./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

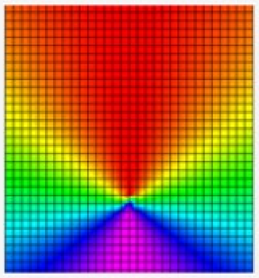
./ Silo

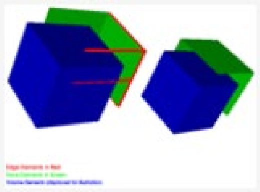
Mesh and Field I/O Library and Scientific Database

View on GitHub

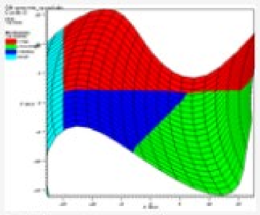
About Silo…

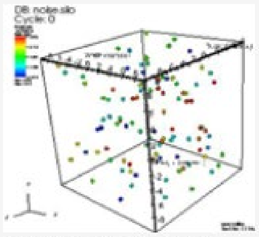
Silo is a library for reading and writing a wide variety of scientific data to binary, disk files. The files Silo produces and the data within them can be easily shared and exchanged between wholly independently developed applications running on disparate computing platforms. Consequently, Silo facilitates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

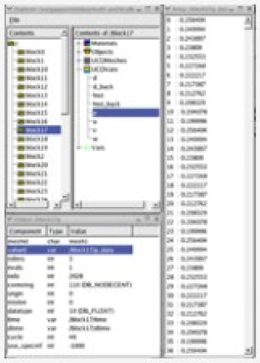

Structured Rectilinear Mesh

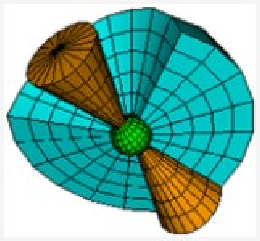

Arbitrary Subsets

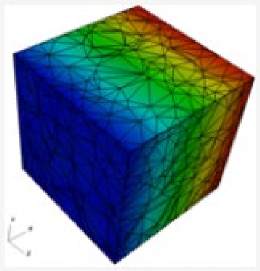
Constructive Solid Geometry (CSG) Mesh

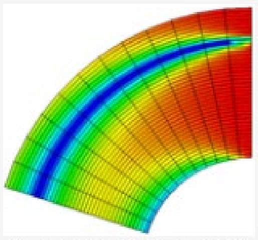

Mixing Materials

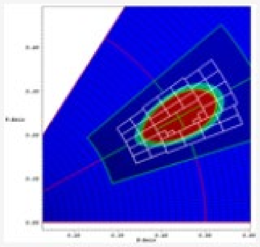

Gridless Point Mesh

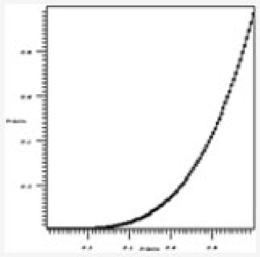

Silex browser for Silo files


Unstructured Zoo (UCD) Mesh


Arbitrary Polyhedral Mesh


Structured (Curvilinear) Mesh


Adaptive Mesh Refinement (AMR) Mesh


XY Curve

Silo supports gridless (point) meshes, structured meshes, unstructured-zoo and unstructured-arbitrary-polyhedral meshes, block structured AMR meshes, constructive solid geometry (CSG) meshes, piecewise-constant (e.g., zone-centered) and piecewise-linear (e.g. node-centered) variables defined on the node, edge, face or volume elements of meshes as well as the decomposition of meshes into arbitrary subset hierarchies including materials and mixing materials. In addition, Silo supports a wide variety of other useful objects to address various scientific computing application needs. Although the Silo library is a serial library, it has some key features which enable it to be applied quite effectively and scalable 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) and PDB (Portable Data Base) drivers.

Read more about Silo.

Manual

Contacts

The SILO team uses the LLNL listserv list system to make announcements. To join the announcement list, send an e-mail message to LISTSERV@LISTSERV.LLNL.GOV with no subject line and a message body containing:

```
SUBscribe silo-announce ANONYMOUS
```

The list is for the SILO team to make announcements to the Silo user community and is not a moderated discussion list.

For any additional inquiries, use our GitHub Discussions space

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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…

`DBErrfuncname()` - Get name of error-generating function

C Signature

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

Fortran Signature:

```
None
```

`DBErrno()` - Get internal error number.

C Signature

```
int DBErrno (void)
```

Fortran Signature

```
integer function dberrno()
```

DBErrString() - Get error message.

C Signature

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

Fortran Signature:

```
None
```

DBShowErrors() - Set the error reporting mode.

C Signature

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

Fortran Signature

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

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

DBErrlvl() - Return current error level setting of the library

C Signature

```
int DBErrlvl(void)
```

Fortran Signature

```
int dberrlvl()
```

`DBErrfunc()` - Get current error function set by DBShowErrors()

C Signature

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

Fortran Signature:

```
None
```

`DBVariableNameValid()` - check if character string represents a valid Silo variable name

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`s`	The character string to check

`DBVersion()` - Get the version of the Silo library.

C Signature

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

Fortran Signature:

```
None
```

`DBVersionDigits()` - Return the integer version digits of the library

C Signature

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

Fortran Signature:

```
None
```

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

`DBVersionGE()` - Greater than or equal comparison for version of the Silo library

C Signature

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

Fortran Signature:

```
None
```

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

`DBSetAllowOverwrites()` - Allow library to over-write existing objects in Silo files

C Signature

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

Fortran Signature

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

Arg name	Description
`allow`	Integer value controlling the Silo library's overwrite behavior. A non-zero value sets the Silo library to permit overwrites of existing objects. A zero value disables overwrites. By default, Silo does NOT permit overwrites.

`DBGetAllowOverwrites()` - Get current setting for the allow overwrites flag

C Signature

```
int DBGetAllowOverwrites(void)
```

Fortran Signature

```
integer function dbgetovrwrt()
```

`DBSetAllowEmptyObjects()` - Permit the creation of empty silo objects

C Signature

```
int DBSetAllowEmptyObjects(int allow)
```

Fortran Signature

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

Arg name	Description
`allow`	Integer value indicating whether or not empty objects should be allowed to be created in Silo files. A zero value prevents callers from creating empty objects in Silo files. A non-zero value permits it. By default, the Silo library does NOT permit callers to create empty objects.

`DBGetAllowEmptyObjects()` - Get current setting for the allow empty objects flag

C Signature

```
int DBGetAllowEmptyobjets(void)
```

Fortran Signature

```
integer function dbgetemptyok()
```

Arguments: None

`DBForceSingle()` - Convert all datatype'd data read in read methods to type float

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetDatatypeString()` - Return a string name for a given Silo datatype

C Signature

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

Fortran Signature:

```
None
```

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

`DBSetDataReadMask2()` - Set the data read mask

C Signature

```
unsigned long long DBSetDataReadMask2 (unsigned long long mask)
```

Fortran Signature:

```
None
```

Arg name	Description
`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.

`DBGetDataReadMask2()` - Get the current data read mask

C Signature

```
unsigned long long DBGetDataReadMask2 (void)
```

Fortran Signature:

```
None
```

`DBSetEnableChecksums()` - Set flag controlling checksum checks

C Signature

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

Fortran Signature

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

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

`DBGetEnableChecksums()` - Get current state of flag controlling checksumming

C Signature

```
int DBGetEnableChecksums(void)
```

Fortran Signature

```
integer function dbgetcksums()
```

`DBSetCompression()` - Set compression options for succeeding writes of Silo data

Arg name	Description

`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.

`DBGetCompression()` - Get current compression parameters

C Signature

```
char const *DBGetCompression()
```

Fortran Signature

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

Arguments: None

`DBSetFriendlyHDF5Names()` - Set flag to indicate Silo should create friendly names for HDF5 datasets

C Signature

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

Fortran Signature

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

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

`DBGetFriendlyHDF5Names()` - Get setting for friendly HDF5 names flag

C Signature

```
int DBGetFriendlyHDF5Names()
```

Fortran Signature

```
integer function dbgethdfnms()
```

Arguments: None

`DBSetDeprecateWarnings()` - Set maximum number of deprecate warnings Silo will issue for any one function, option or convention

C Signature

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

Fortran Signature

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

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

`DBGetDeprecateWarnings()` - Get maximum number of deprecated function warnings Silo will issue

C Signature

```
int DBGetDeprecateWarnings()
```

Fortran Signature

```
integer function dbgetdepwarn()
```

Arguments: None

`DB_VERSION_GE()` - Compile time macro to test silo version number

C Signature

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

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

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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 157.

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 157.), 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…

`DBRegisterFileOptionsSet()` - Register a set of options for advanced control of the low-level I/O driver

C Signature

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

Fortran Signature

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

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

`DBUnregisterFileOptionsSet()` - Unregister a registered file options set

C Signature

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

Fortran Signature

```
```

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

`DBUnregisterAllFileOptionsSets()` - Unregister all file options sets

C Signature

```
int DBUnregisterAllFileOptionsSets()
```

Fortran Signature

```
```

Arguments: None

DBSetUnknownDriverPriorities() - Set driver priorities for opening files with the DB_UNKNOWN driver.

C Signature

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

Fortran Signature:

```
None
```

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

DBGetUnknownDriverPriorities() - Return the currently defined ordering of drivers the DB_UNKNOWN driver will attempt.

C Signature

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

Fortran Signature:

```
None
```

DBCreate() - Create a Silo output file.

C Signature

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

Fortran Signature

```
integer function dbcreate(pathname, lpathname, mode, target,
        fileinfo, lfileinfo, filetype, dbid)
returns created database file handle in dbid
```

Arg name	Description
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, pass NULL for this argument.
filetype	Destination file type. Applications typically use one of either DB_PDB, which will create PDB files, or DB_HDF5, which will create HDF5 files. Other options include DB_PDBP, DB_HDF5_SEC2, DB_HDF5_STDIO, DB_HDF5_CORE, DB_HDF5_SPLIT or DB_FILE_OPTS(optlist_id) where optlist_id is a registered file options set. For a description of the meaning of these options as well as many other advanced features and control of underlying I/O behavior, see "DBRegisterFileOptionsSet" on page 2-40.

DBOpen() - Open an existing Silo file.

C Signature

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

Fortran Signature

```
integer function dbopen(name, lname, type, mode,
   dbid)
returns database file handle in dbid.
```

Arg name	Description
`name`	Name of the file to open. Can be either an absolute or relative path.
`type`	The type of file to open. One of the predefined types, typically DB_UNKNOWN, DB_PDB, or DB_HDF5. However, there are other options as well as subtle but important issues in using them. So, read description, below for more details.
`mode`	The mode of the file to open. One of the values DB_READ or DB_APPEND.

`DBClose()` - Close a Silo database.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.

`DBGetToc()` - Get the table of contents of a Silo database.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.

`DBFileVersion()` - Version of the Silo library used to create the specified file

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file handle

`DBFileVersionDigits()` - Return integer digits of file version number

C Signature

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

Arg name	Description
`dbfile`	Silo database file handle

`maj`	Pointer to returned major version digit
`min`	Pointer to returned minor version digit
`pat`	Pointer to returned patch version digit
`pre`	Pointer to returned pre-release version digit (if any)

`DBFileVersionGE()` - Greater than or equal comparison for version of the Silo library a given file was created with

C Signature

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

Fortran Signature:

```
None
```

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

`DBVersionGEFileVersion()` - Compare library version with file version

C Signature

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

Fortran Signature:

```
None
```

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

`DBSortObjectsByOffset()` - Sort list of object names by order of offset in the file

C Signature

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

Fortran Signature:

```
None
```

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

`DBMkDir()` - Create a new directory in a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description

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

`DBSetDir()` - Set the current directory within the Silo database.

C Signature

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

Fortran Signature

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

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

`DBGetDir()` - Get the name of the current directory.

C Signature

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

Fortran Signature:

```
None
```

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

`DBCpDir()` - Copy a directory hierarchy from one Silo file to another.

C Signature

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

Fortran Signature:

```
None
```

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

`DBCpListedObjects()` - Copy lists of objects from one Silo database to another

C Signature

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

Fortran Signature:

```
None
```

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

`DBGrabDriver()` - Obtain the low-level driver file handle

C Signature

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

Fortran Signature:

```
None
```

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

`DBUngrabDriver()` - Ungrab the low-level file driver

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetDriverType()` - Get the type of driver for the specified file

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetDriverTypeFromPath()` - Guess the driver type used by a file with the given pathname

C Signature

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

Fortran Signature:

```
None
```

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

`DBInqFile()` - Inquire if filename is a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description
`filename`	Name of file.

`DBInqFileHasObjects()` - Determine if an open file has any Silo objects

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	The Silo database file handle

`_silolibinfo()` - character array written by Silo to root directory indicating the Silo library version number used to generate the file

C Signature

```
int n;
    char vers[1024];
    sprintf(vers, “silo-4.6”);
    n = strlen(vers);
    DBWrite(dbfile, “_silolibinfo”, vers, &n, 1, DB_CHAR);
    Description:
    This is a simple array variable written at the root directory in a
Silo file that contains the Silo library version string. It cannot be
disabled.
    _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 157.
    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...

```
    DBGetCSGZonelist    136
    DBPutCsgvar 137
    DBGetCsgvar 139
    DBPutMaterial       140
    DBGetMaterial       144
    DBPutMatspecies     145
    DBGetMatspecies     148
    DBPutDefvars        149
    DBGetDefvars        151
    DBInqMeshname       152
    DBInqMeshtype       153
    DBPutCurve
    —Write a curve object into a Silo file
    Synopsis:
    int DBPutCurve (DBfile *dbfile, char const *curvename,
    void const *xvals, void const *yvals, int datatype,
    int npoints, DBoptlist const *optlist)
```

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer
`curvename`	Name of the curve object
`xvals`	Array of length npoints containing the x-axis data values. Must be NULL when either DBOPT_XVARNAME or DBOPT_REFERENCE is used.
`yvals`	Array of length npoints containing the y-axis data values. Must be NULL when either DBOPT_YVARNAME or DBOPT_REFERENCE is used.
`datatype`	Data type of the xvals and yvals arrays. One of the predefined Silo types.
`npoints`	The number of points in the curve
	Pointer to an option list structure containing additional

`optlist` information to be included in the compound array object written into the Silo file. Use NULL is there are no options.

`_hdf5libinfo()` - character array written by Silo to root directory indicating the HDF5 library version number used to generate the file

C Signature

```
int n;
    char vers[1024];
    sprintf(vers, “hdf5-1.6.6”);
    n = strlen(vers);
    DBWrite(dbfile, “_hdf5libinfo”, vers, &n, 1, DB_CHAR);
    Description:
    This is a simple array variable written at the root directory in a
Silo file that contains the HDF5 library version string. It cannot be
disabled. Of course, it exists, only in files created with the HDF5
driver.
    _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 157.
    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...

```
DBPutMaterial        140
DBGetMaterial        144
DBPutMatspecies      145
DBGetMatspecies      148
DBPutDefvars         149
DBGetDefvars         151
DBInqMeshname        152
DBInqMeshtype        153
DBPutCurve
—Write a curve object into a Silo file
Synopsis:
int DBPutCurve (DBfile *dbfile, char const *curvename,
void const *xvals, void const *yvals, int datatype,
int npoints, DBoptlist const *optlist)
```

Fortran Signature

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

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

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

C Signature

```
int n=1;
    DBWrite(dbfile, "_was_grabbed", &n, &n, 1, DB_INT);
    Description:
    This is a simple array variable written at the root directory in a
Silo whenever a Silo file has been grabbed by the DBGrabDriver()
function. It cannot be disabled.
    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 157.
    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   77
    DBGetCurve   79
    DBPutPointmesh      80
    DBGetPointmesh      83
    DBPutPointvar       84
    DBPutPointvar1      86
    DBGetPointvar       87
```

```
DBPutQuadmesh       88
DBGetQuadmesh       91
DBPutQuadvar        92
DBPutQuadvar1       96
DBGetQuadvar        98
DBPutUcdmesh        99
DBPutUcdsubmesh     107
DBGetUcdmesh        108
DBPutZonelist       109
DBPutZonelist2      110
DBPutPHZonelist     112
DBGetPHZonelist     116
DBPutFacelist       117
DBPutUcdvar 119
DBPutUcdvar1        122
DBGetUcdvar 124
DBPutCsgmesh        125
DBGetCsgmesh        130
DBPutCSGZonelist    131
DBGetCSGZonelist    136
DBPutCsgvar 137
DBGetCsgvar 139
DBPutMaterial       140
DBGetMaterial       144
DBPutMatspecies     145
DBGetMatspecies     148
DBPutDefvars        149
DBGetDefvars        151
DBInqMeshname       152
DBInqMeshtype       153
DBPutCurve
—Write a curve object into a Silo file
Synopsis:
int DBPutCurve (DBfile *dbfile, char const *curvename,
void const *xvals, void const *yvals, int datatype,
int npoints, DBoptlist const *optlist)
```

Fortran Signature

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

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

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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 157.

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

C Signature

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

Fortran Signature

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

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

`DBGetCurve()` - Read a curve from a Silo database.

C Signature

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

Fortran Signature

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

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

`DBPutPointmesh()` - Write a point mesh object into a Silo file.

C Signature

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

Fortran Signature

```
integer function dbputpm(dbid, name, lname, ndims,
   x, y, z, nels, datatype, optlist_id,
   status)
void* x, y, z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
```

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

DBGetPointmesh() - Read a point mesh from a Silo database.

C Signature

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

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

DBPutPointvar() - Write a vector/tensor point variable object into a Silo file.

C Signature

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

Fortran Signature

```
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable set.
meshname	Name of the associated point mesh.
nvars	Number of variables supplied in vars array.
vars	Array of length nvars containing pointers to value arrays.
nels	Number of elements (points) in variable.
datatype	Datatype of the value arrays. One of the predefined Silo data types.
	Pointer to an option list structure containing additional

`optlist`	information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

`DBPutPointvar1()` - Write a scalar point variable object into a Silo file.

C Signature

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

Fortran Signature

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

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

`DBGetPointvar()` - Read a point variable from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutQuadmesh()` - Write a quad mesh object into a Silo file.

C Signature

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

Fortran Signature

```
integer function dbputqm(dbid, name, lname, xname,
   lxname, yname, lyname, zname, lzname, x,
   y, z, dims, ndims, datatype, coordtype,
   optlist_id, status)
void* x, y, z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
character* xname, yname, zname (if ndims<3, zname=0 ok, etc.)
```

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the mesh.
`coordnames`	Array of length ndims containing pointers to the names to be provided when writing out the coordinate arrays. This parameter is currently ignored and can be set as NULL.
`coords`	Array of length ndims containing pointers to the coordinate arrays.

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

`DBGetQuadmesh()` - Read a quadrilateral mesh from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

DBPutQuadvar() - Write a vector/tensor quad variable object into a Silo file.

C Signature

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

Fortran Signature

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

varnames contains the names of the variables either in a matrix of
characters form (if fortran2DStrLen is non null) or in a vector of
characters form (if fortran2DStrLen is null) with the varnames length
being found in the lvarnames integer array,
var is essentially a matrix of size <nvars> x <var-size> where var-size
is determined by dims and ndims. The first “row” of the var matrix is
the first component of the quadvar. The second “row” of the var matrix
goes out as the second component of the quadvar, etc.
```

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the variable.
`meshname`	Name of the mesh associated with this variable (written with DBPutQuadmesh or DBPutUcdmesh). If no association is to be made, this value should be NULL.
`nvars`	Number of sub-variables which comprise this variable. For a scalar array, this is one. If writing a vector quantity, however, this would be two for a 2-D vector and three for a

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

into the Silo file. Typically, this argument is NULL.

`DBPutQuadvar1()` - Write a scalar quad variable object into a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the variable.
`meshname`	Name of the mesh associated with this variable (written with DBPutQuadmesh or DBPutUcdmesh.) If no association is to be made, this value should be NULL.
`var`	Array defining the values associated with this variable. For true edge- or face-centering (as opposed to DB_EDGECENT centering when ndims is 1 and DB_FACECENT centering when ndims is 2), each pointer here should point to an array that holds ndims sub-arrays, one for each of the i-, j-, k-oriented edges or i-, j-, k-intercepting faces, respectively. Read the description for DBPutQuadvar more details.
`dims`	Array of length ndims which describes the dimensionality of the data stored in the var array. For DB_NODECENT centering, this array holds the number of nodes in each dimension. For DB_ZONECENT centering, DB_EDGECENT centering

	when ndims is 1 and DB_FACECENT centering when ndims is 2, this array holds the number of zones in each dimension. Otherwise, for DB_EDGECENT and DB_FACECENT centering, this array should hold the number of nodes in each dimension.
`ndims`	Number of dimensions.
`mixvar`	Array defining the mixed-data values associated with this variable. If no mixed values are present, this should be NULL.
`mixlen`	Length of mixed data arrays, if provided.
`datatype`	Datatype of sub-variables. One of the predefined Silo data types.
`centering`	Centering of the subvariables on the associated mesh. One of the predefined types: DB_NODECENT, DB_EDGECENT, DB_FACECENT or DB_ZONECENT. Note that DB_EDGECENT centering on a 1D mesh is treated identically to DB_ZONECENT centering. Likewise for DB_FACECENT centering on a 2D mesh.
`optlist`	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. Typically, this argument is NULL.

`DBGetQuadvar()` - Read a quadrilateral variable from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

DBPutUcdmesh() - Write a UCD mesh object into a Silo file.

C Signature

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

Fortran Signature

```
integer function dbputum(dbid, name, lname, ndims,
   x, y, z, xname, lxname, yname,
   lyname, zname, lzname, datatype, nnodes nzones, zonel_name,
   lzonel_name, facel_name, lfacel_name, optlist_id, status)
void *x,y,z (if ndims<3, z=0 ok, if ndims<2, y=0 ok)
character* xname,yname,zname (same rules)
```

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the mesh.
`ndims`	Number of spatial dimensions represented by this UCD mesh.
`coordnames`	Array of length ndims containing pointers to the names to be provided when writing out the coordinate arrays. This parameter is currently ignored and can be set as NULL.
`coords`	Array of length ndims containing pointers to the coordinate arrays.
`nnodes`	Number of nodes in this UCD mesh.
`nzones`	Number of zones in this UCD mesh.
	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

`zonel_name`	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.

`DBPutUcdsubmesh()` - Write a subset of a parent, ucd mesh, to a Silo file

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`file`	The Silo database file handle.
`name`	The name of the ucd submesh object to create.
`parentmesh`	The name of the parent ucd mesh this submesh is a portion

	of.
`nzones`	The number of zones in this submesh.
`zlname`	The name of the zonelist object.
`fl`	[OPT] The name of the facelist object.
`opts`	Additional options.

`DBGetUcdmesh()` - Read a UCD mesh from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutZonelist()` - Write a zonelist object into a Silo file.

C Signature

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

Fortran Signature

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

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

`DBPutZonelist2()` - Write a zonelist object containing ghost zones into a Silo file.

C Signature

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

Fortran Signature

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

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

`DBPutPHZonelist()` - Write an arbitrary, polyhedral zonelist object into a Silo file.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the zonelist structure.
`nfaces`	Number of faces in the zonelist. Note that faces shared between zones should only be counted once.
`nodecnts`	Array of length nfaces indicating the number of nodes in each face. That is nodecnts[i] is the number of nodes in face i.
`lnodelist`	Length of the succeeding nodelist array.
`nodelist`	Array of length lnodelist listing the nodes of each face. The list of nodes for face i begins at index Sum(nodecnts[j]) for j=0…i-1.
`extface`	An optional array of length nfaces where extface[i]!=0x0 means that face i is an external face. This argument may be NULL.
`nzones`	Number of zones in the zonelist.
`facecnts`	Array of length nzones where facecnts[i] is number of faces for zone i.
`lfacelist`	Length of the succeeding facelist array.
	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

`facelist`	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.

`DBGetPHZonelist()` - Read a polyhedral-zonelist from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutFacelist()` - Write a facelist object into a Silo file.

C Signature

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

Fortran Signature

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

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

types	Array of length nfaces containing information about each face. This argument is ignored if ntypes is zero, or if this parameter is NULL.
typelist	Array of length ntypes containing the identifiers for each type. This argument is ignored if ntypes is zero, or if this parameter is NULL.
ntypes	Number of types, or zero if type information was not provided.

DBPutUcdvar() - Write a vector/tensor UCD variable object into a Silo file.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the variable.
meshname	Name of the mesh associated with this variable (written with DBPutUcdmesh).
nvars	Number of sub-variables which comprise this variable. For a scalar array, this is one. If writing a vector quantity, however, this would be two for a 2-D vector and three for a 3-D vector.
varnames	Array of length nvars containing pointers to character strings defining the names associated with each

	subvariable.
`vars`	Array of length nvars containing pointers to arrays defining the values associated with each subvariable.
`nels`	Number of elements in this variable.
`mixvars`	Array of length nvars containing pointers to arrays defining the mixed-data values associated with each subvariable. If no mixed values are present, this should be NULL.
`mixlen`	Length of mixed data arrays (i.e., mixvars).
`datatype`	Datatype of sub-variables. One of the predefined Silo data types.
`centering`	Centering of the sub-variables on the associated mesh. One of the predefined types: DB_NODECENT, DB_EDGECENT, DB_FACECENT, DB_ZONECENT or DB_BLOCKCENT. See below for a discussion of centering issues.
`optlist`	Pointer to an option list structure containing additional information to be included in the variable object written into the Silo file. See the table below for the valid options for this function. If no options are to be provided, use NULL for this argument.

`DBPutUcdvar1()` - Write a scalar UCD variable object into a Silo file.

C Signature

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

Fortran Signature

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

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

`DBGetUcdvar()` - Read a UCD variable from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutCsgmesh()` - Write a CSG mesh object to a Silo file

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer
`name`	Name to associate with this DBcsgmesh object
`ndims`	Number of spatial and topological dimensions of the CSG mesh object
`nbounds`	Number of boundaries in the CSG mesh description.
`typeflags`	Integer array of length nbounds of type information for each boundary. This is used to encode various information about the type of each boundary such as, for example, plane, sphere, cone, general quadric, etc as well as the number of coefficients in the representation of the boundary. For more information, see the description, below.

`bndids`	Optional integer array of length nbounds which are the explicit integer identifiers for each boundary. It is these identifiers that are used in expressions defining a region of the CSG mesh. If the caller passes NULL for this argument, a natural numbering of boundaries is assumed. That is, the boundary occurring at position i, starting from zero, in the list of boundaries here is identified by the integer i.
`coeffs`	Array of length lcoeffs of coefficients used in the representation of each boundary or, if the boundary is a transformed copy of another boundary, the coefficients of the transformation. In the case where a given boundary is a transformation of another boundary, the first entry in the coeffs entries for the boundary is the (integer) identifier for the referenced boundary. Consequently, if the datatype for coeffs 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.

`DBGetCsgmesh()` - Get a CSG mesh object from a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutCSGZonelist()` - Put a CSG zonelist object in a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer
`name`	Name to associate with the DBcsgzonelist object
`nregs`	The number of regions in the regionlist.
`typeflags`	Integer array of length nregs of type information for each region. Each entry in this array is one of either DB_INNER, DB_OUTER, DB_ON, DB_XFORM, DB_SWEEP, DB_UNION, DB_INTERSECT, and DB_DIFF.
`The symbols, DB_INNER, DB_OUTER,`	For the unary operators, DB_INNER forms a region

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.	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.
	Integer array of length nregs of references to

`rightids`	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.

`DBGetCSGZonelist()` - Read a CSG mesh zonelist from a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutCsgvar()` - Write a CSG mesh variable to a Silo file

C Signature

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

Fortran Signature

```
integer function dbputcsgv(dbid, vname, lvname, meshname,
   lmeshname, nvars, var_ids, nvals, datatype, centering,
   optlist_id, status)
integer* var_ids (array of “pointer ids” created using dbmkptr)
```

Arg name	Description
`dbfile`	Database file pointer
`vname`	The name to be associated with this DBcsgvar object
`meshname`	The name of the CSG mesh this variable is associated with
`nvars`	The number of subvariables comprising this CSG variable
`varnames`	Array of length nvars containing the names of the

	subvariables
`vars`	Array of pointers to variable data
`nvals`	Number of values in each of the vars arrays
`datatype`	The type of data in the vars arrays (e.g. DB_FLOAT, DB_DOUBLE)
`centering`	The centering of the CSG variable (DB_ZONECENT 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

`DBGetCsgvar()` - Read a CSG mesh variable from a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutMaterial()` - Write a material data object into a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the material data object.
`meshname`	Name of the mesh associated with this information.
`nmat`	Number of materials.
`matnos`	Array of length nmat containing material numbers.
`matlist`	Array whose dimensions are defined by dims and ndims. It contains the material numbers for each single-material (non-mixed) zone, and indices into the mixed data arrays for each multi-material (mixed) zone. A negative value indicates a mixed zone, and its absolute value is used as an index into the mixed data arrays.
`dims`	Array of length ndims which defines the dimensionality of the matlist array.
`ndims`	Number of dimensions in matlist array.
`mix_next`	Array of length mixlen of indices into the mixed data arrays (one-origin).
`mix_mat`	Array of length mixlen of material numbers for the mixed zones.
`mix_zone`	Optional array of length mixlen of back pointers to originating zones. The origin is determined by DBOPT_ORIGIN. Even if mixlen > 0, this argument is optional.
`mix_vf`	Array of length mixlen of volume fractions for the mixed

	zones. Note, this can actually be either single- or double-precision. Specify actual type in datatype.
`mixlen`	Length of mixed data arrays (or zero if no mixed data is present). If mixlen > 0, then the “mix_” arguments describing the mixed data arrays must be 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 provided, use NULL for this argument.

`DBGetMaterial()` - Read material data from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutMatspecies()` - Write a material species data object into a Silo file.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the material species data object.
`matname`	Name of the material object with which the material species object is associated.
`nmat`	Number of materials in the material object referenced by matname.
`nmatspec`	Array of length nmat containing the number of species associated with each material.
`speclist`	Array of dimension defined by ndims and dims of indices into the species_mf array. Each entry corresponds to one zone. If the zone is clean, the entry in this array must be positive or zero. A positive value is a 1-origin index into the species_mf array. A zero can be used if the material in this zone contains only one species. If the zone is mixed, this value is negative and is used to index into the mix_spec array in a manner analogous to the mix_mat array of the DBPutMaterial() call.
`dims`	Array of length ndims that defines the shape of the speclist array. To create an empty matspecies object, set every entry of dims to zero. See description below.
`ndims`	Number of dimensions in the speclist array.
`nspecies_mf`	Length of the species_mf array. To create a homogeneous matspecies object (which is not quite empty), set

	nspecies_mf to zero. See description below.
species_mf	Array of length nspecies_mf containing mass fractions of the material species. Note, this can actually be either single or double precision. Specify type in datatype argument.
mix_spec	Array of length mixlen containing indices into the species_mf array. These are used for mixed zones. For every index j in this array, mix_list[j] corresponds to the DBmaterial structure's material mix_mat[j] and zone mix_zone[j].
mixlen	Length of the mix_spec array.
datatype	The datatype of the mass fraction data in species_mf. One of the predefined Silo data types.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

DBGetMatspecies() - Read material species data from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

DBPutDefvars() - Write a derived variable definition(s) object

into a Silo file.

C Signature

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

Fortran Signature

```
integer function dbputdefvars(dbid, name, lname, ndefs,
   names, lnames, types, defns, ldefns, optlist_id,
   status)
character*N names (See “dbset2dstrlen” on page 288.)
character*N defns (See “dbset2dstrlen” on page 288.)
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the derived variable definition(s) object.
ndefs	number of derived variable definitions.
names	Array of length ndefs of derived variable names
types	Array of length ndefs of derived variable types 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.

DBGetDefvars() - Get a derived variables definition object from a Silo file.

C Signature

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

Fortran Signature:

```
None
```

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

`DBInqMeshname()` - Inquire the mesh name associated with a variable.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Variable name.
`meshname`	Returned mesh name. The caller must allocate space for the returned name. The maximum space used is 256 characters, including the NULL terminator.

`DBInqMeshtype()` - Inquire the mesh type of a mesh.

C Signature

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

Fortran Signature:

```
None
```

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

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.

C Signature

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

Fortran Signature

```
integer function dbputmmesh(dbid, name, lname, nmesh,
   meshnames, lmeshnames, meshtypes, optlist_id, status)
character*N meshnames (See “dbset2dstrlen” on page 288.)
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-block mesh object.
nmesh	Number of meshes pieces (blocks) in this multi-block object.
meshnames	Array of length nmesh containing pointers to the names of each of the mesh blocks written with a DBPutmesh() call. See below for description of how to populate meshnames when the pieces are in different files as well as DBOPT_MB_FILE/BLOCK_NS options to use a printf-style namescheme for large nmesh in lieu of explicitly enumerating them here.
meshtypes	Array of length nmesh containing the type of each mesh block such as DB_QUAD_RECT, DB_QUAD_CURV, DB_UCDMESH, DB_POINTMESH, and DB_CSGMESH. Be sure to see description, below, for DBOPT_MB_BLOCK_TYPE option to use single, constant value when all pieces are the same type.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the

Silo file. Use a NULL if there are no options.

DBGetMultimesh() - Read a multi-block mesh from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

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

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-mesh adjacency object.

nmesh	The number of mesh pieces in the corresponding multi-mesh object. This value must be identical in repeated calls to DBPutMultimeshadj.
mesh_types	Integer array of length nmesh indicating the type of each mesh in the corresponding multi-mesh object. This array must be identical to that which is passed in the DBPutMultimesh call and in repeated calls to DBPutMultimeshadj.
nneighbors	Integer array of length nmesh indicating the number of neighbors for each mesh piece. This array must be identical in repeated calls to DBPutMultimeshadj. In the argument descriptions to follow, let . That is, let be the sum of the first k entries in the nneighbors array.
neighbors	Array of integers enumerating for each mesh piece all other mesh pieces that neighbor it. Entries from index to index enumerate the neighbors of mesh piece k. This array must be identical in repeated calls to DBPutMultimeshadj.
back	Array of integers enumerating for each mesh piece, the local index of that mesh piece in each of its neighbors lists of neighbors. Entries from index to index enumerate the local indices of mesh piece k in each of the neighbors of mesh piece k. This argument may be NULL. In any case, this array must be identical in repeated calls to DBPutMultimeshadj.
nnodes	Array of integers indicating for each mesh piece, the number of nodes that it shares with each of its neighbors. Entries from index to index indicate the number of nodes that mesh piece k shares with each of its neighbors. This array must be identical in repeated calls to DBPutMultimeshadj. This argument may be NULL.
nodelists	Array of pointers to arrays of integers. Entries from index to index enumerate the nodes that mesh piece k shares with each of its neighbors. The contents of a specific nodelist array depend on the types of meshes that are neighboring each other (See description below). nodelists[m] may be NULL even if nnodes[m] is non-zero. See below for a description of repeated calls to

	DBPutMultimeshadj. This argument must be NULL if nnodes is NULL.
nzones	Array of integers indicating for each mesh piece, the number of zones that are adjacent with each of its neighbors. Entries from index to index indicate the number of zones that mesh piece k has adjacent to each of its neighbors. This array must be identical in repeated calls to DBPutMultimeshadj. This argument may be NULL.
zonelists	Array of pointers to arrays of integers. Entries from index to index enumerate the zones that mesh piece k has adjacent with each of its neighbors. The contents of a specific zonelist array depend on the types of meshes that are neighboring each other (See description below). zonelists[m] may be NULL even if nzones[m] is non-zero. See below for a description of repeated calls to DBPutMultimeshadj. This argument must be NULL if nzones is NULL.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

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

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
dbfile	Database file pointer

name	Name of the multi-mesh nodal adjacency object
nmesh	Number of mesh pieces for which nodal adjacency information is being obtained. Pass zero if you want to obtain all nodal adjacency information in a single call.
mesh_pieces	Integer array of length nmesh indicating which mesh pieces nodal adjacency information is desired for. May pass NULL if nmesh is zero.

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

C Signature

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

Fortran Signature

```
integer function dbputmvar(dbid, name, lname, nvar,
   varnames, lvarnames, vartypes, optlist_id, status)
character*N varnames (See “dbset2dstrlen” on page 288.)
```

Arg name	Description
dbfile	Database file pointer.
name	Name of the multi-block variable.
nvar	Number of variables associated with the multi-block variable.
varnames	Array of length nvar containing pointers to the names of the variables written with DBPutvar() call. See “DBPutMultimesh” on page 2-159 for description of how to populate varnames when the pieces are in different files as well as DBOPT_MB_BLOCK/FILE_NS options to use a printf-style namescheme for large nvar in lieu of explicitly enumerating them here.

vartypes	Array of length nvar containing the types of the variables such as DB_POINTVAR, DB_QUADVAR, or DB_UCDVAR. See “DBPutMultimesh” on page 2-159, for DBOPT_MB_BLOCK_TYPE option to use single, constant value when all pieces are the same type.
optlist	Pointer to an option list structure containing additional information to be included in the object written into the Silo file. Use a NULL if there are no options.

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

C Signature

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

Fortran Signature:

```
None
```

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

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

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the multi-material object.
`nmat`	Number of material blocks provided.
`matnames`	Array of length nmat containing pointers to the names of the material block objects, written with DBPutMaterial(). See “DBPutMultimesh” on page 2-159 for description of how to populate matnames when the pieces are in different files as well as DBOPT_MB_BLOCK/FILE_NS options to use a printf-style namescheme for large nmat in lieu of explicitly enumerating them here.
`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

`DBGetMultimat()` - Read a multi-block material object from a Silo database

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutMultimatspecies()` - Write a multi-block species object into a Silo file.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the multi-block species structure.
`nspec`	Number of species objects provided.
`specnames`	Array of length nspec containing pointers to the names of each of the species. See “DBPutMultimesh” on page 2-159 for description of how to populate specnames when the pieces are in different files as well as DBOPT_MB_BLOCK/FILE_NS options to use a printf-style namescheme for large nspec in lieu of explicitly enumerating them here.
`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.

`DBGetMultimatspecies()` - Read a multi-block species from a Silo database.

C Signature

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

Fortran Signature:

```
None
```

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

`DBOpenByBcast()` - Specialized, read-only open method for parallel applications needing all processors to read all (or most of) a given Silo file

C Signature

```
DBfile *DBOpenByBcast(char const *filename, MPI_Comm comm,
    int rank_of_root)
```

Fortran Signature:

```
None
```

Arg name	Description
`filename`	name of the Silo file to open
`comm`	MPI communicator to use for the broadcast operation
`rank_of_root`	MPI rank of the processor in the communicator comm that shall serve as the root of the broadcast (typically 0).

`PMPIO_Init()` - Initialize a Poor Man's Parallel I/O interaction with the Silo library

C Signature

```
PMPIO_baton_t *PMPIO_Init(int numFiles, PMPIO_iomode_t ioMode,
    MPI_Comm mpiComm, int mpiTag,
    PMPIO_CreateFileCallBack createCb,
    PMPIO_OpenFileCallBack openCb,
    PMPIO_CloseFileCallBack closeCB,
    void *userData)
```

Fortran Signature:

```
None
```

Arg name	Description
`numFiles`	The number of individual Silo files to generate. Note, this is the number of parallel I/O streams that will be running simultaneously during I/O. A value of 1 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.

`PMPIO_CreateFileCallBack()` - The PMPIO file creation callback

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`fname`	The name of the Silo file to create.
`dname`	The name of the directory within the Silo file to create.
`udata`	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

`PMPIO_OpenFileCallBack()` - The PMPIO file open callback

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description

Arg name	Description
fname	The name of the Silo file to open.
dname	The name of the directory within the Silo file to work in.
iomode	The iomode of this PMPIO interaction. This is the value passed as ioMode argument to PMPIO_Init().
udate	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

PMPIO_CloseFileCallBack() - The PMPIO file close callback

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
file	void pointer to the file handle (DBfile pointer).
udata	A pointer to any additional user data. This is the pointer passed as the userData argument to PMPIO_Init().

PMPIO_WaitForBaton() - Wait for exclusive access to a Silo file

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`bat`	The PMPIO baton handle obtained via a call to PMPIO_Init().
`filename`	The name of the Silo file this processor will create or open.
`dirname`	The name of the directory within the Silo file this processor will work in.

`PMPIO_HandOffBaton()` - Give up all access to a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`PMPIO_Finish()` - Finish a Poor Man’s Parallel I/O interaction with the Silo library

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description

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

PMPIO_GroupRank() - Obtain 'group rank' of a processor

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
rankInComm	Rank of processor in the MPI communicator passed in PMPIO_Init() for which group rank is to be queried.

PMPIO_RankInGroup() - Obtain the rank of a processor within its PMPIO group

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
bat	The PMPIO baton handle obtained via a call to PMPIO_Init().
rankInComm	Rank of the processor in the MPI communicator used in PMPIO_Init() to be queried.

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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

C Signature

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

Fortran Signature

```
integer function dbmkmrgtree(mesh_type, info_bits, max_children,
optlist_id,
   tree_id)
returns handle to newly created tree in tree_id.
```

Arg name	Description
`mesh_type`	The type of mesh object the MRG tree will be associated with. An example would be DB_MULTIMESH, DB_QUADMESH, DB_UCDMESH.
`info_bits`	UNUSED
`max_children`	Maximum number of immediate children of the root.
`opts`	Additional options

`DBAddRegion()` - Add a region to an MRG tree

C Signature

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

Fortran Signature

```
integer function dbaddregion(tree_id, reg_name, lregname, info_bits,
   max_children, maps_name, lmaps_name, nsegs, seg_ids, seg_lens,
   seg_types, optlist_id, status)
```

Arg name	Description
`tree`	The MRG tree object to add a region to.
`reg_name`	The name of the new region.
`info_bits`	UNUSED
`max_children`	Maximum number of immediate children this region will have.
`maps_name`	[OPT] Name of the groupel map object to associate with this region. Pass NULL if none.
`nsegs`	[OPT] Number of segments in the groupel map object specified by the maps_name argument that are to be associated with this region. Pass zero if none.
`seg_ids`	[OPT] Integer array of length nsegs of groupel map segment ids. Pass NULL (0) if none.
`seg_lens`	[OPT] Integer array of length nsegs of groupel map segment lengths. Pass NULL (0) if none.
`seg_types`	[OPT] Integer array of length nsegs of groupel map segment element types. Pass NULL (0) if none. These types are the same as the centering options for variables; DB_ZONECENT, DB_NODECENT, DB_EDGECENT, DB_FACECENT and DB_BLOCKCENT (for the blocks of a multimesh)
`opts`	[OPT] Additional options. Pass NULL (0) if none.

`DBAddRegionArray()` - Efficiently add multiple, like-kind regions to an MRG tree

C Signature

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

Fortran Signature

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

Arg name	Description
tree	The MRG tree object to add the regions to.
nregn	The number of regions to add.
regn_names	This is either an array of nregn pointers to character string names for each region or it is an array of 1 pointer to a character string specifying a printf-style naming scheme for the regions. The existence of a percent character (‘%’) (used to introduce conversion specifications) anywhere in regn_names[0] will indicate the latter mode.The latter mode is almost always preferable, especially if nergn is large (say more than 100). See below for the format of the printf-style naming string.
info_bits	UNUSED
maps_name	[OPT] Name of the groupel maps object to be associated with these regions. Pass NULL (0) if none.
nsegs	[OPT] The number of groupel map segments to be associated with each region. Note, this is a per-region value. Pass 0 if none.
	[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

`seg_ids`	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.

`DBSetCwr()` - Set the current working region for an MRG tree

C Signature

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

Fortran Signature

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

Arg name	Description
`tree`	The MRG tree object.
`path`	The path to set.

`DBGetCwr()` - Get the current working region of an MRG tree

C Signature

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

Arg name	Description
`tree`	The MRG tree.

`DBPutMrgtree()` - Write a completed MRG tree object to a Silo file

C Signature

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

Fortran Signature

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

Arg name	Description
`file`	The Silo file handle
`name`	The name of the MRG tree object in the file.
`mesh_name`	The name of the mesh the MRG tree object is associated with.
`tree`	The MRG tree object to write.
`opts`	[OPT] Additional options. Pass NULL (0) if none.

`DBGetMrgtree()` - Read an MRG tree object from a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBFreeMrgtree()` - Free the memory associated by an MRG tree object

C Signature

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

Fortran Signature

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

Arg name	Description
`tree`	The MRG tree object to free.

`DBMakeNamescheme()` - Create a DBnamescheme object for on-demand name generation

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`ns_str`	The namescheme string as described below. The remaining arguments take one of three forms depending on how the caller wants external array references, if any are

... present in the format substring of ns_str to be handled. In the first form, the format substring of ns_str involves no externally referenced arrays and so there are no additional arguments other than the ns_str string itself. In the second form, the caller has all externally referenced arrays needed in the format substring of ns_str already in memory and simply passes their pointers here as the remaining arguments in the same order in which they appear in the format substring of ns_str. The arrays are bound to the returned namescheme object and should not be freed until after the caller is done using the returned namescheme object. In this case, DBFreeNamescheme() does not free these arrays and the caller is required to explicitly free them. In the third form, the caller makes a request for the Silo library to find in a given file, read and bind to the returned namescheme object all externally referenced arrays in the format substring of ns_str. To achieve this, the caller passes a 3-tuple of the form… "(void) *0, (DBfile*) file, (char) *mbobjpath" as the remaining arguments. The initial (void*)0 is required. The (DBfile*)*file is the database handle of the Silo file in which all externally referenced arrays exist. The third (char*)*mbobjpath, which may be 0/NULL, is the path within the file, either relative to the file's current working directory, or absolute, at which the multi-block object holding the ns_str was found in the file. All necessary externally referenced arrays must exist within the specified file using either relative paths from multi-block object's home directory or the file's current working directory or absolute paths. In this case DBFreeNamescheme() also frees memory associated with these arrays.

DBGetName() - Generate a name from a DBnamescheme object

C Signature

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

Fortran Signature:

```
None
```

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

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

C Signature

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

Fortran Signature

```
integer function dbputmrgv(dbid, name, lname, mrgt_name,
   lmrgt_name, ncomps, compnames, lcompnames, nregns, reg_names,
   lreg_names, datatype, data_ids, optlist_id, status)
character*N compnames (See “dbset2dstrlen” on page 288.)
character*N reg_names (See “dbset2dstrlen” on page 288.)
int* data_ids (use dbmkptr to get id for each pointer)
```

Arg name	Description
file	Silo database file handle.
name	Name of this mrgvar object.
tname	name of the mrg tree this variable is associated with.
ncomps	An integer specifying the number of variable components.
compnames	[OPT] Array of ncomps pointers to character strings representing the names of the individual components. Pass NULL(0) if no component names are to be specified.

`nregns`	The number of regions this variable is being written for.
`reg_pnames`	Array of nregns pointers to strings representing the pathnames of the regions for which the variable is being written. If nregns>1 and reg_pnames[1]==NULL, it is assumed that reg_pnames[i]=NULL for all i>0 and reg_pnames[0] contains either a printf-style naming convention for all the regions to be named or, if reg_pnames[0] is found to contain no printf-style conversion specifications, it is treated as the pathname of a single region in the MRG tree that is the parent of all the regions for which attributes are being written.
`data`	Array of ncomps pointers to variable data. The pointer, data[i] points to an array of nregns values of type datatype.
`opts`	Additional options.

`DBGetMrgvar()` - Retrieve an MRG variable object from a silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutGroupelmap()` - Write a groupel map object to a Silo file

C Signature

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

Fortran Signature

```
integer function dbputgrplmap(dbid, name, lname, num_segs,
   seg_types, seg_lens, seg_ids, seg_data_ids, seg_fracs_ids,
fracs_type,
   optlist_id, status)
integer* seg_data_ids (use dbmkptr to get id for each pointer)
integer* seg_fracs_ids (use dbmkptr to get id for each pointer)
```

Arg name	Description
`file`	The Silo database file handle.
`name`	The name of the groupel map object in the file.
`nsegs`	The number of segments in the map.
`seg_types`	Integer array of length nsegs indicating the groupel type associated with each segment of the map; one of DB_BLOCKCENT, DB_NODECENT, DB_ZONECENT, DB_EDGECENT, DB_FACECENT.
`seg_lens`	Integer array of length nsegs indicating the length of each segment
`seg_ids`	[OPT] Integer array of length nsegs indicating the identifier to associate with each segment. By default, segment identifiers are 0…negs-1. If default identifiers are sufficient, pass NULL (0) here. Otherwise, pass an explicit list of integer identifiers.
`seg_data`	The groupel map data, itself. An array of nsegs pointers to arrays of integers where array seg_data[i] is of length seg_lens[i].
	[OPT] Array of nsegs pointers to floating point values indicating fractional inclusion for the associated groupels. Pass NULL (0) if fractional inclusions are not

`seg_fracs`	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

`DBGetGroupelmap()` - Read a groupel map object from a Silo file

C Signature

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

Fortran Signature:

```
None
```

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

`DBFreeGroupelmap()` - Free memory associated with a groupel map object

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`map`	Pointer to a DBgroupel map object.

`DBOPT_REGION_PNAMES()` - option for defining variables on specific regions of a mesh

C Signature

```
DBOPT_REGION_PNAMES
    char**
    A null-pointer terminated array of pointers to strings specifying
the pathnames of regions in the mrg tree for the associated mesh where
the variable is defined. If there is no mrg tree associated with the
mesh, the names specified here will be assumed to be material names of
the material object associated with the mesh. The last pointer in the
array must be null and is used to indicate the end of the list of
names.
    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 material(s). In this case, the ordering convention requires further explanation and is described below.

For DB_ZONECENT variables, as one traverses the zones of a mesh from the first zone to the last, if a zone contains a material listed in DBOPT_REGION_PNAMES (wholly or partially), that zone is considered in the traversal and placed conceptually in an ordered list of traversed zones. In addition, if the zone contains the material only partially, that zone is also placed conceptually in an ordered list of traversed mixed zones. In this case, the values in the vars[] array must be one-to-one with this traversed zones list. Likewise, the values of the mixvars[] array must be one-to-one with the traversed mixed zones list. However, in the special case that the list of materials specified in DBOPT_REGION_PNAMES is of size one (1), an additional optimization is supported.

For the special case that the list of materials defined in DBOPT_REGION_PNAMES is of size one (1), the requirement to specify separate values for zones containing the material only partially in the mixvars[] array is removed. In this case, if the mixlen arg is zero (0) in the cooresponding DBPutXXXvar() call, only the vars[] array, which is one-to-one with (all) traversed zones containing the material either cleanly or partially, will be used. The reason this works is that in the single material case, there is only ever one zonal variable value per zone regardless of whether the zone contains the material cleanly or partially.

For DB_NODECENT variables, the situation is complicated by the fact that materials are zone-centric but the variable being defined is node-centered. So, an additional level of local traversal over a zone’s nodes is required. In this case, as one traverses the zones of a mesh from the first zone to the last, if a zone contains a material listed in DBOPT_REGION_PNAMES (wholly or partially), then that zone’s nodes are traversed according to the ordering specified in “Node, edge and face ordering for zoo-type UCD zone shapes.” on page 2-104. 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 2-104.

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 2-94.

6 API Section Object Allocation, Free and IsEmpty

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

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)
     DBnamescheme      *DBAllocNamescheme(void);
     DBgroupelmap      *DBAllocGroupelmap(int, DBdatatype);
```

Fortran Signature:

```
None
```

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Object Allocation, Free and IsEmpty

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

DBAlloc… 221 DBFree… 222

DBAlloc…() - Allocate and initialize a Silo structure.

C Signature

```
DBcompoundarray  *DBAllocCompoundarray (void)
    DBcsgmesh        *DBAllocCsgmesh (void)
    DBcsgvar         *DBAllocCsgvar (void)
    DBcurve          *DBAllocCurve (void)
    DBcsgzonelist    *DBAllocCSGZonelist (void)
    DBdefvars        *DBAllocDefvars (void)
    DBedgelist       *DBAllocEdgelist (void)
    DBfacelist       *DBAllocFacelist (void)
    DBmaterial       *DBAllocMaterial (void)
    DBmatspecies     *DBAllocMatspecies (void)
    DBmeshvar        *DBAllocMeshvar (void)
    DBmultimat       *DBAllocMultimat (void)
    DBmultimatspecies *DBAllocMultimatspecies (void)
    DBmultimesh      *DBAllocMultimesh (void)
    DBmultimeshadj   *DBAllocMultimeshadj (void)
    DBmultivar       *DBAllocMultivar (void)
    DBpointmesh      *DBAllocPointmesh (void)
    DBquadmesh       *DBAllocQuadmesh (void)
    DBquadvar        *DBAllocQuadvar (void)
    DBucdmesh        *DBAllocUcdmesh (void)
    DBucdvar         *DBAllocUcdvar (void)
    DBzonelist       *DBAllocZonelist (void)
    DBphzonelist     *DBAllocPHZonelist (void)
    DBnamescheme     *DBAllocNamescheme(void);
    DBgroupelmap     *DBAllocGroupelmap(int, DBdatatype);
```

Fortran Signature:

```
None
```

DBFree…() - Release memory associated with a Silo structure.

C Signature

```
void DBFreeCompoundarray (DBcompoundarray *x)
    void DBFreeCsgmesh (DBcsgmesh *x)
    void DBFreeCsgvar (DBcsgvar *x)
    void DBFreeCSGZonelist (DBcsgzonelist *x)
    void DBFreeCurve(DBcurve *);
    void DBFreeDefvars (DBdefvars *x)
    void DBFreeEdgelist (DBedgelist *x)
    void DBFreeFacelist (DBfacelist *x)
    void DBFreeMaterial (DBmaterial *x)
    void DBFreeMatspecies (DBmatspecies *x)
    void DBFreeMeshvar (DBmeshvar *x)
    void DBFreeMultimesh (DBmultimesh *x)
    void DBFreeMultimeshadj (DBmultimeshadj *x)
    void DBFreeMultivar (DBmultivar *x)
    void DBFreeMultimat(DBmultimat *);
    void DBFreeMultimatspecies(DBmultimatspecies *);
    void DBFreePointmesh (DBpointmesh *x)
    void DBFreeQuadmesh (DBquadmesh *x)
    void DBFreeQuadvar (DBquadvar *x)
    void DBFreeUcdmesh (DBucdmesh *x)
    void DBFreeUcdvar (DBucdvar *x)
    void DBFreeZonelist (DBzonelist *x)
    void DBFreePHZonelist (DBphzonelist *x)
    void DBFreeNamescheme(DBnamescheme *);
    void DBFreeMrgvar(DBmrgvar *mrgv);
    void DBFreeMrgtree(DBmrgtree *tree);
    void DBFreeGroupelmap(DBgroupelmap *map);
```

Arg name	Description
x	A pointer to a structure which is to be freed. Its type must correspond to the type in the function name.
Fortran Equivalent:	None

DBIsEmpty() - Query a object returned from Silo for “emptiness”

C Signature

```
int     DBIsEmptyCurve(DBcurve const *curve);
    int     DBIsEmptyPointmesh(DBpointmesh const *msh);
    int     DBIsEmptyPointvar(DBpointvar const *var);
    int     DBIsEmptyMeshvar(DBmeshvar const *var);
    int     DBIsEmptyQuadmesh(DBquadmesh const *msh);
    int     DBIsEmptyQuadvar(DBquadvar const *var);
    int     DBIsEmptyUcdmesh(DBucdmesh const *msh);
    int     DBIsEmptyFacelist(DBfacelist const *fl);
    int     DBIsEmptyZonelist(DBzonelist const *zl);
    int     DBIsEmptyPHZonelist(DBphzonelist const *zl);
    int     DBIsEmptyUcdvar(DBucdvar const *var);
    int     DBIsEmptyCsgmesh(DBcsgmesh const *msh);
    int     DBIsEmptyCSGZonelist(DBcsgzonelist const *zl);
    int     DBIsEmptyCsgvar(DBcsgvar const *var);
    int     DBIsEmptyMaterial(DBmaterial const *mat);
    int     DBIsEmptyMatspecies(DBmatspecies const *spec);
```

Arg name	Description
x	Pointer to a silo object structure to be queried

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Calculational and Utility

This section of the API manual describes functions that can be used to compute things such as Facelists as well as utility functions to, for example, catentate an array of strings into a single string for simple output with DBWrite().

DBCalcExternalFacelist() - Calculate an external facelist for a UCD mesh.

C Signature

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

Fortran Signature

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

Arg name	Description
nodelist	Array of node indices describing mesh zones.
nnodes	Number of nodes in associated mesh.
origin	Origin for indices in the nodelist array. Should be zero

	or one.
`shapesize`	Array of length nshapes containing the number of nodes used by each zone shape.
`shapecnt`	Array of length nshapes containing the number of zones having each shape.
`nshapes`	Number of zone shapes.
`matlist`	Array containing material numbers for each zone (else NULL).
`bnd_method`	Method to use for calculating external faces. See description below.

`DBCalcExternalFacelist2()` - Calculate an external facelist for a UCD mesh containing ghost zones.

C Signature

```
DBfacelist *DBCalcExternalFacelist2 (int nodelist[], int nnodes,
    int low_offset, int hi_offset, int origin,
    int shapetype[], int shapesize[],
    int shapecnt[], int nshapes, int matlist[], int bnd_method)
```

Fortran Signature:

```
None
```

Arg name	Description
`nodelist`	Array of node indices describing mesh zones.
`nnodes`	Number of nodes in associated mesh.
`lo_offset`	The number of ghost zones at the beginning of the nodelist.
`hi_offset`	The number of ghost zones at the end of the nodelist.
`origin`	Origin for indices in the nodelist array. Should be zero or one.

`shapetype`	Array of length nshapes containing the type of each zone shape. See description below.
`shapesize`	Array of length nshapes containing the number of noes used by each zone shape.
`shapecnt`	Array of length nshapes containing the number of zones having each shape.
`nshapes`	Number of zone shapes.
`matlist`	Array containing material numbers for each zone (else NULL).
`bnd_method`	Method to use for calculating external faces. See description below.

`DBStringArrayToStringList()` - Utility to catentate a group of strings into a single, semi-colon delimited string.

C Signature

```
void DBStringArrayToStringList(char const * const *strArray,
    int n, char **strList, int *m)
```

Fortran Signature:

```
None
```

Arg name	Description
`strArray`	Array of strings to catenate together. Note that it can be ok if some entries in strArray are the empty string, “” or NULL (0).
`n`	The number of entries in strArray. Passing -1 here indicates that the function should count entries in strArray until reaching the first NULL entry. In this case, embedded NULLs (0s) in strArray are, of course, not allowed.
`strList`	The returned catenated, semi-colon separated, single, string.

m	The returned length of strList.

DBStringListToStringArray() - Given a single, semi-colon delimited string, de-catenate it into an array of strings.

C Signature

```
char **DBStringListToStringArray(char *strList, int n,
    int handleSlashSwap, int skipFirstSemicolon)
```

Fortran Signature:

```
None
```

Arg name	Description
strList	A semi-colon separated, single string. Note that this string is modified by the call. If the caller doesn't want this, it will have to make a copy before calling.
n	The expected number of individual strings in strList. Pass -1 here if you have no aprior knowledge of this number. Knowing the number saves an additional pass over strList.
handleSlashSwap	a boolean to indicate if slash characters should be swapped as per differences in windows/linux filesystems.
This is specific to Silo's internal handling of strings used in multi-block objects. So, you should pass zero (0) here.	skipFirstSemicolon
a boolean to indicate if the first semicolon in the string should be skipped.	This is specific to Silo's internal usage for legacy compatibility. You should pass zero (0) here.

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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.

C Signature

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

Fortran Signature

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

Arg name	Description
`maxopts`	Initial maximum number of options expected in the optlist. If this maximum is exceeded, the library will silently re-allocate more space using the golden-rule.

DBAddOption() - Add an option to an option list.

C Signature

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

Fortran Signature

```
integer function dbaddcopt (optlist_id, option, cvalue, lcvalue)
integer function dbaddcaopt (optlist_id, option, nval, cvalue,
lcvalue)
integer function dbadddopt (optlist_id, option, dvalue)
integer function dbaddiopt (optlist_id, option, ivalue)
integer function dbaddropt (optlist_id, option, rvalue)

integer ivalue, optlist_id, option, lcvalue, nval
double precision dvalue
real rvalue
character*N cvalue (See "dbset2dstrlen" on page 288.)
```

Arg name	Description
optlist	Pointer to an option list structure containing option/value pairs. This structure is created with the DBMakeOptlist function.
option	Option definition. One of the predefined values described in the table in the notes section of each command which accepts an option list.
value	Pointer to the value associated with the provided option description. The data type is implied by option.

DBClearOption() - Remove an option from an option list

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`optlist`	The option list object for which you wish to remove an option
`optid`	The option id of the option you would like to remove

`DBGetOption()` - Retrieve the value set for an option in an option list

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`optlist`	The optlist to query
`optid`	The option id to query the value for

`DBFreeOptlist()` - Free memory associated with an option list.

C Signature

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

Fortran Signature

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

Arg Description

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

DBClearOptlist() - Clear an optlist.

C Signature

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

Fortran Signature:

```
None
```

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

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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.

C Signature

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

Fortran Signature

```
```

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`var`	Array defining the values associated with the variable.

`dims`	Array of length ndims which describes the dimensionality of the variable. Each value in the dims array indicates the number of elements contained in the variable along that dimension.
`ndims`	Number of dimensions.
`datatype`	Datatype of the variable. One of the predefined Silo data types.

`DBWriteSlice()` - Write a (hyper)slab of a simple variable

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`var`	Array defining the values associated with the slab.
`datatype`	Datatype of the variable. One of the predefined Silo data types.
`offset`	Array of length ndims of offsets in each dimension of the variable. This is the 0-origin position from which to begin writing the slice.
`length`	Array of length ndims of lengths of data in each dimension to write to the variable. All lengths must be positive.
	Array of length ndims of stride steps in each dimension. If

`stride`	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.

`DBReadVar()` - Read a simple Silo variable.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`result`	Pointer to memory into which the variable should be read. It is up to the application to provide sufficient space in which to read the variable.

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

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`varname`	Name of the simple variable.
`offset`	Array of length ndims of offsets in each dimension of the variable. This is the 0-origin position from which to begin reading the slice.
`length`	Array of length ndims of lengths of data in each dimension to read from the variable. All lengths must be positive.
`stride`	Array of length ndims of stride steps in each dimension. If no striding is desired, zeroes should be passed in this array.
`ndims`	Number of dimensions in the variable.
`result`	Pointer to location where the slice is to be written. It is up to the application to provide sufficient space in which to read the variable.

`DBGetVar()` - Allocate space for, and return, a simple variable.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.

`varname`	Name of the variable

`DBInqVarExists()` - Queries variable existence

C Signature

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

Fortran Signature:

```
None
```

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

`DBInqVarType()` - Return the type of the given object

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetVarByteLength()` - Return the byte length of a simple variable.

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetVarDims()` - Get dimension information of a variable in a Silo file

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`file`	The Silo database file handle.
`name`	The name of the Silo object to obtain dimension information for.
`maxdims`	The maximum size of dims.
`dims`	An array of maxdims integer values to be populated with the dimension information returned by this call.

`DBGetVarLength()` - Return the number of elements in a simple variable.

C Signature

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

Fortran Signature

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

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

`DBGetVarType()` - Return the Silo datatype of a simple variable.

C Signature

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

Fortran Signature:

```
None
```

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

`DBPutCompoundarray()` - Write a Compound Array object into a Silo file.

C Signature

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

Fortran Signature

```
integer function dbputca(dbid, name, lname, elemnames,
   lelemnames, elemlengths, nelems, values, datatype, optlist_id,
   status)
character*N elemnames (See “dbset2dstrlen” on page 288.)
```

Arg name	Description
dbfile	Database file pointer
name	Name of the compound array structure.
elemnames	Array of length nelems containing pointers to the names of the elements.
elemlengths	Array of length nelems containing the lengths of the elements.
nelems	Number of simple array elements.
values	Array whose length is determined by nelems and elemlengths containing the values of the simple array elements.
nvalues	Total length of the values array.
datatype	Data type of the values array. One of the predefined Silo types.
optlist	Pointer to an option list structure containing additional information to be included in the compound array object written into the Silo file. Use NULL is there are no options.

DBInqCompoundarray() - Inquire Compound Array attributes.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.
`name`	Name of the compound array.
`elemnames`	Returned array of length nelems containing pointers to the names of the array elements.
`elemlengths`	Returned array of length nelems containing the lengths of the array elements.
`nelems`	Returned number of array elements.
`nvalues`	Returned number of total values in the compound array.
`datatype`	Datatype of the data values. One of the predefined Silo data types.

`DBGetCompoundarray()` - Read a compound array from a Silo database.

C Signature

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

Fortran Signature

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

Arg name	Description
`dbfile`	Database file pointer.

`arrayname`	Name of the compound array.

`DBMakeObject()` - Allocate an object of the specified length and initialize it.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`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`	Initial maximum number of components needed for this object. If this number is exceeded, the library will silently re-allocate more space using the golden rule.

`DBFreeObject()` - Free memory associated with an object.

C Signature

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

Fortran Signature:

```
None
```

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

`DBChangeObject()` - Overwrite an existing object in a Silo file with a new object

C Signature

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

Fortran Signature:

```
None
```

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

`DBClearObject()` - Clear an object.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
	Pointer to the object to be cleared. This object is created

`object`	with the DBMakeObject function.

`DBAddDblComponent()` - Add a double precision floating point component to an object.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`object`	Pointer to an object. This object is created with the DBMakeObject function.
`compname`	The component name.
`d`	The value of the double precision floating point component.

`DBAddFltComponent()` - Add a floating point component to an object.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`object`	Pointer to an object. This object is created with the DBMakeObject function.

Arg name	Description
`compname`	The component name.
`f`	The value of the floating point component.

`DBAddIntComponent()` - Add an integer component to an object.

C Signature

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

Fortran Signature:

```
None
```

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

`DBAddStrComponent()` - Add a string component to an object.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`object`	Pointer to the object. This object is created with the DBMakeObject function.

`compname`	The component name.
`s`	The value of the string component. Silo copies the contents of the string.

`DBAddVarComponent()` - Add a variable component to an object.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`object`	Pointer to the object. This object is created with the DBMakeObject function.
`compname`	Component name.
`vardata`	Name of the variable object associated with the component (see Description).

`DBWriteComponent()` - Add a variable component to an object and write the associated data.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.
`object`	Pointer to the object.
`compname`	Component name.
`prefix`	Path name prefix of the object.
`datatype`	Data type of the component's data. One of: "short", "integer", "long", "float", "double", "char".
`var`	Pointer to the component's data.
`nd`	Number of dimensions of the component.
`count`	An array of length nd containing the length of the component in each of its dimensions.

`DBWriteObject()` - Write an object into a Silo file.

C Signature

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

Fortran Signature:

```
None
```

Arg name	Description
`dbfile`	Database file pointer.
`object`	Object created with DBMakeObject and populated with DBAddFltComponent, DBAddIntComponent, DBAddStrComponent, and DBAddVarComponent.
`freemem`	If non-zero, then the object will be freed after writing.

`DBGetObject()` - Read an object from a Silo file as a generic

object

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetComponent()` - Allocate space for, and return, an object component.

C Signature

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

Fortran Signature:

```
None
```

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

`DBGetComponentType()` - Return the type of an object component.

C Signature

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

Fortran Signature:

```
None
```

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

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

JSON Interface to Silo Objects

WARNING: JSON support in Silo is experimental. The interface may be dramatically re-worked, eliminated or replaced with something like Conduit. The Silo library must be configured with –enable-json option to enable these JSON support functions. When this option is enabled, the json-c library is compiled with Silo and installed to a json sub-directory at the same install point as the Silo library. In addition, applications using Silo's JSON interface will have to link with the json-c library (-I/json/include -L/json/lib -ljson).

JSON stands for JavaScript Object Notation. You can learn more about JSON at json. org. You can learn more about the json-c library at https://github. com/json-c/json-c/wiki.

Silo's JSON interface consists of two parts. The first part is just the json-c library interface which includes methods such as json_object_new_int() which creates a new integer valued JSON object and json_object_to_json_string() which returns an ascii string representation of a JSON object as well as many other methods. This interface is documented with the json-c library and is not documented here.

The second part is some extensions to the json-c library we have defined for the purposes of providing a higher performance JSON interface for Silo objects. This includes the definition of a new JSON object type; a pointer to an external array. This is called an extptr object and is actually a specific assemblage of the following 4 JSON sub-objects.

Member name "datatype" "ndims" "dims" "ptr" JSON type json_type_int json_type_int json_type_array json_type_string Meaning An integer value representing one of the Silo types DB_FLOAT, DB_INT, DB_DOUBLE, etc.

number of dimensions in the external array array of json_type_ints indicating size in each dimension The ascii hexidecimal representation of a void* pointer holding the data of the array

The extpr object is used for all Silo data representing problem-sized array data. For example, it is used to hold coordinate data for a mesh object, or variable data for a variable object or nodelist data for a zonelist object.

Another extension of JSON we have defined for Silo is a binary format for serialized JSON objects and methods to serialize and unserialize a JSON object to a binary buffer. Although JSON implementations other than json-c also define a binary format (see for example, BSON) we have defined one here as an extension to json-c. Silo's binary format can be used, for example, by a parallel application to conveniently send Silo objects between processors by serializing to a binary buffer at the sender and then unserializing at the receiver.

Any application wishing to use the JSON Silo interface must include the silo_json. h header file.

In this section we describe only those methods we have defined beyond those that come with the json-c library. The functions in this part of the library are json-c extensions 271

json-c extensions() - Extensions to json-c library to support Silo

C Signature

```
/* Create/delete extptr object */
    json_object* json_object_new_extptr(void *p, int ndims,
    int const *dims, int datatype);
    void json_object_extptr_delete(json_object *jso);

    /* Inspect various members of an extptr object */
    int json_object_is_extptr(json_object *obj);
    int json_object_get_extptr_datatype(json_object *obj);
    int json_object_get_extptr_ndims(json_object *obj);
    int json_object_get_extptr_dims_idx(json_object *obj, int idx);
    void* json_object_get_extptr_ptr(json_object *obj);
    /* binary serialization */
    int json_object_to_binary_buf(json_object *obj, int flags,
    void **buf, int *len);
    json_object* json_object_from_binary_buf(void *buf, int len);

    /* Read/Write raw binary data to a file */
    int json_object_to_binary_file(char const *filename,
    json_object *obj);
    json_object* json_object_from_binary_file(char const *filename);

    /* Fix extptr members that were ascii-fied via standard json
    string serialization */
    void json_object_reconstitute_extptrs(json_object *o);
```

Fortran Signature:

```
None
```

`DBWriteJsonObject()` - Write a JSON object to a Silo file

C Signature

```
DBWriteJsonObject(DBfile *db, json_object *jobj)
```

Fortran Signature:

```
None
```

Arg name	Description
`db`	Silo database file handle
`jobj`	JSON object pointer

`DBGetJsonObject()` - Get an object from a Silo file as a JSON object

C Signature

```
json_object *DBGetJsonObject(DBfile *db, char const *name)
```

Fortran Signature:

```
None
```

Arg name	Description
`db`	Silo database file handle
`name`	Name of object to read

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

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

C Signature

```
int n;
    char defs[1024];
    sprintf(defs, "foo scalar x+y;bar vector {x,y,z};"
    "gorfo scalar sqrt(x)";
    n = strlen(defs);
    DBWrite(dbfile, "_visit_defvars", defs, &n, 1, DB_CHAR);
    Description:
    Do not use this convention. Instead See "DBPutDefvars" on page 152.
    _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 196.

_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 shortcoming in Silo where multivar objects did not know the multimesh object they were associated with. This has since been corrected by the DBOPT_MMESH_NAME optlist option for a DBPutMultivar() call.

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 283.

12 API Section   Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within a single application interacting with a Silo file. Note that when Silo was originally written, the vision was that only visualization/post-processing tools would ever attempt to read the contents of Silo files. Therefore, the Fortran interface has never included all the companion functions to read objects. That said, it is possible to write simple fortran callable wrappers to the C functions much like the write

```
interface already implemented. Have a look in the source file silo_f.c
for examples.

    The functions described here are...
    dbmkptr     283
    dbrmptr     284
    dbset2dstrlen       285
    dbget2dstrlen       286
    DBFortranAllocPointer       287
    DBFortranAccessPointer      288
    DBFortranRemovePointer      289
    dbwrtfl     290
    dbmkptr
    —create a pointer-id from a pointer
    Synopsis:
    integer function dbmkptr(void p)
```

Arg name	Description
p	pointer for which a pointer-id is needed

_visit_searchpath() - directory order to search when opening a Silo file

C Signature

```
int n;
    char dirs[1024];
    sprintf(dirs, "nodesets;slides;");
    n = strlen(dirs);
    DBWrite(dbfile, "_visit_searchpath", dirs, &n, 1, DB_CHAR);
    Description:
    When opening a Silo file, an application is free to traverse
directories in whatever order it wishes. The _visit_searchpath
convention is used by the data producer to control how downstream,
post-processing tools traverse a Silo file's directory hierarchy.
    _visit_searchpath is an array of characters representing a semi-
colon separated list of directory names. If a character array of this
name is found at any directory in a Silo file, the directories it lists
(which are considered to be relative to the directory in which this
```

array is found unless the directory names begin with a slash '/') and only those directories are searched in the order they are specified in the list.

_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 196.

_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 shortcoming in Silo where multivar objects
did not know the multimesh object they were associated with. This has
since been corrected by the DBOPT_MMESH_NAME optlist option for a
DBPutMultivar() call.
    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 283.
    12 API Section      Fortran Interface
    The functions described in this section are either unique to the
Fortran interface or facilitate the mixing of C/C++ and Fortran within
a single application interacting with a Silo file. Note that when Silo
was originally written, the vision was that only visualization/post-
processing tools would ever attempt to read the contents of Silo files.
Therefore, the Fortran interface has never included all the companion
functions to read objects. That said, it is possible to write simple
fortran callable wrappers to the C functions much like the write
interface already implemented. Have a look in the source file silo_f.c
for examples.

    The functions described here are...
    dbmkptr     283
    dbrmptr     284
    dbset2dstrlen       285
    dbget2dstrlen       286
    DBFortranAllocPointer       287
    DBFortranAccessPointer      288
    DBFortranRemovePointer      289
    dbwrtfl     290
    dbmkptr
    —create a pointer-id from a pointer
    Synopsis:
    integer function dbmkptr(void p)
```

Arg name	Description
`p`	pointer for which a pointer-id is needed

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

C Signature

```
int domToGroupMap[16];
int j;
for (j = 0; j < 16; j++) domToGroupMap[j] = j%4;
DBWrite(dbfile, “_visit_domain_groups”, domToGroupMap,
&j, 1, DB_INT);
```

Description:

Do not use this convention. Instead use Mesh Region Grouping (MRG) trees. See “DBMakeMrgtree” on page 196.

_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 shortcoming in Silo where multivar objects did not know the multimesh object they were associated with. This has since been corrected by the DBOPT_MMESH_NAME optlist option for a DBPutMultivar() call.

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 283.

12 API Section Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within

```
a single application interacting with a Silo file. Note that when Silo
was originally written, the vision was that only visualization/post-
processing tools would ever attempt to read the contents of Silo files.
Therefore, the Fortran interface has never included all the companion
functions to read objects. That said, it is possible to write simple
fortran callable wrappers to the C functions much like the write
interface already implemented. Have a look in the source file silo_f.c
for examples.

    The functions described here are...
    dbmkptr     283
    dbrmptr     284
    dbset2dstrlen       285
    dbget2dstrlen       286
    DBFortranAllocPointer       287
    DBFortranAccessPointer      288
    DBFortranRemovePointer      289
    dbwrtfl     290
    dbmkptr
    —create a pointer-id from a pointer
    Synopsis:
    integer function dbmkptr(void p)
```

Arg name	Description
`p`	pointer for which a pointer-id is needed

`AlphabetizeVariables()` - flag to tell post-processor to alphabetize variable lists

C Signature

```
int doAlpha = 1;
    int n = 1;
    DBWrite(dbfile, “AlphabetizeVariables”, &doAlpha, &n, 1, DB_INT);
    Description:
    The AlphabetizeVariables convention is a simple integer value
which, if non-zero, indicates that the post-processor should
alphabetize its variable lists. In VisIt, this would mean that various
menus in the GUI, for example, are constructed such that variable names
```

placed near the top of the menus come alphabetically before variable names near the bottom of the menus. Otherwise, variable names are presented in the order they are encountered in the database which is often the order they were written to the database by the data producer.

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 shortcoming in Silo where multivar objects did not know the multimesh object they were associated with. This has since been corrected by the DBOPT_MMESH_NAME optlist option for a DBPutMultivar() call.

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 283.

12 API Section Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within a single application interacting with a Silo file. Note that when Silo was originally written, the vision was that only visualization/post-processing tools would ever attempt to read the contents of Silo files. Therefore, the Fortran interface has never included all the companion functions to read objects. That said, it is possible to write simple fortran callable wrappers to the C functions much like the write interface already implemented. Have a look in the source file silo_f.c for examples.

The functions described here are...

dbmkptr

—create a pointer-id from a pointer

Synopsis:

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

Arg name	Description
`p`	pointer for which a pointer-id is needed

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

C Signature

```
int isTimeVarying = 1;
    int n = 1;
    DBWrite(dbfile, “ConnectivityIsTimeVarying”, &isTimeVarying, &n, 1,
DB_INT);
    Description:
    The ConnectivityIsTimeVarying convention is a simple integer flag
which, if non-zero, indicates to post-processing tools that the
connectivity for the mesh(s) in the database varies with time. This has
important performance implications and should only be specified if
indeed it is necessary as, for instance, in post-processors that assume
connectivity is NOT time varying. This is an assumption made by VisIt
and the ConnectivityIsTimeVarying convention is a way to tell VisIt to
NOT make this assumption.
    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 shortcoming in Silo where multivar objects did not know the multimesh object they were associated with. This has since been corrected by the DBOPT_MMESH_NAME optlist option for a DBPutMultivar() call.

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 283.

12 API Section Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within a single application interacting with a Silo file. Note that when Silo was originally written, the vision was that only visualization/post-processing tools would ever attempt to read the contents of Silo files. Therefore, the Fortran interface has never included all the companion functions to read objects. That said, it is possible to write simple fortran callable wrappers to the C functions much like the write interface already implemented. Have a look in the source file silo_f.c for examples.

The functions described here are...

dbmkptr

—create a pointer-id from a pointer

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

Arg name	Description
`p`	pointer for which a pointer-id is needed

`MultivarToMultimeshMap_vars()` - list of multivars to be associated with multimeshes

C Signature

```
int len;
    char tmpStr[256];
    sprintf(tmpStr, "d;p;u;v;w;hist;mat1");
    len = strlen(tmpStr);
    DBWrite(dbfile, "MultivarToMultimeshMap_vars", tmpStr, &len, 1,
DB_CHAR);
    Description:
    Do not use this convention. Instead use the DBOPT_MMESH_NAME
optlist option for a DBPutMultivar() call to associate a multimesh with
a multivar.
    The MultivarToMultimeshMap_vars use convention goes hand-in-hand
with the MultivarToMultimeshMap_meshes use convention. The _vars
portion is an array of characters defining a semi-colon separated list
of multivar object names to be associated with multi-mesh names. The
_mesh portion is an array of characters defining a semi-colon separated
list of associated multimesh object names. This convention was
introduced to deal with a shortcoming in Silo where multivar objects
did not know the multimesh object they were associated with. This has
since been corrected by the DBOPT_MMESH_NAME optlist option for a
DBPutMultivar() call.
    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 283.
    12 API Section      Fortran Interface
    The functions described in this section are either unique to the
Fortran interface or facilitate the mixing of C/C++ and Fortran within
a single application interacting with a Silo file. Note that when Silo
was originally written, the vision was that only visualization/post-
processing tools would ever attempt to read the contents of Silo files.
Therefore, the Fortran interface has never included all the companion
functions to read objects. That said, it is possible to write simple
fortran callable wrappers to the C functions much like the write
interface already implemented. Have a look in the source file silo_f.c
for examples.

    The functions described here are...
    dbmkptr     283
    dbrmptr     284
    dbset2dstrlen       285
    dbget2dstrlen       286
    DBFortranAllocPointer       287
    DBFortranAccessPointer      288
    DBFortranRemovePointer      289
    dbwrtfl     290
    dbmkptr
    —create a pointer-id from a pointer
    Synopsis:
    integer function dbmkptr(void p)
```

Arg name	Description
p	pointer for which a pointer-id is needed

MultivarToMultimeshMap_meshes() - list of multimeshes to be associated with multivars

C Signature

```
int len;
    char tmpStr[256];
    sprintf(tmpStr, "mesh1;mesh1;mesh1;mesh1;mesh1;mesh1;mesh1");
    len = strlen(tmpStr);
    DBWrite(dbfile, "MultivarToMultimeshMap_meshes", tmpStr, &len, 1,
DB_CHAR);
    Description:
    See “MultivarToMultimeshMap_vars” on page 283.
    12 API Section      Fortran Interface
    The functions described in this section are either unique to the
Fortran interface or facilitate the mixing of C/C++ and Fortran within
a single application interacting with a Silo file. Note that when Silo
was originally written, the vision was that only visualization/post-
processing tools would ever attempt to read the contents of Silo files.
Therefore, the Fortran interface has never included all the companion
functions to read objects. That said, it is possible to write simple
fortran callable wrappers to the C functions much like the write
interface already implemented. Have a look in the source file silo_f.c
for examples.

    The functions described here are...
    dbmkptr     283
    dbrmptr     284
    dbset2dstrlen       285
    dbget2dstrlen       286
    DBFortranAllocPointer       287
    DBFortranAccessPointer      288
    DBFortranRemovePointer      289
    dbwrtfl     290
    dbmkptr
    —create a pointer-id from a pointer
    Synopsis:
    integer function dbmkptr(void p)
```

Arg name	Description
p	pointer for which a pointer-id is needed

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Fortran Interface

The functions described in this section are either unique to the Fortran interface or facilitate the mixing of C/C++ and Fortran within a single application interacting with a Silo file. Note that when Silo was originally written, the vision was that only visualization/post-processing tools would ever attempt to read the contents of Silo files. Therefore, the Fortran interface has never included all the companion functions to read objects. That said, it is possible to write simple fortran callable wrappers to the C functions much like the write interface already implemented. Have a look in the source file silo_f. c for examples.

The functions described here are…

`dbmkptr()` - create a pointer-id from a pointer

C Signature

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

Arg name	Description
`p`	pointer for which a pointer-id is needed

`dbrmptr()` - remove an old and no longer needed pointer-id

C Signature

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

Arg name	Description
`ptr_id`	the pointer-id to remove

`dbset2dstrlen()` - Set the size of a ‘row’ for pointers to ‘arrays’ of strings

C Signature

```
integer function dbset2dstrlen(int len)

    integer len
```

Arg name	Description
`len`	The length to set

`dbget2dstrlen()` - Get the size of a ‘row’ for pointers to ‘arrays’ of character strings

C Signature

```
integer function dbget2dstrlen()
```

Arguments: None

`DBFortranAllocPointer()` - Facilitates accessing C objects through Fortran

C Signature

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

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

`DBFortranAccessPointer()` - Access Silo objects created through the Fortran Silo interface.

C Signature

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

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

`DBFortranRemovePointer()` - Removes a pointer from the Fortran-C index table

C Signature

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

Arg name	Description
`value`	An integer returned by DBFortranAllocPointer

`dbwrtfl()` - Write a facelist object referenced by its object_id to a silo file

C Signature

```
dbwrtfl(dbid, name, lname, object_id, status)
```

Arg name	Description
`dbid`	The identifier for the Silo database to write the object to.
`name`	The name to be assigned to the object in the file.
`lname`	The length of the name argument.

`object_id`	The identifier for the facelist object, obtained via dbcalcfl.
`status`	Return value indicating success or failure of the operation; 0 on success, -1 on failure.

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Python Interface

It is probably easiest to understand the Python interface to Silo by examining some examples and tests. In the source code distribution, you can find some examples in tools/python and tests in tests directories. Here, we briefly describe Silo's Python interface.

In order for an installation of Silo to have the Python interface, Silo must have been configured with –enable-pythonmodule and NOT with –disable-shared Autoconf configuration switches.

The Python interface will be in the lib dir of the Silo installation, named Silo. so. To use it, Python needs to be told where to find it. You can do this a couple of ways; through the PYTHONPATH environment variable or by explicitly adding the Silo installation lib dir to Python's path using sys. path. append(). For example, if Silo is installed to /foo/bar, this works…

% env PYTHONPATH=/foo/bar/lib python Python 2. 7. 10 (default, Oct 23 2015, 19:19:21) [GCC 4. 2. 1 Compatible Apple LLVM 7. 0. 0] on darwin Type "help", "copyright", "credits" or "license" for more info.

> import Silo Or, if you prefer to use sys. path. append…

python Python 2. 7. 10 (default, Oct 23 2015, 19:19:21) [GCC 4. 2. 1 Compatible Apple LLVM 7. 0. 0] on darwin Type "help", "copyright", "credits" or "license" for more info.

> import sys sys. path. append("/foo/bar/lib") import Silo

`Silo.Open()` - Open a Silo file (See DBOpen)

C Signature

```
DBfile Silo.Open(filename, flags);
```

Arg name	Description
`filename`	Name of the Silo file to open
`flags`	Pass either Silo.DB_READ if you will only read objects from the file or Silo.DB_APPEND if you need to also write data to the file.

`Silo.Open()` - Open a Silo file (See DBOpen)

C Signature

```
DBfile Silo.Open(filename, flags);
```

Arg name	Description
`filename`	Name of the Silo file to open
`flags`	Pass either Silo.DB_READ if you will only read objects from the file or Silo.DB_APPEND if you need to also write data to the file.

`Silo.Create()` - Create a new silo file (See DBCreate)

C Signature

```
DBfile Silo.Create(filename, info, driver, clobber)
```

Arg name	Description
`filename`	[required string] name of the file to create
`info`	[required string] comment to be stored in the file

`driver`	[optional int] which driver to use. Pass either Silo.DB_PDB or Silo.DB_HDF5. Note that advanced driver features are not available through the Python interface. Default is Silo.DB_PDB.
`clobber`	[optional int] indicate whether any existing file should be clobbered. Pass either Silo.DB_CLOBBER or Silo.DB_NOCLOBBER. Default is Silo.DB_CLOBBER.

`<DBfile>.GetToc()` - Get the table of contents

C Signature

```
DBtoc <DBfile>.GetToc()
    Description:
    Returns a DBToc object as a Python object. This probably should
really be a Python dictionary object but it is not presently. There are
no methods defined for a DBToc object but if you print it, you can get
the list of objects in the current working directory in the file.
    <DBfile>.GetVarInfo
    —Get metadata and bulk data of any object (See DBGetObject)
    Synopsis:
    dict <DBfile>.GetVarInfo(name, flag)
```

Arg name	Description
`name`	[required string] name of object to read
`flag`	[optional int] flag to indicate if object bulk/raw data should be included. Pass 0 to NOT also read object bulk/raw data. Pass non-zero to also read object bulk/raw data. Default is 0.

`<DBfile>.GetVarInfo()` - Get metadata and bulk data of any object (See DBGetObject)

C Signature

```
dict <DBfile>.GetVarInfo(name, flag)
```

Arg name	Description
`name`	[required string] name of object to read
`flag`	[optional int] flag to indicate if object bulk/raw data should be included. Pass 0 to NOT also read object bulk/raw data. Pass non-zero to also read object bulk/raw data. Default is 0.

`<DBfile>.GetVar()` - Get a primitive array (See DBReadVar)

C Signature

```
tuple <DBfile>.GetVar(name)
```

Arg name	Description
`name`	[required string] name of primitive array to read

`<DBfile>.SetDir()` - Set current working directory of the Silo file (See DBSetDir)

C Signature

```
NoneType <DBfile>.SetDir(name)
```

Arg name	Description
`name`	[required string] name of directory to set

`<DBfile>.Close()` - Close the Silo file

C Signature

```
NoneType <DBfile>.Close()
    Description:
    Close the Silo file
    <DBfile>.WriteObject
    —Write a Python dictionary as a Silo object (See DBWriteObject)
    Synopsis:
    NoneType <DBfile>.WriteObject(name, obj_dict)
```

Arg name	Description
`name`	[required string] name of the new object to write
`obj_dict`	[required dict] Python dictionary containing object data

`<DBfile>.WriteObject()` - Write a Python dictionary as a Silo object (See DBWriteObject)

C Signature

```
NoneType <DBfile>.WriteObject(name, obj_dict)
```

Arg name	Description
`name`	[required string] name of the new object to write
`obj_dict`	[required dict] Python dictionary containing object data

`<DBfile>.Write()` - Write primitive array data to a Silo file (see DBWrite)

C Signature

```
NoneType <DBfile>.Write(name, data)
```

Arg name	Description
`name`	[required string] name of the primitive array
`data`	[required tuple] the data to write

`<DBfile>.MkDir()` - Make a directory in a Silo file

C Signature

```
NoneType <DBfile>.MkDir(name)
```

Arg name	Description
`name`	[required string] name of the directory to create

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Deprecated Functions

The following functions were deprecated from Silo in version 4.

1. Attempts to call these methods in later versions may still succeed. However, deprecation warnings will be generated on stderr (See “DBSetDeprecateWarnings” on page 35.). There is no guarantee that these methods will exist in later versions of Silo.

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Silo Library Header File

We include the contents of the Silo header file here including a description of all DBxxx object structs that are returned in DBGetXXX() calls as well as all other constant and symbols defined by the library.

./ Silo

Mesh and Field I/O Library and Scientific Database

View on GitHub

Brief History and Background of Silo

LLNL began developing the Silo library in the early 1990s 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 subefforts 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.
```