	double	Problem time value.	0.0
DBOPT_XUNITS	char *	Character string defining the units associated with the X dimension.	NULL

Option Name	Value Data Type	Option Meaning	Default Value
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
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}.

DBGetQuadmesh—Read a quadrilateral mesh from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`meshname`	Name of the mesh.

Returns:

DBGetQuadmesh 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutQuadvar—Write a vector/tensor quad variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
None (see DBPutQuadvar1)
```

Arguments:

`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.

Returns:

DBPutQuadvar 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 quadvar 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.

Notes:

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.

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	Zone 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 or DB_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

Option Name	Value Data Type	Option Meaning	Default Value
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” on page 188.	NULL

DBPutQuadvar1— Write a scalar quad variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

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.

Returns:

DBPutQuadvar1 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:

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.

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	Zone 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 or DB_ON) value specifying whether or not to weight the variable by the species mass fraction when using material species data.	DB_OFF
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

Option Name	Value Data Type	Option Meaning	Default Value
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

DBGetQuadvar—Read a quadrilateral variable from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the variable.

Returns:

DBGetQuadvar 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutUcdmesh—Write a UCD mesh object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

`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.
`zonel_name`	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.
`facel_name`	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.

Returns:

DBPutUcdmesh 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.

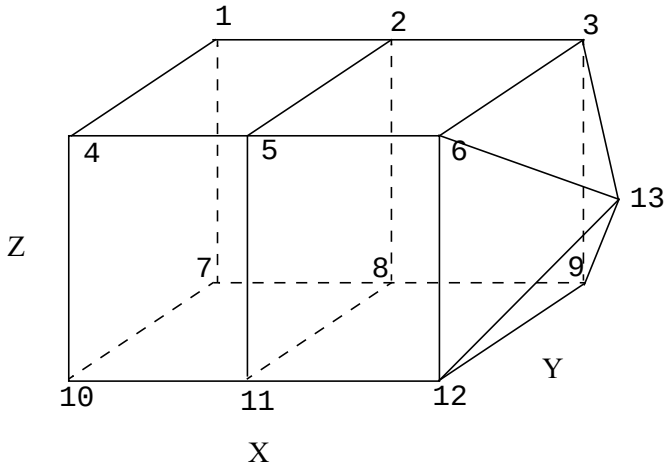
A Silo UCD mesh object contains all necessary information for describing a mesh. This includes the coordinate arrays, the rank of the mesh (1,2,3,...) and the type (collinear or non-collinear.) In addition, other information is useful and is therefore included (time and cycle of mesh, plus coordinate 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 the description of “DBCalcExternalFacelist” on page 2-194 or “DBCalcExternalFacelist2” on page 2-196 for an automated way of computing the facelist needed for this call.




```
nnodes     = 13
nzones     = 3
nzshapes   = 2
lznodelist = 2*8 + 1*5 = 21 zone nodes
nfaces     = 13 external faces
nfshapes   = 2 external face shapes
nftypes    = 0
lfnodelist = 9*4 + 4*3 = 48 external face nodes

fnodelist = { 1,2,8,7 external face nodelist
              2,3,9,8,
              8,9,12,11,
              5,6,12,11,...}

fshapesize = {4,3} external face shape sizes
fshapecnt = {9,4} external face shape counts
fzoneno    = {1,2,2,2,...}external face zone nos

znodelist = { 7,10,11,8,1,4,5,2, zone nodelist
              8,11,12,9,2,5,6,3,
              3,9,12,6,13}
zshapesize = {8,5} zone shape sizes
zshapecnt = {2,1} zone shape counts
x = {0,1,2,0,1,2,0,1,2,0,1,2,3}
y = {1,1,1,0,0,0,1,1,1,0,0,0,.5}
z = {1,1,1,1,1,1,0,0,0,0,0,0,.5}
```

Figure 0-1: Example usage of UCD zonelist and external facelist variables.

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

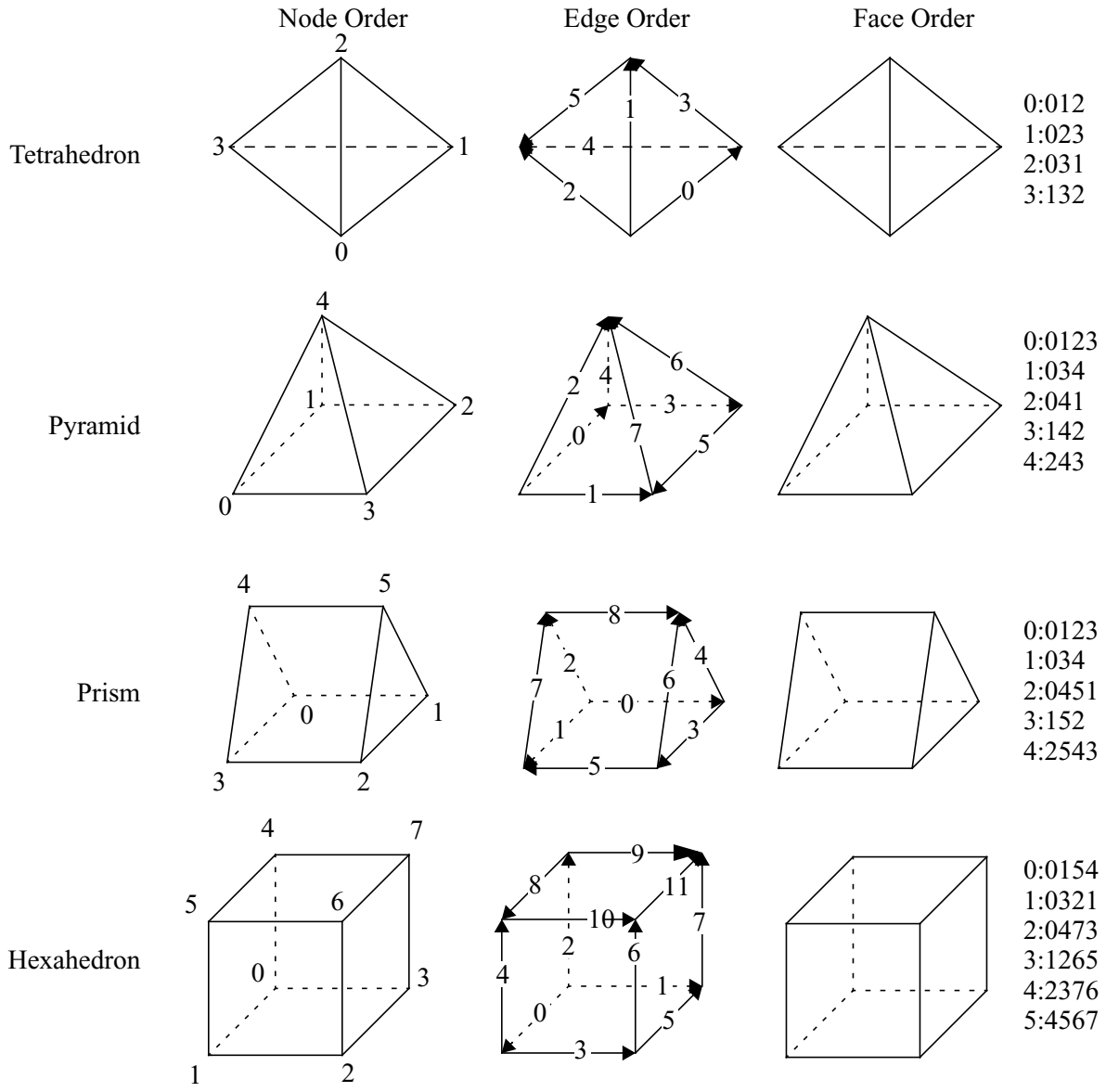


Figure 0-2: Node, edge and face ordering for zoo-type UCD zone shapes.

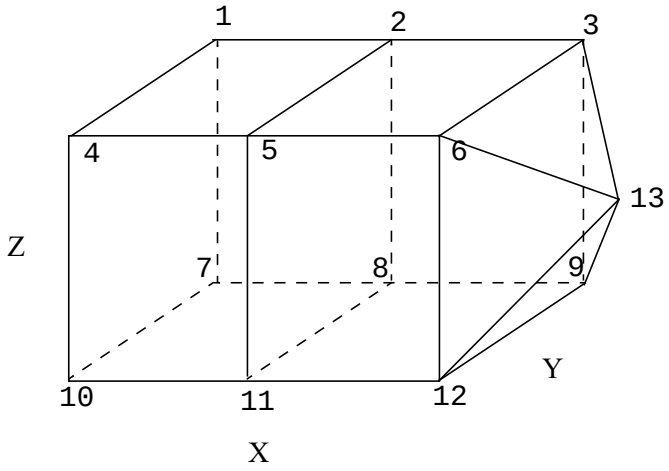
Given the node ordering in the left-most column, there is indeed an algorithm for determining the other orderings for each cell type.

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 in Figure 0-3 on page 82. 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. 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 82, 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.




```
nzones     = 3
nzshapes   = 2
lznodelist = 8 + 1 + 6 * 5 + 1 + 5 + 4 * 4 = 61
znodelist = {7,10,11,8,1,4,5,2,
             6,
             4,11,12,9,8,
             4,12,6,3,9,
             4,6,5,2,3,
             4,5,11,8,2,
             4,5,6,12,11,
             4,3,2,8,9,
             5,
             4,3,6,12,9,
             3,6,13,12,
             3,12,13,9,
             3,9,13,3,
             3,3,13,6}
zshapetype = {DB_ZONETYPE_HEX,
              DB_ZONETYPE_POLYHEDRON}
zshapecnt = {1, 2}
zshapesize = {8, 53}
```

Figure 0-3: 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:

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_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

Option Name	Value Data Type	Option Meaning	Default Value
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_CONNECTIVTY	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
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)

DBPutUcdsubmesh—Write a subset of a parent, ucd mesh, to a Silo file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

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” on page 165.

DBGetUcdmesh—Read a UCD mesh from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`meshname`	Name of the mesh.

Returns:

DBGetUcdmesh returns 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutZonelist—Write a zonelist object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

DBPutZonelist 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.

Notes:

See the write-up of DBPutUcdmesh for a full description of the zonelist data structures.

DBPutZonelist2—Write a zonelist object containing ghost zones into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

DBPutZonelist2 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:

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	Value 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_EDGELIST	char*	Name of explicit edgelist object	NULL

For a description of how the nodes for the allowed shapes are enumerated, see “DBPutUcdmesh” on page 2-77

DBPutPHZonelist—Write an arbitrary, polyhedral zonelist object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBPutPHZonelist returns zero on success or -1 on failure.

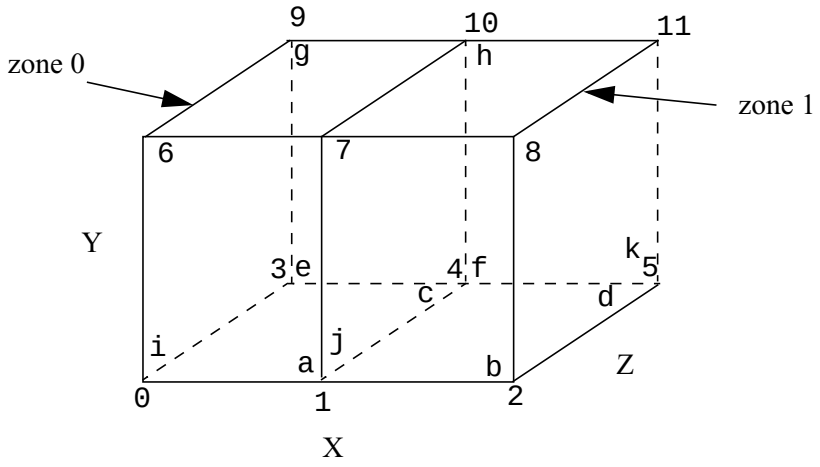
Description:

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.

Notes:

The following table describes the options accepted by this function:

Option Name	Value 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



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.

```
#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};
```

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

DBGetPHZonelist—Read a polyhedral-zonelist from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`phzlname`	Name of the polyhedral-zonelist.

Returns:

DBGetPHZonelist 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutFacelist—Write a facelist object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

DBPutFacelist 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.

Notes:

See the write-up of DBPutUcdmesh for a full description of the facelist data structures.

DBPutUcdvar—Write a vector/tensor UCD variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

None

Arguments:

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.

Returns:

DBPutUcdvar 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 `DBPut-Facelist` 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 0-2 on page 80. 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 DBPutPHZonelist), 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.

Notes:

The following table describes the options accepted by this function:

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

Option Name	Value Data Type	Option Meaning	Default Value
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 or DB_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 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” on page 188.	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

DBPutUcdvar1—Write a scalar UCD variable object into a Silo file.

Synopsis:

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

Fortran Equivalent:

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

Arguments:

`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.

Returns:

DBPutUcdvar1 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.

Notes:

The following table describes the options accepted by this function:

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 or DB_ON) value specifying whether or not to weight the variable by the species mass fraction when using material species data.	DB_OFF
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_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

DBGetUcdvar—Read a UCD variable from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the variable.

Returns:

DBGetUcdvar 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutCsgmesh—Write a CSG mesh object to a Silo file

Synopsis:

```
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 *optlist);
```

Fortran Equivalent:

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

Arguments:

`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 is `DB_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.

Returns:

DBPutCsgMesh 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.

typeflag	num-coeffs	coefficients and equation
DBCSG_QUADRIC_G	10	$a_0x^2 + a_1y^2 + a_2z^2 + a_3xy + a_4yz + a_5xz + a_6x + a_7y + a_8z + a_9 = 0$
DBCSG_SPHERE_PR	4	$(x - a_0)^2 + (y - a_1)^2 + (z - a_2)^2 - a_3^2 = 0$
DBCSG_ELLIPSOID_PRRR	6	$(x - a_0)^2/a_3^2 + (y - a_1)^2/a_4^2 + (z - a_2)^2/a_5^2 - 1 = 0$
DBCSG_PLANE_G	4	$a_0x + a_1y + a_2z + a_3 = 0$
DBCSG_PLANE_X	1	$x - a_0 = 0$
DBCSG_PLANE_Y	1	$y - a_0 = 0$
DBCSG_PLANE_Z	1	$z - a_0 = 0$
DBCSG_PLANE_PN	6	$(x - a_0)a_3 + (y - a_1)a_4 + (z - a_2)a_5 = 0$
DBCSG_PLANE_PPP	9	$\begin{vmatrix} x - a_0 & y - a_1 & z - a_2 \\ a_3 - a_0 & a_4 - a_1 & a_5 - a_2 \\ a_6 - a_0 & a_7 - a_1 & a_8 - a_2 \end{vmatrix} = 0$
DBCSG_CYLINDER_PNLR	8	to be completed

typeflag	num-coeffs	coefficients and equation
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	to be completed
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	$a_0x^2 + a_1y^2 + a_2xy + a_3x + a_4y + a_5 = 0$
DBCSG_CIRCLE_PR	3	$(x - a_0)^2 + (y - a_1)^2 - a_2{}^2 = 0$
DBCSG_ELLIPSE_PRR	4	$(x - a_0)^2 / a_2^2 + (y - a_1)^2 / a_3^2 - 1 = 0$
DBCSG_LINE_G	3	$a_0x + a_1y + a_2 = 0$
DBCSG_LINE_X	1	$x - a_0 = 0$
DBCSG_LINE_Y	1	$y - a_0 = 0$
DBCSG_LINE_PN	4	$(x - a_0)a_2 + (y - a_1)a_3 = 0$
DBCSG_LINE_PP	4	$\frac{a_3 - a_1}{a_2 - a_0} - \frac{y - a_1}{x - a_0} = 0$
DBCSG_BOX_XYXY	4	to be completed
DBCSG_POLYGON_KP | K	2K	to be completed
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, `DBPutCSG-Zonelist`.

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	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_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_CONNECTIVTY	int	A non-zero value indicates that the connectivity of the mesh varies with time	0

Option Name	Value Data Type	Option Meaning	Default Value
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
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)

DBGetCsgmesh—Get a CSG mesh object from a Silo file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer
`meshname`	Name of the CSG mesh object to read

Returns:

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

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutCSGZonelist—Put a CSG zonelist object in a Silo file.

Synopsis:

```
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 Equivalent:

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

Arguments:

`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 of `DB_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.

Returns:

`DBPutCSGZonelist` 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.

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

op. symbol name	type	meaning
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	Value 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

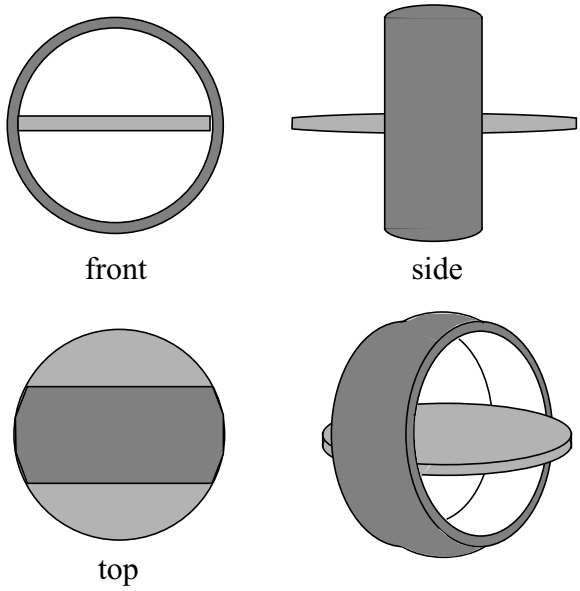


Figure 0-5: A relatively simple object to represent as a CSG mesh. It models an A/C vent outlet for a 1994 Toyota Tercel. It consists of two zones. One is a partially-spherical shaped ring housing (darker area). The other is a lens-shaped fin used to direct airflow (lighter area).

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	leftids	rightids	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);
```

DBGetCSGZonelist—Read a CSG mesh zonelist from a Silo file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer
`zlname`	Name of the CSG mesh zonelist object to read

Returns:

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

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutCsgvar—Write a CSG mesh variable to a Silo file

Synopsis:

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

Fortran Equivalent:

```
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:

`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` or `DB_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:

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 or DB_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 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” on page 188.	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

DBGetCsgvar—Read a CSG mesh variable from a Silo file

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer
`varname`	Name of CSG variable object to read

Returns:

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

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMaterial—Write a material data object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
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:

`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 non-NULL.
`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.

Returns:

DBPutMaterial 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" on page 165.

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.

Notes:

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.

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

Option Name	Value Data Type	Option Meaning	Default Value
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

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.

.

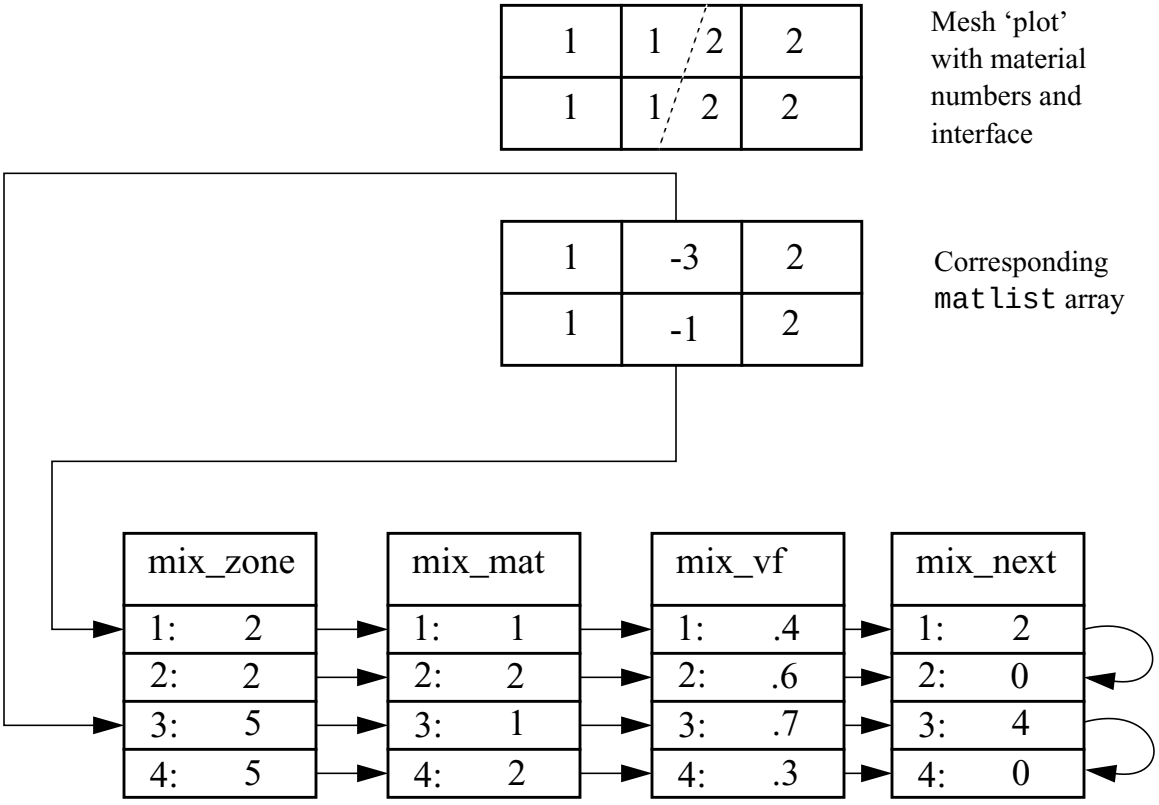


Figure 0-6: Example using mixed data arrays for representing material information

DBGetMaterial—Read material data from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`mat_name`	Name of the material variable to read.

Returns:

DBGetMaterial 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMatspecies—Write a material species data object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
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:

`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.
`ndims`	Number of dimensions in the `speclist` array.
`nspecies_mf`	Length of the `species_mf` array.
`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.

Returns:

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* (99.962% 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 (2 values)		T-1 Steel (5 values starting at offset 3)					O-1 Steel (6 values starting at offset 8)					
species_mf	.65	.35	.763	.007	.18	.04	.001	.99962	.009	.014	.005	.005	.005
element	Cu	Zn	Fe	C	W	Cr	V	Fe	C	Mn	Cr	Ni	W
1-origin 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.

Notes:

The following table describes the options accepted by this function:

Option Name	Value Data Type	Option Meaning	Default Value
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_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. 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

DBGetMatspecies—Read material species data from a Silo database.

Synopsis:

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

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`ms_name`	Name of the material species data to read.

Returns:

DBGetMatspecies 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutDefvars—Write a derived variable definition(s) object into a Silo file.

Synopsis:

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

Fortran Equivalent:

```
integer function dbputdefvars(dbid, name, lname, ndefs, names,
                   lnames, types, defns, ldefns, optlist_id,
                   status)
character*N names (See “dbset2dstrlen” on page 248.)
character*N defns (See “dbset2dstrlen” on page 248.)
```

Arguments:

`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 such as `DB_VARTYPE_SCALAR, DB_VARTYPE_VECTOR, DB_VARTYPE_TENSOR, DB_VARTYPE_SYMTENSOR, DB_VARTYPE_ARRAY, DB_VARTYPE_MATERIAL, DB_VARTYPE_SPECIES, DB_VARTYPE_LABEL`
`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.

Returns:

DBPutDefvars 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
0	“totaltemp”	DB_VARTYPE_SCALAR	“nodet+zonetemp”
1	“<stress/sz>”	DB_VARTYPE_SCALAR	“-<stress/sx>-<stress/sy>”
2	“vel”	DB_VARTYPE_VECTOR	“{Vx, Vy, Vz}”
3	“speed”	DB_VARTYPE_SCALAR	“magntidue(vel)”
4	“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.

DBGetDefvars—Get a derived variables definition object from a Silo file.

Synopsis:

```
DBdefvars DBGetDefvars(DBfile *dbfile, const char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`name`	The name of the DBdefvars object to read

Returns:

DBGetDefvars 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBInqMeshname—Inquire the mesh name associated with a variable.

Synopsis:

```
int DBInqMeshname (DBfile *dbfile, char *varname, char *meshname)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

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.

DBInqMeshtype—Inquire the mesh type of a mesh.

Synopsis:

```
int DBInqMeshtype (DBfile *dbfile, char *meshname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`meshname`	Mesh name.

Returns:

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

4 API Section Multi-Block Objects, Parallelism and Poor-Man's 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. All characters after a colon specify the directory path *within* the 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 affectionately called *Poor Man's Parallel I/O (PMPIO)*.

A separate convenience interface, PMPIO, is provided for this purpose. The PMPIO interface provides almost all of the functionality necessary to use Silo in a Poor Man's Parallel way. 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...

DBPutMultimesh—Write a multi-block mesh object into a Silo file.

Synopsis:

```
int DBPutMultimesh (DBfile *dbfile, char *name, int nmesh,
                    char *meshnames[],int meshtypes[],
                    DBoptlist *optlist)
```

Fortran Equivalent:

```
integer function dbputmmesh(dbid, name, lname, nmesh, meshnames,
                     lmeshnames, meshtypes, optlist_id, status)
character*N meshnames (See “dbset2dstrlen” on page 248.)
```

Arguments:

`dbfile` Database file pointer.

`name` Name of the multi-block mesh structure.

`nmesh` Number of meshes provided.

`meshnames` Array of length `nmesh` containing pointers to the names of each of the mesh blocks written with DBPutPointmesh(), DBPutQuadmesh(), DBPutUcdmesh, DBPutCsgmesh(). Ordinarily, the meshes are stored in different sub-directories within a Silo file and, optionally, even in different Silo files altogether. So, the name of each mesh is specified using its *full Silo path* name. The full Silo pathname is the form...

`[<silo-filename>:]<path-to-mesh>`

The existence of a colon (‘:’) anywhere in the `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 filesystem on which the file(s) reside. Use whatever slash character (‘\’ for Windows or ‘/’ for Unix) is appropriate for the underlying filesystem *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 filesystem. 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.

Finally, 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.

`meshtypes` Array of length `nmesh` containing the type of each mesh. One of the predefined types such as `DB_QUAD_RECT`, `DB_QUAD_CURV`, `DB_UCDMESH`,

DB_POINTMESH, and DB_CSGMESH.

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.

Returns:

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 descriptions of the various sub-meshes (blocks) which are part of this mesh.

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.

Note also that what is described here for the mulitmesh object in the way of names of objects in different files applies as well to all other multi-block objects.

Notes:

The following table describes the options accepted by this function:

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[a]	int	Number of values in each extent tuple	0

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_EXTENTS[a]	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[a]	int*	Pointer to an array of length `nmesh` indicating the number of zones in each block.	NULL
DBOPT_HAS_EXTERNAL_ZONES[a]	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_CONNECTIVTY	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)
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 multi-mesh object.	0
DBOPT_ADJACENCY_NAME[a]	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

Option Name	Value Data Type	Option Meaning	Default Value
DBOPT_GROUPINGS	int *	Integer array of length specified by DBOPT_GROUPINGS_SIZE containing information on how different mesh blocks are organized into, possibly hierarchical, groups. See below for detailed discussion.	NULL
DBOPT_GROUPINGS_NAMES	char **	Optional set of names to be associated with each group in the groupings array	NULL

a. Indicates a *Down-stream Performance Option*. See notes below.

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.

DBGetMultimesh—Read a multi-block mesh from a Silo database.

Synopsis:

```
DBmultimesh *DBGetMultimesh (DBfile *dbfile, char *meshname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`meshname`	Name of the multi-block mesh.

Returns:

DBGetMultimesh 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMultimeshadj—Write some or all of a multi-mesh adjacency object into a Silo file.

Synopsis:

```
int DBPutMultimeshadj(DBfile *dbfile, const char *name,
        int nmesh, const int *mesh_types, const int *nneighbors,
        const int *neighbors, const int *back,
        const int *nnodes, const int *nodelists[],
        const int *nzones, const int *zonelists[],
        DBoptlist *optlist)
```

Fortran Equivalent:

```
None
```

Arguments:

`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 $S_k = \sum_{i=0}^{k} \texttt{nneighbors[i]}$. That is, let S_k be the sum of the first `k` entries in the `nneighbors` array.
`neighbors`	Array of S_{nmesh} integers enumerating for each mesh piece all other mesh pieces that neighbor it. Entries from index S_k to index $S_{k+1} - 1$ enumerate the neighbors of mesh piece `k`. This array must be identical in repeated calls to DBPutMultimeshadj.
`back`	Array of S_{nmesh} integers enumerating for each mesh piece, the local index of that mesh piece in each of its neighbors lists of neighbors. Entries from index S_k to index $S_{k+1} - 1$ 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 S_{nmesh} integers indicating for each mesh piece, the number of nodes that it *shares* with each of its neighbors. Entries from index S_k to index $S_{k+1} - 1$ 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 s_{nmesh} pointers to arrays of integers. Entries from index s_k to index $s_{k+1}-1$ 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 s_{nmesh} integers indicating for each mesh piece, the number of zones that are *adjacent* with each of its neighbors. Entries from index s_k to index $s_{k+1}-1$ 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 s_{nmesh} pointers to arrays of integers. Entries from index s_k to index $s_{k+1}-1$ 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 that the functionality this object provides is now more efficiently and conveniently handled via a Mesh Region Grouping (MRG) tree. Users are encouraged to use MRG trees as an alternative to DBPutMultimeshadj(). See “DBMakeMrgtree” on page 165.

DBPutMultimeshadj is another *Down-stream Performance Option* (See “DBPutMultimesh” on page 2-132). It is an alternative to including *ghost-zones* (See “DBPutMultimesh” on page 2-132) in the mesh and can therefore 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 pieces 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 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.

		Neighbor mesh type	
		DB_POINT or DB_UCD	DB_QUAD
Given mesh type	DB_POINT or DB_UCD	nnodes[m] pairs of integers	nnodes[m]+6 integers. The first nnodes[m] integers identify the nodes in the given point or ucd mesh. The next 6 integers identify ijk bounds of the corresponding nodes in the quad mesh neighbor.
	DB_QUAD	6+nnodes[m] integers. The first 6 integers identify ijk bounds of the nodes in the given quad mesh. The last nnodes[m] integers identify the nodes in the neighbor point or ucd mesh.	15 integers The first set of 6 integers identify ijk bounds of nodes in the given quad mesh. The second set of 6 integers identify ijk bounds of nodes in the neighbor quad mesh The next 3 integers specify the orientation of the neighbor quad mesh relative to the given mesh.

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.

DBGetMultimeshadj—Get some or all of a multi-mesh nodal adjacency object

Synopsis:

```
DBmultimeshadj *DBGetMultimeshadj(DBfile *dbfile,
                    const char *name,
                    int nmesh, const int *mesh_pieces)
```

Fortran Equivalent:

None

Arguments:

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.

Returns:

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMultivar—Write a multi-block variable object into a Silo file.

Synopsis:

```
int DBPutMultivar (DBfile *dbfile, char *name, int nvar,
                   char *varnames[], int vartypes[],
                   DBoptlist *optlist);
```

Fortran Equivalent:

```
integer function dbputmvar(dbid, name, lname, nvar, varnames,
                   lvarnames, vartypes, optlist_id, status)
character*N varnames (See “dbset2dstrlen” on page 248.)
```

Arguments:

`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. These are variables written with DBPutPointvar, DBPutQuadvar, and DBPutUcdvar. Ordinarily, the variables are stored in different sub-directories within a Silo file and, optionally, even in different Silo files altogether. So, the name of each block variable is specified using its *full Silo path* name. The full Silo pathname is the form...

```
[<silo-filename>:]<path-to-mesh>
```

The existence of a colon (‘:’) anywhere in the `meshnames[i]` indicates that the ith block variable 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 filesystem on which the file(s) reside. Use whatever slash character (‘\’ for Windows or ‘/’ for Unix) is appropriate for the underlying filesystem *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 filesystem. 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 variable 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.

Finally, the individual variable names referenced here CANNOT be the names of other multi-block variables. In other words, it is not valid to create a multi-var that references other multi-vars.

`vartypes` Array of length `nvar` containing the types of the variables. Each entry must be one of the following: DB_POINTVAR, DB_QUADVAR, or DB_UCDVAR.

`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.

Returns:

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:

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[a]	int	Number of values in each extent tuple	0
DBOPT_EXTENTS[a]	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_ARRAY DB_VARTYPE_LABEL	DB_VARTYPE_SCALAR

Option Name	Value Data Type	Option Meaning	Default Value
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" on page 188.	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
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 multi-mesh object.	0

a. Indicates a *Down-stream Performance Option*. See notes for DBPutMultimesh.

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.

DBGetMultivar—Read a multi-block variable definition from a Silo database.

Synopsis:

```
DBmultivar *DBGetMultivar (DBfile *dbfile, char *varname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the multi-block variable.

Returns:

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMultimat—Write a multi-block material object into a Silo file.

Synopsis:

```
int DBPutMultimat (DBfile *dbfile, char *name, int nmat,
                   char *matnames[], DBoptlist *optlist)
```

Fortran Equivalent:

```
integer function dbputmmat(dbid, name, lname, nmat, matnames,
                   lmatnames, optlist_id, status)
```

Arguments:

`dbfile` Database file pointer.

`name` Name of the multi-material object.

`nmat` Number of materials provided.

`matnames` Array of length `nmat` containing pointers to the names of the material block objects, written with DBPutMaterial(), to be associated with the multi-material object. Ordinarily, the material objects are stored in different sub-directories within a Silo file and, optionally, even in different Silo files altogether. So, the name of each material object is specified using its full Silo path name. The full Silo pathname is the form...

```
[<silo-filename>:]<path-to-mesh>
```

The existence of a colon (‘:’) anywhere in the meshnames[i] indicates that the ith material object 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 filesystem on which the file(s) reside. Use whatever slash character (‘\’ for Windows or ‘/’ for Unix) is appropriate for the underlying filesystem *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 filesystem. 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 material 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.

Finally, the individual material object names referenced here CANNOT be the names of other multi-block materials.

`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

Returns:

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:

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[a]	int*	Array of nmat ints which are the values of the mixlen arguments in each of the individual block's material objects.	
DBOPT_MATCOUNTS[a]	int*	Array of nmat counts indicating the number of materials actually in each block.	NULL
DBOPT_MATLISTS[a]	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

Option Name	Value Data Type	Option Meaning	Default Value
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
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

a. Indicates a *Down-stream Performance Option*. See notes for DBPutMultimesh.

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.

DBGetMultimat—Read a multi-block material object from a Silo database

Synopsis:

```
DBmultimat *DBGetMultimat (DBfile *dbfile, char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer
`name`	Name of the multi-block material object

Returns:

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutMultimatspecies—Write a multi-block species object into a Silo file.

Synopsis:

```
int DBPutMultimatspecies (DBfile *dbfile, char *name, int nspec,
                          char *specnames[], DBoptlist *optlist)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.
`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.

Returns:

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:

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

Option Name	Value Data Type	Option Meaning	Default Value
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
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

DBGetMultimatspecies—Read a multi-block species from a Silo database.

Synopsis:

```
DBmultimesh *DBGetMultimatspecies (DBfile *dbfile, char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`name`	Name of the multi-block material species.

Returns:

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

PMPIO_Init—Initialize a Poor Man's Parallel I/O interaction with the Silo library

Synopsis:

```
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 Equivalent:

```
None
```

Arguments:

`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 cause 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.

Returns:

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 Poor Man's 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 filesystem. 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 filesystem 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, ...);

/* 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.

Poor Man's 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 Poor Man's 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.

PMPIO_CreateFileCallBack—The PMPIO file creation callback

Synopsis:

```
typedef void *(*PMPIO_CreateFileCallBack)(const char *fname,
    const char *dname, void *udata);
```

Fortran Equivalent:

```
None
```

Arguments:

`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()`.

Returns:

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.

PMPIO_OpenFileCallBack—The PMPIO file open callback

Synopsis:

```
typedef void *(*PMPIO_OpenFileCallBack)(const char *fname,
    const char *dname, PMPIO_iomode_t iomode, void *udata);
```

Fortran Equivalent:

```
None
```

Arguments:

`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()`.

Returns:

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.

`PMPIO_CloseFileCallBack`—The PMPIO file close callback

Synopsis:

```
typedef void  (*PMPIO_CloseFileCallBack)(void *file, void *udata);
```

Fortran Equivalent:

```
None
```

Arguments:

`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()`.

Returns:

None

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.

PMPIO_WaitForBaton—Wait for exclusive access to a Silo file

Synopsis:

```
void *PMPIO_WaitForBaton(PMPIO_baton_t *bat,
    const char *filename, const char *dirname)
```

Fortran Equivalent:

```
None
```

Arguments:

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.

Returns:

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” on page 162.) 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” on page 163.) of the processor.

PMPIO_HandOffBaton—Give up all access to a Silo file

Synopsis:

```
void PMPIO_HandOffBaton(const PMPIO_baton_t *bat, void *file)
```

Fortran Equivalent:

```
None
```

Arguments:

`bat`	The PMPIO baton handle obtained via a call to PMPIO_Init().
`file`	A void pointer to the Silo DBfile object.

Returns:

None

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.

PMPIO_Finish—Finish a Poor Man's Parallel I/O interaction with the Silo library

Synopsis:

```
void PMPIO_Finish(PMPIO_baton *bat)
```

Fortran Equivalent:

```
None
```

Arguments:

`bat`	The PMPIO baton handle obtained via a call to PMPIO_Init().

Returns:

None.

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.

PMPIO_GroupRank—Obtain ‘group rank’ of the calling processor

Synopsis:

```
int PMPIO_GroupRank(const PMPIO_baton_t *bat, int rankInComm)
```

Fortran Equivalent:

```
None
```

Arguments:

`bat`	The PMPIO baton handle obtained via a call to PMPIO_Init().
`rankInComm`	Rank of calling processor in the MPI communicator passed in `PMPIO_Init()`.

Returns:

The ‘group rank’ of the calling processor. In other words, the group number of the calling processor, indexed from zero.

Description:

This is a convenience function to help applications identify which PMPIO group a given processor belongs to.

PMPIO_RankInGroup—Obtain the rank of the calling processor within its PMPIO group

Synopsis:

```
int PMPIO_RankInGroup(const PMPIO_baton_t *bat, int rankInComm)
```

Fortran Equivalent:

```
None
```

Arguments:

`bat`	The PMPIO baton handle obtained via a call to PMPIO_Init().
`rankInComm`	Rank of the calling processor in the MPI communicator used in `PMPIO_Init()`.

Returns:

The rank of the calling 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.

5 API Section 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.

The functions described in this section of the API manual are...

DBMakeMrgtree—Create and initialize an empty mesh region grouping tree

Synopsis:

```
DBmrgtree *DBMakeMrgtree(int mesh_type, int info_bits,
    int max_children, DBoptlist *opts)
```

Fortran Equivalent:

```
integer function dbmkmrgtree(mesh_type, info_bits, max_children,
                  optlist_id, tree_id)
```

Arguments:

`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

Returns:

A pointer to a new DBmrgtree object on success and NULL on failure

Description:

This function creates a *Mesh Region Grouping Tree* (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 165.

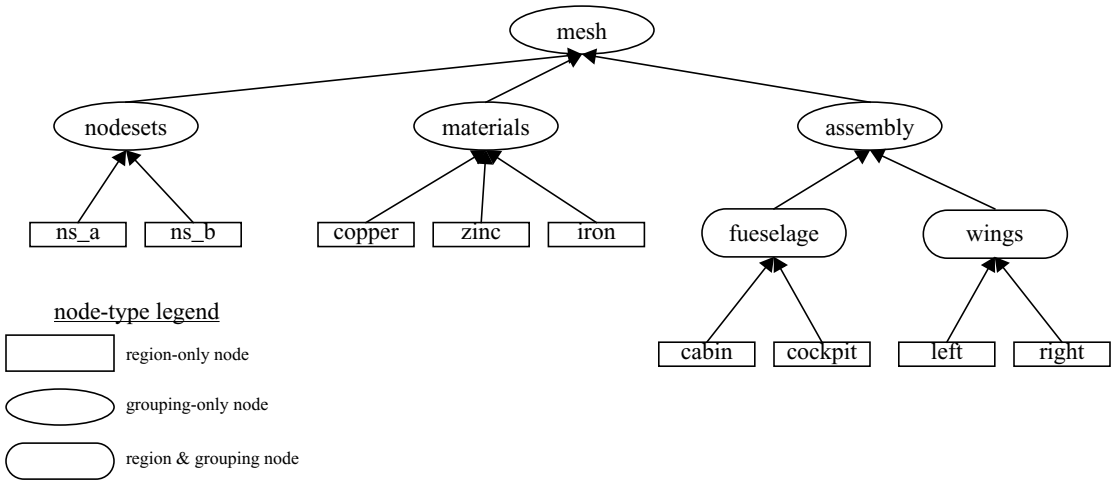


Figure 0-8: Example of MRGTree

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, _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.

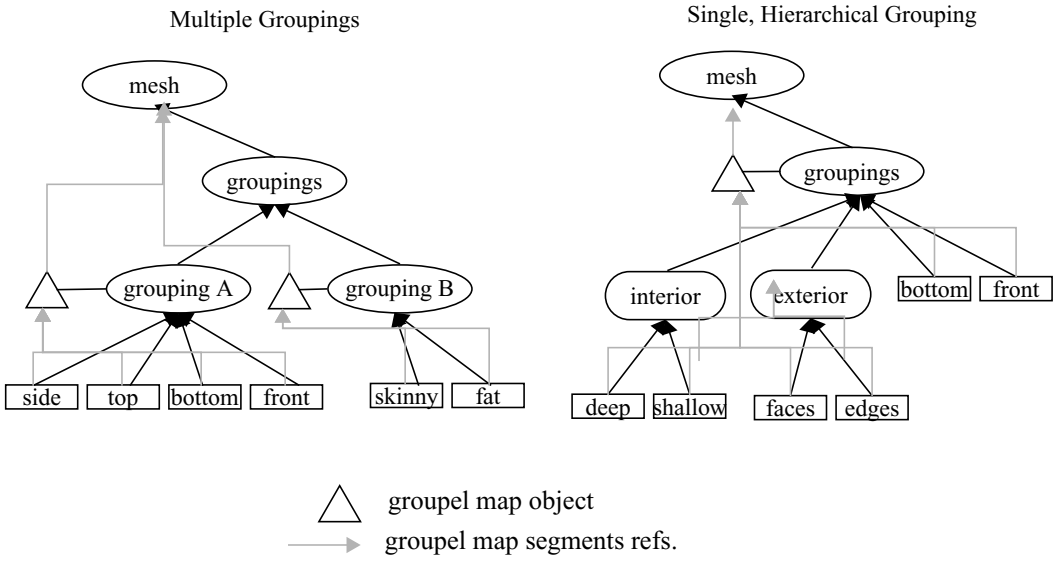


Figure 0-9: Examples of MRG trees for single and multiple groupings.

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.

DBAddRegion—Add a region to an MRG tree

Synopsis:

```
int DBAddRegion(DBmrgtree *tree, const char *reg_name,
    int info_bits, int max_children, const char *maps_name,
    int nsegs, int *seg_ids, int *seg_lens, int *seg_types,
    DBoptlist *opts)
```

Fortran Equivalent:

```
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:

`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.

Returns:

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” on page 173.).

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_<name>” 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_<name>” underneath the “groupings” region node.
“amr-levels”	Top-level region below which Adaptive Mesh Refinement *level* groupings are defined.
“amr-refinements”	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.

DBAddRegionArray—Efficiently add multiple, like-kind regions to an MRG tree

Synopsis:

```
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)
```

Fortran Equivalent:

```
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:

`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.

Returns:

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” on page 178.)

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” on page 12.) except that MRG region names can in no circumstance contain either a semi-colon character (‘;’) or a new-line character (‘\n’).

DBSetCwr—Set the current working region for an MRG tree

Synopsis:

```
int DBSetCwr(DBmrgtree *tree, const char *path)
```

Fortran Equivalent:

```
integer function dbsetcwr(tree, path, lpath)
```

Arguments:

`tree`	The MRG tree object.
`path`	The path to set.

Returns:

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 filesystem.

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 the next release.

DBGetCwr—Get the current working region of an MRG tree

Synopsis:

```
const char *GetCwr(DBmrgtree *tree)
```

Arguments:

`tree` The MRG tree.

Returns:

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.

Description:

DBPutMrgtree—Write a completed MRG tree object to a Silo file

Synopsis:

```
int DBPutMrgtree(DBfile *file, const char *name,
    const char *mesh_name, DBmrgtree *tree, DBoptlist *opts)
```

Fortran Equivalent:

```
int dbputmrgtree(dbid, name, lname, mesh_name, lmesh_name,
                  tree_id, optlist_id, status)
```

Arguments:

`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.

Returns:

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().

DBGetMrgtree—Read an MRG tree object from a Silo file

Synopsis:

```
DBmrgtree *DBGetMrgtree(DBfile *file, const char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`file`	The Silo database file handle
`name`	The name of the MRG tree object in the file.

Returns:

A pointer to a DBmrgtree object on success; NULL (0) on failure.

Description:

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBFreeMrgtree—Free the memory associated by an MRG tree object

Synopsis:

```
void DBFreeMrgtree(DBmrgtree *tree)
```

Fortran Equivalent:

```
integer function dbfreemrgtree(tree_id)
```

Arguments:

`tree` The MRG tree object to free.

Returns:

None.

Description:

Frees all the memory associated with an MRG tree.

DBMakeNamescheme—Create a DBnamescheme object for generating names

Synopsis:

```
DBnamescheme *DBMakeNamescheme(const char *fmt, ...)
```

Fortran Equivalent:

```
None
```

Arguments:

`fmt`	Format string for the name scheme as described below.
`...`	[Optional] additional arguments for external array references.

Description:

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 from printf-style formatting of simple expressions. Nameschemes are most often used to define names of regions in region arrays.

The format of a printf-style namescheme is as follows. The first character of `fmt` is treated as *delimiter character definition*. Wherever this delimiter character appears (except as the first character), this will indicate the end of one substring within `fmt` 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 first substring of `fmt` (that is the characters from position 1 to the first delimiter character) will contain the complete printf-style format string. 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 region 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 string (single quote) characters ‘’’ and ‘’’. 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 ‘$’ appearing before an array’s name.

Finally, the special operator ‘n’ appearing in an expression represents a region’s *natural* number within the region array (zero-origin region index). See below for some examples...

Except for singly quoted strings which evaluate to a literal string suitable for output via a %s type conversion specifier, 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..

fmt	Interpretation
“|slide_%s|(n%2)?’master’:’slave’:”	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)?’master’:’slave’:” 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 ‘master’. Otherwise, it will evaluate to ‘slave’. Note the terminating colon for the ?:: operator. This naming scheme might be useful for an array of regions representing, alternately, master and slave sides of slide surfaces.
“#block_%02dx%02d#n/16#n%16”	The delimiter character is ‘#’. 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 ‘$P[n]’ where the index into the array is just the natural number, n. Because ‘P’ is the first externally referenced array in the format string, it must be the first array to appear in the varargs list of additional args to DBMakeNamescheme. The expression substring for the second argument is another external array reference, ‘$U[n%4]’ where the index is an expression ‘n%4’ on the natural number n. Because U is the second externally referenced array, it must appear second in the varargs list of additional args to DBMakeNamescheme.

Use DBFreeNamescheme() to free up the space associated with a namescheme. Note, however, that DBFreenamescheme() does not free memory associated with external arrays.

DBGetName—Generate a name from a DBnamescheme object

Synopsis:

```
const char *DBGetName(DBnamescheme *ns, int natnum)
```

Fortran Equivalent:

```
None
```

Arguments:

`natnum`	Natural number of the entry in a namescheme to be generated. Must be greater than or equal to zero.

Returns:

A string representing the generated name. 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.

DBPutMrgvar—Write variable data to be associated with (some) regions in an MRG tree

Synopsis:

```
int DBPutMrgvar(DBfile *file, const char *name,
    const char *mrgt_name,
    int ncomps, const char **compnames,
    int nregns, const char **reg_pnames,
    int datatype, void **data, DBoptlist *opts)
```

Fortran Equivalent:

```
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” on page 248.)
character*N reg_names (See “dbset2dstrlen” on page 248.)
int* data_ids (use dbmkptr to get id for each pointer)
```

Arguments:

`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.

Returns:

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” on page 171. 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)
“<var>-extents”	A double precision variable defining the block-by-block extents of a multi-block variable. If <var>==”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.

DBGetMrgvar—Retrieve an MRG variable object from a silo file

Synopsis:

```
DBmrgvar *DBGetMrgvar(DBfile *file, const char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`file`	Silo database file handle.
`name`	The name of the region variable object to retrieve.

Returns:

A pointer to a DBmrgvar object on success; NULL (0) on failure.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBPutGroupelmap—Write a groupel map object to a Silo file

Synopsis:

```
int DBPutGroupelmap(DBfile *file, const char *name, int num_segs,
    int *seg_types, int *seg_lens, int *seg_ids, int **seg_data,
    void **seg_fracs, int fracs_type, DBoptlist *opts)
```

Fortran Equivalent:

```
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:

`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

Returns:

Zero on success; -1 on failure.

Description:

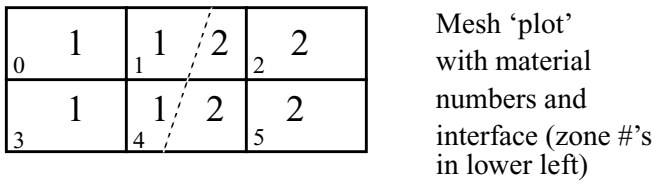
By itself, an MRG tree is not sufficient to fully characterize the decomposition of a mesh into various regions. The MRG tree serves only to identify the regions and their relationships in *gross* terms. This frees MRG trees from growing linearly (or worse) with problem size.

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 on page 120.




```
num_segs = 4;
seg_types[] = {DB_ZONECENT, DB_ZONECENT, DB_ZONECENT, DB_ZONECENT};
seg_lens[] = {2,2,2,2};
seg_ids[] ={1,1,2,2}; /* material numbers used as ids */
```

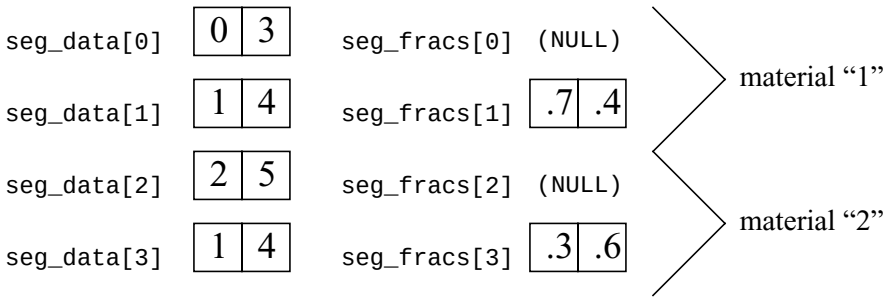


Figure 0-10: Example of using groupel map for (mixing) materials.

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.

DBGetGroupelmap—Read a groupel map object from a Silo file

Synopsis:

```
DBgroupelmap *DBGetGroupelmap(DBfile *file, const char *name)
```

Fortran Equivalent:

```
None
```

Arguments:

`file`	The Silo database file handle.
`name`	The name of the groupel map object to read.

Returns:

A pointer to a DBgroupelmap object on success. NULL (0) on failure.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBFreeGroupelmap—Free memory associated with a groupel map object

Synopsis:

```
void DBFreeGroupelmap(DBgroupelmap *map)
```

Fortran Equivalent:

```
None
```

Arguments:

`map`	Pointer to a DBgroupel map object.

Returns:

None

Description:

DBOPT_REGION_PNAMES—option for defining variables on specific regions of a mesh

Synopsis:

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.	NULL

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 materials. 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.

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 "Node, edge and face ordering for zoo-type UCD zone shapes." on page 2-80. 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 "Node, edge and face ordering for zoo-type UCD zone shapes." on page 2-80.

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" on page 2-69.

6 API Section Object Allocation and Free

This section describes methods to allocate and initialize many of Silo's objects. The functions described here are...

DBFree…188**DBAlloc…**—Allocate and initialize a Silo structure.

Synopsis:

```
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)
```

Fortran Equivalent:

```
None
```

Returns:

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….

DBFree…—Release memory associated with a Silo structure.

Synopsis:

```
void DBFreeCompoundarray (DBcompoundarray *x)
void DBFreeCsgmesh (DBcsgmesh *x)
void DBFreeCsgvar (DBcsgvar *x)
void DBFreeCSGZonelist (DBcsgzonelist *x)
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 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)
```

Arguments:

`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
```

Returns:

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.

7 API Section Calculational

This section of the API manual describes functions that can be used to compute things such as Facelists. Currently, only functions for calculating facelists are described here.

DBCalcExternalFacelist—Calculate an external facelist for a UCD mesh.

Synopsis:

```
DBfacelist *DBCalcExternalFacelist (int nodelist[], int nnodes,
                    int origin, int shapesize[],
                    int shapecnt[], int nshapes, int matlist[],
                    int bnd_method)
```

Fortran Equivalent:

```
integer function dbcalcfl(nodelist, nnodes, origin, shapesize,
                    shapecnt, nshapes, matlist, bnd_method)
returns the pointer-id of the created object.
```

Arguments:

`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.

Returns:

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 to nodes for the allowed shares are enumerated, see “DBPutUcdmesh” on page 2-77.

DBCalcExternalFacelist2—Calculate an external facelist for a UCD mesh containing ghost zones.

Synopsis:

```
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 Equivalent:

None

Arguments:

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.

Returns:

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” on page 2-77.

8 API Section 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. These are...

DBMakeOptlist—Allocate an option list.

Synopsis:

```
DBoptlist *DBMakeOptlist (int maxopts)
```

Fortran Equivalent:

```
integer function dbmkoptlist(maxopts, optlist_id)
returns created optlist pointer-id in optlist_id
```

Arguments:

`maxopts` Maximum number of options needed for this option list.

Returns:

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.

DBAddOption—Add an option to an option list.

Synopsis:

```
int DBAddOption (DBoptlist *optlist, int option, void *value)
```

Fortran Equivalent:

```
integer function dbaddcopt (optlist_id, option, 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
double precision dvalue
real rvalue
character*N cvalue (See “dbset2dstrlen” on page 248.)
```

Arguments:

`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`.

Returns:

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 ancillary nature.

DBClearOption—Remove an option from an option list

Synopsis:

```
int DBClearOption(DBoptlist *optlist, int optid)
```

Fortran Equivalent:

```
None
```

Arguments:

`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

Returns:

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.

DBGetOption—Retrieve the value set for an option in an option list

Synopsis:

```
void *DBGetOption(DBoptlist *optlist, int optid)
```

Fortran Equivalent:

```
None
```

Arguments:

`optlist`	The optlist to query
`optid`	The option id to query the value for

Returns:

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`.

DBFreeOptlist—Free memory associated with an option list.

Synopsis:

```
int DBFreeOptlist (DBoptlist *optlist)
```

Fortran Equivalent:

```
integer function dbfreeoptlist(optlist_id)
```

Arguments:

`optlist`	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.

Returns:

DBFreeOptlist returns a zero on success and -1 on failure.

Description:

The DBFreeOptlist function releases the memory associated with the given option list. The individual option values are not freed.

DBFreeOptlist will not fail if a NULL pointer is passed to it.

DBClearOptlist—Clear an optlist.

Synopsis:

```
int DBClearOptlist (DBoptlist *optlist)
```

Fortran Equivalent:

```
None
```

Arguments:

`optlist`	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.

Returns:

DBClearOptlist returns zero on success and -1 on failure.

Description:

The DBClearOptlist function removes all options from the given option list.

9 API Section 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 user-defined Silo *objects.* These include...

DBWrite—Write a simple variable.

Synopsis:

```
int DBWrite (DBfile *dbfile, char *varname, void *var, int *dims,
                    int ndims, int datatype)
```

Fortran Equivalent:

```
integer function dbwrite(dbid, varname, lvarname, var, dims,
                    ndims, datatype
```

Arguments:

`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.

Returns:

DBWrite returns zero on success and -1 on failure.

Description:

The DBWrite function writes a simple variable into a Silo file.

DBWriteSlice—Write a (hyper)slab of a simple variable

Synopsis:

```
int DBWriteSlice (DBfile *dbfile, char *varname, void *var,
                  int datatype, int *offset, int *length,
                  int *stride, int *dims, int ndims)
```

Fortran Equivalent:

```
integer function dbwriteslice(dbid, varname, lvarname, var,
                  datatype, offset, length, stride, dims, ndims)
```

Arguments:

`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.

Returns:

DBWriteSlice returns 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:

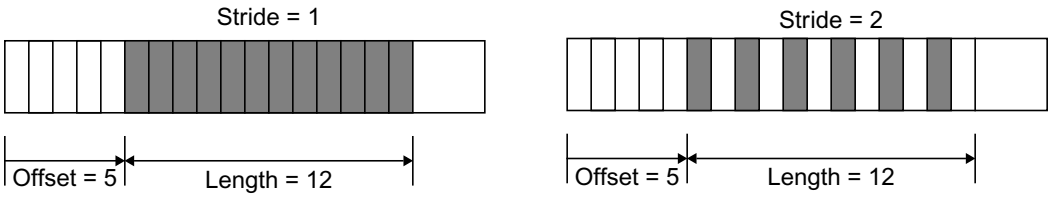


Figure 0-11: Array slice

DBReadVar—Read a simple Silo variable.

Synopsis:

```
int DBReadVar (DBfile *dbfile, char *varname, void *result)
```

Fortran Equivalent:

```
integer function dbrdvar(dbid, varname, lvarname, ptr)
```

Arguments:

`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.

Returns:

DBReadVar returns zero on success and -1 on failure.

Description:

The DBReadVar function reads a simple variable into the given space.

Notes:

See DBGetVar for a memory-allocating version of this function.

DBReadVar1—Read one element from a simple variable.

Synopsis:

```
int DBReadVar1 (DBfile *dbfile, char *varname, int offset,
                    void *result)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`offset`	Offset of one element to read.
`result`	Pointer to memory in which the element should be read. It is up to the application to provide sufficient space in which to read the element.

Returns:

DBReadVar1 returns zero on success and -1 on failure.

Description:

The DBReadVar1 function reads one element from a simple variable into the provided space.

DBReadVarSlice—Read a (hyper)slab of data from a simple variable.

Synopsis:

```
int DBReadVarSlice (DBfile *dbfile, char *varname, int *offset,
                    int *length, int *stride, int ndims,
                    void *result)
```

Fortran Equivalent:

```
integer function dbrdvarslice(dbid, varname, lvarname, offset,
                    length, stride, ndims, ptr)
```

Arguments:

`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.

Returns:

DBReadVarSlice returns 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:

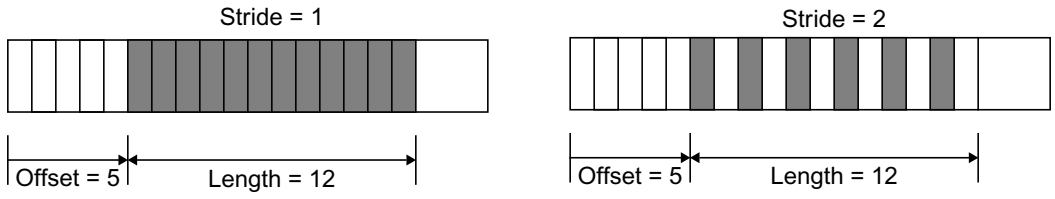


Figure 0-12: Array slice

DBGetVar—Allocate space for, and return, a simple variable.

Synopsis:

```
void *DBGetVar (DBfile *dbfile, char *varname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Name of the variable

Returns:

DBGetVar returns a pointer to newly allocated space 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.

Notes:

See DBReadVar and DBReadVar1 for non-memory allocating versions of this function.

DBInqVarExists—Queries variable existence

Synopsis:

```
int DBInqVarExists (DBfile *dbfile, char *name);
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`name`	Object name.

Returns:

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.

DBInqVarType—Return the type of the given object

Synopsis:

```
DBObjectType DBInqVarType (DBfile *dbfile, char *name);
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`name`	Object name.

Returns:

DBInqVarType 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 object or the object was not found in the file.	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

Object Type	Returned Value
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

The function will signal an error if the given `name` does not exist in the file.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBGetVarByteLength—Return the byte length of a simple variable.

Synopsis:

```
int DBGetVarByteLength (DBfile *dbfile, char *varname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Variable name.

Returns:

DBGetVarByteLength returns the length of the given simple variable in bytes on success and -1 on failure.

Description:

The DBGetVarByteLength function returns the 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.

DBGetVarDims—Get dimension information of a variable in a Silo file

Synopsis:

```
int DBGetVarDims(DBfile *file, const char *name, int maxdims,
    int *dims)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

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.

DBGetVarLength—Return the number of elements in a simple variable.

Synopsis:

```
int DBGetVarLength (DBfile *dbfile, char *varname)
```

Fortran Equivalent:

```
integer function dbinqlen(dbid, varname, lvarname, len)
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Variable name.

Returns:

DBGetVarLength returns 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.

DBGetVarType—Return the Silo datatype of a simple variable.

Synopsis:

```
int DBGetVarType (DBfile *dbfile, char *varname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`varname`	Variable name.

Returns:

DBGetVarType returns the Silo datatype of the given simple variable on success and -1 on failure.

Description:

The DBGetVarType function returns the Silo datatype of the requested simple variable. For example, DB_FLOAT for float variables.

Notes:

This only works for simple Silo variables (those written using DBWrite or DBWriteSlice). To query the type of other variables, use DBInqVarType instead.

DBPutCompoundarray—Write a Compound Array object into a Silo file.

Synopsis:

```
int DBPutCompoundarray (DBfile *dbfile, char *name,
                        char *elemnames[], int *elemlengths,
                        int nelems, void *values, int nvalues,
                        int datatype, DBoptlist *optlist);
```

Fortran Equivalent:

```
integer function dbputca(dbid, name, lname, elemnames, lelemnames,
                        elemlengths, nelems, values, nvalues,
                        datatype, optlist_id, status)
character*N elemnames (See “dbset2dstrlen” on page 248.)
```

Arguments:

`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.

Returns:

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 the simple arrays have elements of the same data type, and each have a name.

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.

DBInqCompoundarray—Inquire Compound Array attributes.

Synopsis:

```
int DBInqCompoundarray (DBfile *dbfile, char *name,
                        char *elemnames[], int *elemlengths,
                        int nelems, int nvalues, int datatype)
```

Fortran Equivalent:

```
integer function dbinqca(dbid, name, lname, maxwidth, nelems,
                        nvalues, datatype)
```

Arguments:

`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.

Returns:

DBInqCompoundarray returns 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.

DBGetCompoundarray—Read a compound array from a Silo database.

Synopsis:

```
DBcompoundarray *DBGetCompoundarray (DBfile *dbfile,
                    char *arrayname)
```

Fortran Equivalent:

```
integer function dbgetca(dbid, name, lname, lelemnames, elemnames,
                    elemlengths, nelems, values, nvalues,
                    datatype)
```

Arguments:

`dbfile`	Database file pointer.
`arrayname`	Name of the compound array.

Returns:

DBGetCompoundarray returns 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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBMakeObject—Allocate an object of the specified length and initialize it.

Synopsis:

```
DBobject *DBMakeObject (char *objname, int objtype, int maxcomps)
```

Fortran Equivalent:

```
None
```

Arguments:

`objname`	Name of the object.
`objtype`	Type of object. One of the predefined types: DB_QUADMESH, DB_QUAD_RECT, DB_QUAD_CURV, DB_DEFVARS, DB_QUADVAR, DB_UCDMESH, DB_UCDVAR, DB_POINTMESH, DB_POINTVAR, DB_CSGMESH, DB_CSGVAR, DB_MULTIMESH, DB_MULTIVAR, DB_MULTIADJ, DB_MATERIAL, DB_MATSPECIES, DB_FACELIST, DB_ZONELIST, DB_PHZONELIST, DB_EDGELIST, DB_CURVE, DB_ARRAY, or DB_USERDEF.
`maxcomps`	Maximum number of components needed for this object.

Returns:

DBMakeObject returns a pointer to the newly allocated and initialized object on success and NULL on failure.

Description:

The DBMakeObject function allocates space for an object of `maxcomps` components.

DBFreeObject—Free memory associated with an object.

Synopsis:

```
int DBFreeObject (DBobject *object)
```

Fortran Equivalent:

```
None
```

Arguments:

`object`	Pointer to the object to be freed. This object is created with the DBMakeObject function.

Returns:

DBFreeObject returns 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.

DBChangeObject—Overwrite an existing object in a Silo file with a new object

Synopsis:

```
int DBChangeObject(DBfile *file, DBobject *obj)
```

Fortran Equivalent:

```
None
```

Arguments:

`file`	The Silo database file handle.
`obj`	The new DBobject object (which knows its name) to write to the file.

Returns:

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.

DBClearObject—Clear an object.

Synopsis:

```
int DBClearObject (DBobject *object)
```

Fortran Equivalent:

```
None
```

Arguments:

`object`	Pointer to the object to be cleared. This object is created with the DBMakeObject function.

Returns:

DBClearObject returns 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.

DBAddDblComponent—Add a double precision floating point component to an object.

Synopsis:

```
int DBAddDblComponent (DBobject *object, char *compname, double d)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBAddDblComponent returns zero on success and -1 on failure.

Description:

The DBAddDblComponent function adds a component of double precision floating point data to an existing object.

DBAddFltComponent—Add a floating point component to an object.

Synopsis:

```
int DBAddFltComponent (DBobject *object, char *compname, double f)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBAddFltComponent returns zero on success and -1 on failure.

Description:

The DBAddFltComponent function adds a component of floating point data to an existing object.

DBAddIntComponent—Add an integer component to an object.

Synopsis:

```
int DBAddIntComponent (DBobject *object, char *compname, int i)
```

Fortran Equivalent:

```
None
```

Arguments:

`object`	Pointer to an object. This object is created with the DBMakeObject function.
`compname`	The component name.
`i`	The value of the integer component.

Returns:

DBAddIntComponent returns zero on success and -1 on failure.

Description:

The DBAddIntComponent function adds a component of integer data to an existing object.

DBAddStrComponent—Add a string component to an object.

Synopsis:

```
int DBAddStrComponent (DBobject *object, char *compname, char *s)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBAddStrComponent returns zero on success and -1 on failure.

Description:

The DBAddStrComponent function adds a component of string data to an existing object.

DBAddVarComponent—Add a variable component to an object.

Synopsis:

```
int DBAddVarComponent (DBobject *object, char* compname,
                       char *vardata)
```

Fortran Equivalent:

```
None
```

Arguments:

`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).

Returns:

DBAddVarComponent returns zero on success and -1 on failure.

Description:

The DBAddVarComponent function adds a component of the variable type to an existing object.

The variable in `vardata` is stored verbatim into the object. No translation or typing is done on the variable as it is added to the object.

DBWriteComponent—Add a variable component to an object and write the associated data.

Synopsis:

```
int DBWriteComponent (DBfile *dbfile, DBobject *object,
                      char *compname, char *prefix, char *datatype,
                      void *var, int nd, long *count)
```

Fortran Equivalent:

None

Arguments:

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.

Returns:

DBWriteComponent returns 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.

DBWriteObject—Write an object into a Silo file.

Synopsis:

```
int DBWriteObject (DBfile *dbfile, DBobject *object, int freemem)
```

Fortran Equivalent:

```
None
```

Arguments:

`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.

Returns:

DBWriteObject returns zero on success and -1 on failure.

Description:

The DBWriteObject function writes an object into a Silo file. This is a user-defined object that consists of various components. They are used when the basic Silo structures are not sufficient.

DBGetObject—Read an object from a Silo file as a generic object

Synopsis:

```
DBobject *DBGetObject(DBfile *file, const char *objname)
```

Fortran Equivalent:

```
None
```

Arguments:

`file`	The Silo database file handle.
`objname`	The name of the object to get.

Returns:

On success, a pointer to a DBobject struct containing the object's data. NULL on failure.

Description:

Each of the object Silo supports has corresponding methods to both write them to a Silo database file (DBPut...) and get them from a file (DBGet...).

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.

Notes:

For the details of the data structured returned by this function, see the Silo library header file, silo.h, also attached to the end of this manual.

DBGetComponent—Allocate space for, and return, an object component.

Synopsis:

```
void *DBGetComponent (DBfile *dbfile, char *objname,
                      char *compname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`objname`	Object name.
`compname`	Component name.

Returns:

DBGetComponent returns 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.

DBGetComponentType—Return the type of an object component.

Synopsis:

```
int DBGetComponentType (DBfile *dbfile, char *objname,
                        char *compname)
```

Fortran Equivalent:

```
None
```

Arguments:

`dbfile`	Database file pointer.
`objname`	Object name.
`compname`	Component name.

Returns:

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
Integer	DB_INT
Float	DB_FLOAT
Double	DB_DOUBLE
String	DB_CHAR
Variable	DB_VARIABLE
all others	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.

10 API Section 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.

`_visit_defvars`—convention for derived variable definitions

Synopsis:

```
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” on page 126.

`_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.

`_visit_searchpath`—directory order to search when opening a Silo file

Synopsis:

```
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.

`_visit_domain_groups`—method for grouping blocks in a multi-block mesh

Synopsis:

```
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" on page 165.

`_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 above, there are 16 blocks assigned to 4 groups.

`AlphabetizeVariables`—flag to tell post-processor to alphabetize variable lists

Synopsis:

```
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.

`ConnectivityIsTimeVarying`—flag telling post-processor if connectivity of meshes in the Silo file is time varying or not

Synopsis:

```
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.

`MultivarToMultimeshMap_vars`—list of multivars to be associated with multimeshes

Synopsis:

```
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 short-coming 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 `DBPut-Multivar()` call.

MultivarToMultimeshMap_meshes—list of multimeshes to be associated with multivars

Synopsis:

```
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 243.

11 API Section Silo's 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. The functions described here are...

dbmkptr—create a *pointer-id* from a pointer

Synopsis:

```
integer function dbmkptr(void p)
```

Arguments:

p	pointer for which a *pointer-id* is needed

Returns:

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-id*s. 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()`

dbrmptr—remove an old and no longer needed pointer-id

Synopsis:

```
integer function dbrmptr(ptr_id)
```

Arguments:

`ptr_id` the pointer-id to remove

Returns:

always 0

dbset2dstrlen—Set the size of a ‘row’ for pointers to ‘arrays’ of strings

Synopsis:

```
integer function dbset2dstrlen(int len)

integer len
```

Arguments:

`len`	The length to set

Returns:

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.

dbget2dstrlen—Get the size of a ‘row’ for pointers to ‘arrays’ of character strings

Synopsis:

```
integer function dbget2dstrlen()
```

Arguments:

```
None
```

Returns:

The current setting for the 2D string length.

DBFortranAllocPointer—Facilitates accessing C objects through Fortran

Synopsis:

```
int DBFortranAllocPointer (void *pointer)
```

Arguments:

`pointer` A pointer to a Silo object for which a Fortran identifier is needed

Returns:

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” on page 2-251 and “DBFortranRemovePointer” on page 2-252 for more information about how to use Silo objects in code that uses C and Fortran together.

DBFortranAccessPointer—Access Silo objects created through the Fortran Silo interface.

Synopsis:

```
void *DBFortranAccessPointer (int value)
```

Arguments:

`value` The value returned by a Silo Fortran function, referencing a Silo object.

Returns:

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” on page 2-250 and “DBFortranRemovePointer” on page 2-252 for more information about how to use Silo objects in code that uses C and Fortran together.

DBFortranRemovePointer—Removes a pointer from the Fortran-C index table

Synopsis:

```
void DBFortranRemovePointer (int value)
```

Arguments:

`value`	An integer returned by DBFortranAllocPointer

Returns:

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” on page 2-251 and “DBFortranAllocPointer” on page 2-250 for more information about how to use Silo objects in code that uses C and Fortran together.

12 API Section Deprecated Functions

The following functions were deprecated from Silo in version 4.6. Attempts to call these methods in version 4.6 will still succeed. However, deprecation warnings will be generated on stderr (See “DBSetDeprecateWarnings” on page 31.). There is no guarantee that these methods will exist in versions of Silo after 4.6.

DBGetComponentNames
DBGetAtt
DBListDir
DBReadAtt
DBGetQuadvar1
DBcontinue
DBPause
DBPutZonelist
DBPutUcdsubmesh

```
/*
 * SILO Public header file.
 *
 * This header file defines public constants and public prototypes.
 * Before including this file, the application should define
 * which file formats will be used.
 *
 */
#ifndef SILO_H
#define SILO_H

#ifdef __cplusplus
extern "C" {
#endif

/* Set the base type for datatype'd pointers (that is pointers whose
   ultimate type is deteremined by an additional 'int datatype' function
   argument or struct member) as float (legacy) and void (modern). The
   DB_DTPTR is the base type. The '1' and '2' variants are for singley
   subscripted and doubley subscripted arrays, respectively. If the
   definitions of DB_DTPTR below reference 'float', then this silo.h
   header file was configured with --enable-legacy-datatyped-pointers
   and it represents the legacy (float) pointers that the silo
   library has always had since its original writing. If, instead,
   you see 'void' (the default configuration), then this silo.h header
   file is using the modern (void) pointers. In that case, note also
   that because C compiler's often do not handle correctly nor
   distinguish between a void* and a void**, both the singley and
   doubley subscripted variants will have only a single star. Rest
   assured they are still treated as doubley subscripted in the
   implementation. */
#define DB_DTPTR  @SILO_DTYPPTR@  /* NO_FORTRAN_DEFINE */
#define DB_DTPTR1 @SILO_DTYPPTR1@ /* NO_FORTRAN_DEFINE */
#define DB_DTPTR2 @SILO_DTYPPTR2@ /* NO_FORTRAN_DEFINE */

/* Permit client to explicitly require the legacy mode
   for datatyped pointers */
#ifdef DB_USE_LEGACY_DTPTR
#ifdef DB_USE_MODERN_DTPTR
#error cannot specify BOTH legacy and modern datatyped pointers
#endif
#undef DB_DTPTR  /* NO_FORTRAN_DEFINE */
#undef DB_DTPTR1 /* NO_FORTRAN_DEFINE */
#undef DB_DTPTR2 /* NO_FORTRAN_DEFINE */
#define DB_DTPTR  float   /* NO_FORTRAN_DEFINE */
#define DB_DTPTR1 float*  /* NO_FORTRAN_DEFINE */
#define DB_DTPTR2 float** /* NO_FORTRAN_DEFINE */
#endif

/* Permit client to explicitly require the modern mode
   for datatyped pointers */
#ifdef DB_USE_MODERN_DTPTR
#undef DB_DTPTR  /* NO_FORTRAN_DEFINE */
#undef DB_DTPTR1 /* NO_FORTRAN_DEFINE */
#undef DB_DTPTR2 /* NO_FORTRAN_DEFINE */
#define DB_DTPTR  void  /* NO_FORTRAN_DEFINE */
#define DB_DTPTR1 void* /* NO_FORTRAN_DEFINE */
#define DB_DTPTR2 void* /* NO_FORTRAN_DEFINE */
#endif
```

```
#include <stdio.h>

#ifndef FALSE
#define FALSE   0
#endif
#ifndef TRUE
#define TRUE    1
#endif

/* In the definitions for different parts of the version number, below,
   we use leading '0x0' to deal with possible blank minor and/or patch
   version number but still allow base-10, numeric comparison in the GE
   macro. */

/* Major release number of silo library. */
#define SILO_VERS_MAJ @SILO_VERS_MAJ@

/* Minor release number of silo library. Can be empty. */
#define SILO_VERS_MIN 0x0@SILO_VERS_MIN@

/* Patch release number of silo library.  Can be empty. */
#define SILO_VERS_PAT 0x0@SILO_VERS_PAT@

/* Pre-release release number of silo library.  Can be empty. */
#define SILO_VERS_PRE @SILO_VERS_PRE@

/* The symbol Silo uses to enforce link-time
   header/object version compatibility */
#define SILO_VERS_TAG @SILO_VERS_TAG@

/* Useful macro for comparing Silo versions (and DB_ alias) */
#define SILO_VERSION_GE(Maj,Min,Pat)  \
        (((SILO_VERS_MAJ==Maj) && (SILO_VERS_MIN==0x0 ## Min) && (SILO_VERS_PAT>=0x0
## Pat)) || \
         ((SILO_VERS_MAJ==Maj) && (SILO_VERS_MIN>0x0 ## Min)) || \
         (SILO_VERS_MAJ>Maj))
#define DB_VERSION_GE(Maj,Min,Pat) SILO_VERSION_GE(Maj,Min,Pat)

/*-------------------------------------------------------------------------
 * Drivers.  This is a list of every driver that a user could use.  Not all of
 * them are necessarily compiled into the library.  However, users are free
 * to try without getting compilation errors.  They are listed here so that
 * silo.h doesn't have to be generated every time the library is recompiled.
 *-------------------------------------------------------------------------*/
#define DB_NETCDF 0
#define DB_PDB 2
#define DB_TAURUS 3
#define DB_UNKNOWN 5
#define DB_DEBUG 6
#define DB_HDF5 7           /* equivalent to DB_HDF5_SEC2 */

/* special driver ids to affect which Virtual File Driver HDF5 uses */
#define DB_HDF5_SEC2   256  /* section 2 I/O (open/read/write/close) */
#define DB_HDF5_STDIO  512  /* stdio (fopen/fread/fwrite/fclose) */
#define DB_HDF5_CORE   768  /* file in memory. MSbits specify alloc. inc. */
#define DB_HDF5_MPIO   1024  /* use MPI-IO on MPI_COMM_SELF */
#define DB_HDF5_MPIOP  1280  /* use MPI for any messaging, sec 2 for I/O */
```

```
/*-------------------------------------------------------------------------
 * Other library-wide constants.
 *-------------------------------------------------------------------------*/
#define DB_NFILES       256         /*Max simultaneously open files */
#define DB_NFILTERS     32          /*Number of filters defined */

/*-------------------------------------------------------------------------
 * Constants.  All of these constants are always defined in the application.
 * Each group of constants defined here are small integers used as an index
 * into an array.  Many of the groups have a total count of items in the
 * group that will be used for array allocation and error checking--don't
 * forget to increment the value when adding a new item to a constant group.
 *-------------------------------------------------------------------------
 */

/* The following identifiers are for use with the DBDataReadMask() call.  They
 * specify what portions of the data beyond the metadata is allocated
 * and read.  Note that since these values are only necessary when reading
 * and since the Fortran interface is primarily a write interface, it is not
 * necessary for these symbols to appear in the silo.inc file. However, the
 * reason they DO NOT APPEAR there inspite of the fact that DO NOT HAVE the
 * 'NO_FORTRAN_DEFINE' text appearing on each line is that the mkinc script
 * requires an underscore in the symbol name for it to appear in silo.inc. */
#define DBAll                0xffffffff
#define DBNone               0x00000000
#define DBCalc               0x00000001
#define DBMatMatnos          0x00000002
#define DBMatMatlist         0x00000004
#define DBMatMixList         0x00000008
#define DBCurveArrays        0x00000010
#define DBPMCoords           0x00000020
#define DBPVData             0x00000040
#define DBQMCoords           0x00000080
#define DBQVData             0x00000100
#define DBUMCoords           0x00000200
#define DBUMFacelist         0x00000400
#define DBUMZonelist         0x00000800
#define DBUVData             0x00001000
#define DBFacelistInfo       0x00002000
#define DBZonelistInfo       0x00004000
#define DBMatMatnames        0x00008000
#define DBUMGlobNodeNo       0x00010000
#define DBZonelistGlobZoneNo 0x00020000
#define DBMatMatcolors       0x00040000
#define DBCSGMBoundaryInfo   0x00080000
#define DBCSGMZonelist       0x00100000
#define DBCSGMBoundaryNames  0x00200000
#define DBCSGVData           0x00400000
#define DBCSGZonelistZoneNames 0x00800000
#define DBCSGZonelistRegNames  0x01000000
#define DBMMADJNodelists     0x02000000
#define DBMMADJZonelists     0x04000000
#define DBPMGlobNodeNo       0x08000000

/* Definitions for COORD_TYPE */
/* Placed before DBObjectType enum because the
   DB_QUAD_CURV and DB_QUAD_RECT symbols are
   sometimes used as DBObjectType */
```

```
#define  DB_COLLINEAR           130
#define  DB_NONCOLLINEAR        131
#define  DB_QUAD_RECT           DB_COLLINEAR
#define  DB_QUAD_CURV           DB_NONCOLLINEAR

/* 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_VARIABLE=610,
    DB_MRGTREE=611,
    DB_GROUPELMAP=612,
    DB_MRGVAR=613,
    DB_USERDEF=700
} DBObjectType;

/* 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;

/* Flags for DBCreate */
#define         DB_CLOBBER      0
#define         DB_NOCLOBBER    1
```

```
/* Flags for DBOpen */
#define         DB_READ         1
#define         DB_APPEND       2

/* 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

/* Options */
#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
#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
#define DBOPT_SPECNAMES         326
#define DBOPT_SPECCOLORS        327
#define DBOPT_LLONGNZNUM        328
#define DBOPT_CONSERVED         329
#define DBOPT_EXTENSIVE         330

/* 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 */

/* 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 */
#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_NERRORS   50

/* Definitions for MAJOR_ORDER */
#define  DB_ROWMAJOR            0
#define  DB_COLMAJOR            1

/* 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 */

/* Definitions for COORD_SYSTEM */
#define  DB_CARTESIAN           120
#define  DB_CYLINDRICAL         121
#define  DB_SPHERICAL           122
#define  DB_NUMERICAL           123
#define  DB_OTHER               124

/* Definitions for ZONE FACE_TYPE */
#define  DB_RECTILINEAR         100
#define  DB_CURVILINEAR         101

/* Definitions for PLANAR */
#define  DB_AREA                140
#define  DB_VOLUME              141

/* Definitions for flag values */
#define DB_ON                    1000
#define DB_OFF                  -1000

/* Definitions for disjoint flag */
#define DB_ABUTTING              142
#define DB_FLOATING              143

/* 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

/* 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

/* 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

/* definitions for MRG Tree traversal flags */
#define DB_PREORDER             0x00000001
#define DB_POSTORDER            0x00000002
#define DB_FROMCWR              0x00000004

/* Miscellaneous constants */
#define     DB_F77NULL  (-99)   /*Fortran NULL pointer      */
#define     DB_F77NULLSTRING  "NULLSTRING"  /* FORTRAN STRING */

/*-------------------------------------------------------------------------
 * Index selection macros
 *-------------------------------------------------------------------------
 */
#define I4D(s,i,j,k,l) (l)*s[3]+(k)*s[2]+(j)*s[1]+(i)*s[0]
#define I3D(s,i,j,k)   (k)*s[2]+(j)*s[1]+(i)*s[0]
#define I2D(s,i,j)     (j)*s[1]+(i)*s[0]

/*-------------------------------------------------------------------------
 * Structures (just the public parts).
 *-------------------------------------------------------------------------
 */

/*
 * Database table of contents for the current directory only.
 */
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;

    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            narrays;

    char         **mrgtree_names;
    int            nmrgtrees;

    char         **groupelmap_names;
    int            ngroupelmaps;

    char         **mrgvar_names;
    int            nmrgvars;

} DBtoc;

/*----------------------------------------------------------------------------
 * Database Curve Object
 *--------------------------------------------------------------------------
 */
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  */
    DB_DTPTR      *x;           /* Domain values for curve */
    DB_DTPTR      *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 */
} DBcurve;

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;

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 */

    DB_DTPTR      *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 */
} DBpointmesh;

/*-----------------------------------------------------------------------------
 * Multi-Block Mesh Object
 *---------------------------------------------------------------------------
 */
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 */
    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. */
} DBmultimesh;

/*-----------------------------------------------------------------------------
 * Multi-Block Mesh Adjacency Object
 *---------------------------------------------------------------------------
 */
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;

/*-----------------------------------------------------------------------------
 * Multi-Block Variable Object
 *---------------------------------------------------------------------------
 */
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) */
} DBmultivar;

/*----------------------------------------------------------------------------
 * Multi-material
 *----------------------------------------------------------------------------
 */
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;
} DBmultimat;

/*----------------------------------------------------------------------------
 * Multi-species
 *----------------------------------------------------------------------------
 */
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 */
} DBmultimatspecies;

/*----------------------------------------------------------------------
 *  Definitions for the FaceList, ZoneList, and EdgeList structures
 *  used for describing UCD meshes.
 *--------------------------------------------------------------------
 */

#define DB_ZONETYPE_BEAM        10

#define DB_ZONETYPE_POLYGON     20
```

```
#define DB_ZONETYPE_TRIANGLE    23
#define DB_ZONETYPE_QUAD        24

#define DB_ZONETYPE_POLYHEDRON  30
#define DB_ZONETYPE_TET         34
#define DB_ZONETYPE_PYRAMID     35
#define DB_ZONETYPE_PRISM       36
#define DB_ZONETYPE_HEX         38

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 */
} DBzonelist;

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 */
} DBphzonelist;

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 ---------*/
    int           *nodeno;      /* [lnodelist] node number of each node */

/*----------- Optional zone-reference component---------*/
    int           *zoneno;      /* [nfaces] Zone number for each face */
} DBfacelist;

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;

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? */

    DB_DTPTR      *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 */
    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 */
} DBquadmesh;

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 */

    DB_DTPTR      *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' */

    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 */
} DBucdmesh;
```

```
/*----------------------------------------------------------------------------
 * Database Mesh-Variable Object
 *----------------------------------------------------------------------------
 */
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)
*/

    DB_DTPTR     **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 */
    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 */

    DB_DTPTR     **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. */
} DBquadvar;
```

```
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)
*/

    DB_DTPTR     **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' */

    int            centering;   /* Centering within mesh (nodal or zonal) */
    DB_DTPTR     **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) */
} DBucdvar;

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)
*/

    DB_DTPTR     **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 */

     /* 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) */
} DBmeshvar;

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) */
     DB_DTPTR      *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 */

     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;

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. */
    DB_DTPTR      *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 */
    char         **specnames;   /* Array of species names; length is sum of nmatspec
*/
    char         **speccolors;  /* Array of species colors; length is sum of nmatspec
*/
} DBmatspecies;

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 */
} DBcsgzonelist;

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 */
    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;
} DBcsgmesh;

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
                                   physical property (as opposed to intensive) */
} DBcsgvar;

/*----------------------------------------------------------------------------
 * A compound array is an array whose elements are simple arrays. A simple
 * array is an array whose elements are all of the same primitive data
 * type: float, double, integer, long...  All of the simple arrays of
 * a compound array have elements of the same data type.
 *----------------------------------------------------------------------------
 */
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;
```