Oracle Berkeley DB

Berkeley DB API Reference for C

11g Release 2

ORACLE
BERKELEY DB

Legal Notice

To obtain a copy of this document's original source code, please submit a request to the Oracle Technology Network forum at: http://forums.oracle.com/forums/forum.jspa?forumID=271

Published 3/30/2010

Table of Contents

Preface

Welcome to Berkeley DB 11*g* Release 2 (DB). This document describes the C API for DB library version 11.2.5.0. It is intended to describe the DB API, including all classes, methods, and functions. As such, this document is intended for C developers who are actively writing or maintaining applications that make use of DB databases.

Conventions Used in this Book

The following typographical conventions are used within in this manual:

Structure names are represented in `monospaced font`, as are `method names`. For example: "`DB->open()` is a method on a `DB` handle."

Variable or non-literal text is presented in *italics*. For example: "Go to your *DB_INSTALL* directory."

Program examples are displayed in a `monospaced font` on a shaded background. For example:

```
/* File: gettingstarted_common.h */
typedef struct stock_dbs {
    DB *inventory_dbp; /* Database containing inventory information */
    DB *vendor_dbp;    /* Database containing vendor information */

    char *db_home_dir;       /* Directory containing the database files */
    char *inventory_db_name; /* Name of the inventory database */
    char *vendor_db_name;    /* Name of the vendor database */
} STOCK_DBS;
```

Note

Finally, notes of interest are represented using a note block such as this.

For More Information

Beyond this manual, you may also find the following sources of information useful when building a DB application:

- Getting Started with Berkeley DB for C
- Getting Started with Transaction Processing for C
- Berkeley DB Getting Started with Replicated Applications for C
- Berkeley DB C++ API
- Berkeley DB STL API
- Berkeley DB TCL API
- Berkeley DB Installation and Build Guide

- Berkeley DB Programmer's Reference Guide
- Berkeley DB Getting Started with the SQL APIs

Chapter 1. Introduction to Berkeley DB APIs

Welcome to the Berkeley DB API Reference Manual for C.

DB is a general-purpose embedded database engine that is capable of providing a wealth of data management services. It is designed from the ground up for high-throughput applications requiring in-process, bullet-proof management of mission-critical data. DB can gracefully scale from managing a few bytes to terabytes of data. For the most part, DB is limited only by your system's available physical resources.

This manual describes the various APIs and command line utilities available for use in the DB library.

For a general description of using DB beyond the reference material available in this manual, see the Getting Started Guides which are identified in this manual's preface.

This manual is broken into chapters, each one of which describes a series of APIs designed to work with one particular aspect of the DB library. In many cases, each such chapter is organized around a "handle", or class, which provides an interface to DB structures such as databases, environments or locks. However, in some cases, methods for multiple handles are combined together when they are used to control or interface with some isolated DB functionality. See, for example, the The DB_LSN Handle (page 330) chapter.

Within each chapter, methods, functions and command line utilities are organized alphabetically.

Chapter 2. The DB Handle

The DB is the handle for a single Berkeley DB database. A Berkeley DB database provides a mechanism for organizing key-data pairs of information. From the perspective of some database systems, a Berkeley DB database could be thought of as a single table within a larger database.

You create a DB handle using the db_create (page 16) function. For most database activities, you must then open the handle using the DB->open() (page 66) method. When you are done with them, handles must be closed using the DB->close() (page 13) method.

Alternatively, you can create a DB and then rename, remove or verify the database without performing an open. See DB->rename() (page 77), DB->remove() (page 75) or DB->verify() (page 144) for information on these activities.

It is possible to create databases such that they are organized within a *database environment*. Environments are optional for simple Berkeley DB applications that do not use transactions, recovery, replication or any other advanced features. For simple Berkeley DB applications, environments still offer some advantages. For example, they provide some organizational benefits on-disk (all databases are located on disk relative to the environment). Also, if you are using multiple databases, then environments allow your databases to share a common in-memory cache, which makes for more efficient usage of your hardware's resources.

See DB_ENV for information on using database environments.

You specify the underlying organization of the data in the database (e.g. BTree, Hash, Queue, and Recno) when you open the database. When you create a database, you are free to specify any of the available database types. On subsequent opens, you must either specify the access method used when you first opened the database, or you can specify `DB_UNKNOWN` in order to have this information retrieved for you. See the DB->open() (page 66) method for information on specifying database types.

Database and Related Methods

Database Operations	Description
DB->associate()	Associate a secondary index
DB->associate_foreign()	Associate a foreign index
DB->close()	Close a database
db_create	Create a database handle
DB->compact()	Compact a database
DB->del()	Delete items from a database
DB->err()	Error message
DB->exists()	Return if an item appears in a database
DB->fd()	Return a file descriptor from a database
DB->get()	Get items from a database
DB->get_byteswapped()	Return if the underlying database is in host order
DB->get_dbname()	Return the file and database name
DB->get_multiple()	Return if the database handle references multiple databases
DB->get_open_flags()	Returns the flags specified to DB->open
DB->get_type()	Return the database type
DB->join()	Perform a database join on cursors
DB->key_range()	Return estimate of key location
DB->open()	Open a database
DB->put()	Store items into a database
DB->remove()	Remove a database
DB->rename()	Rename a database
DB->set_priority(), DB->get_priority()	Set/get cache page priority
DB->stat()	Database statistics
DB->stat_print()	Display database statistics
DB->sync()	Flush a database to stable storage
DB->truncate()	Empty a database
DB->upgrade()	Upgrade a database
DB->verify()	Verify/salvage a database
DB->cursor()	Create a cursor handle
Database Configuration	
DB->set_alloc()	Set local space allocation functions

Database Operations	Description
DB->set_cachesize(), DB->get_cachesize()	Set/get the database cache size
DB->set_create_dir(), DB->get_create_dir()	Set/get the directory in which a database is placed
DB->set_dup_compare()	Set a duplicate comparison function
DB->set_encrypt(), DB->get_encrypt_flags()	Set/get the database cryptographic key
DB->set_errcall()	Set error message callback
DB->set_errfile(), DB->get_errfile()	Set/get error message FILE
DB->set_errpfx(), DB->get_errpfx()	Set/get error message prefix
DB->set_feedback()	Set feedback callback
DB->set_flags(), DB->get_flags()	Set/get general database configuration
DB->set_lorder(), DB->get_lorder()	Set/get the database byte order
DB->set_msgcall()	Set informational message callback
DB->set_msgfile(), DB->get_msgfile()	Set/get informational message FILE
DB->set_pagesize(), DB->get_pagesize()	Set/get the underlying database page size
DB->set_partition()	Set database partitioning
DB->set_partition_dirs(), DB->get_partition_dirs()	Set/get the directories used for database partitions
Btree/Recno Configuration	
DB->set_append_recno()	Set record append callback
DB->set_bt_compare()	Set a Btree comparison function
DB->set_bt_compress()	Set Btree compression functions
DB->set_bt_minkey(), DB->get_bt_minkey()	Set/get the minimum number of keys per Btree page
DB->set_bt_prefix()	Set a Btree prefix comparison function
DB->set_re_delim(), DB->get_re_delim()	Set/get the variable-length record delimiter
DB->set_re_len(), DB->get_re_len()	Set/get the fixed-length record length
DB->set_re_pad(), DB->get_re_pad()	Set/get the fixed-length record pad byte
DB->set_re_source(), DB->get_re_source()	Set/get the backing Recno text file
Hash Configuration	
DB->set_h_compare()	Set a Hash comparison function
DB->set_h_ffactor(), DB->get_h_ffactor()	Set/get the Hash table density
DB->set_h_hash()	Set a hashing function
DB->set_h_nelem(), DB->get_h_nelem()	Set/get the Hash table size
Queue Configuration	

Database Operations	Description
DB->set_q_extentsize(), DB->get_q_extentsize()	Set/get Queue database extent size

DB->associate()

```
#include <db.h>

int
DB->associate(DB *primary, DB_TXN *txnid, DB *secondary,
    int (*callback)(DB *secondary,
    const DBT *key, const DBT *data, DBT *result), u_int32_t flags);
```

The `DB->associate()` function is used to declare one database a secondary index for a primary database. The DB handle that you call the `associate()` method from is the primary database.

After a secondary database has been "associated" with a primary database, all updates to the primary will be automatically reflected in the secondary and all reads from the secondary will return corresponding data from the primary. Note that as primary keys must be unique for secondary indices to work, the primary database must be configured without support for duplicate data items. See Secondary Indices in the *Berkeley DB Programmer's Reference Guide* for more information.

The `DB->associate()` method returns a non-zero error value on failure and 0 on success.

Parameters

primary

The **primary** parameter should be a database handle for the primary database that is to be indexed.

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

secondary

The **secondary** parameter should be an open database handle of either a newly created and empty database that is to be used to store a secondary index, or of a database that was previously associated with the same primary and contains a secondary index. Note that it is not safe to associate as a secondary database a handle that is in use by another thread of control or has open cursors. If the handle was opened with the DB_THREAD flag it is safe to use it in multiple threads of control after the `DB->associate()` method has returned. Note also that either secondary keys must be unique or the secondary database must be configured with support for duplicate data items.

callback

The **callback** parameter is a callback function that creates the set of secondary keys corresponding to a given primary key and data pair.

The callback parameter may be NULL if both the primary and secondary database handles were opened with the DB_RDONLY flag.

The callback takes four arguments:

- `secondary`

  The **secondary** parameter is the database handle for the secondary.

- `key`

  The **key** parameter is a DBT referencing the primary key.

- `data`

  The **data** parameter is a DBT referencing the primary data item.

- `result`

  The **result** parameter is a zeroed DBT in which the callback function should fill in **data** and **size** fields that describe the secondary key or keys.

  ### Note

  Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

The result DBT can have the following flags set in its **flags** field:

- `DB_DBT_APPMALLOC`

  If the callback function needs to allocate memory for the **result** data field (rather than simply pointing into the primary key or datum), DB_DBT_APPMALLOC should be set in the **flags** field of the **result** DBT, which indicates that Berkeley DB should free the memory when it is done with it.

- `DB_DBT_MULTIPLE`

  To return multiple secondary keys, DB_DBT_MULTIPLE should be set in the **flags** field of the **result** DBT, which indicates Berkeley DB should treat the **size** field as the number of secondary keys (zero or more), and the **data** field as a pointer to an array of that number of DBTs describing the set of secondary keys.

  When multiple secondary keys are returned, keys may not be repeated. In other words, there must be no repeated record numbers in the array for Recno and Queue databases, and keys must not compare equally using the secondary database's comparison function for Btree and Hash databases. If keys are repeated, operations may fail and the secondary may become inconsistent with the primary.

  The DB_DBT_APPMALLOC flag may be set for any DBT in the array of returned DBT's to indicate that Berkeley DB should free the memory referenced by that particular DBT's data field when it is done with it.

The DB_DBT_APPMALLOC flag may be combined with DB_DBT_MULTIPLE in the **result** DBT's **flag** field to indicate that Berkeley DB should free the array once it is done with all of the returned keys.

In addition, the callback can optionally return the following special value:

- `DB_DONOTINDEX`

  If any key/data pair in the primary yields a null secondary key and should be left out of the secondary index, the callback function may optionally return DB_DONOTINDEX. Otherwise, the callback function should return 0 in case of success or an error outside of the Berkeley DB name space in case of failure; the error code will be returned from the Berkeley DB call that initiated the callback.

  If the callback function returns DB_DONOTINDEX for any key/data pairs in the primary database, the secondary index will not contain any reference to those key/data pairs, and such operations as cursor iterations and range queries will reflect only the corresponding subset of the database. If this is not desirable, the application should ensure that the callback function is well-defined for all possible values and never returns DB_DONOTINDEX.

  Returning DB_DONOTINDEX is equivalent to setting DB_DBT_MULTIPLE on the **result** DBT and setting the **size** field to zero.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_CREATE`

  If the secondary database is empty, walk through the primary and create an index to it in the empty secondary. This operation is potentially very expensive.

  If the secondary database has been opened in an environment configured with transactions, the entire secondary index creation is performed in the context of a single transaction.

  Care should be taken not to use a newly-populated secondary database in another thread of control until the `DB->associate()` call has returned successfully in the first thread.

  If transactions are not being used, care should be taken not to modify a primary database being used to populate a secondary database, in another thread of control, until the `DB->associate()` call has returned successfully in the first thread. If transactions are being used, Berkeley DB will perform appropriate locking and the application need not do any special operation ordering.

- `DB_IMMUTABLE_KEY`

  Specifies the secondary key is immutable.

  This flag can be used to optimize updates when the secondary key in a primary record will never be changed after the primary record is inserted. For immutable secondary keys, a best effort is made to avoid calling the secondary callback function when primary records are updated. This optimization may reduce the overhead of update operations significantly if the callback function is expensive.

Be sure to specify this flag only if the secondary key in the primary record is never changed. If this rule is violated, the secondary index will become corrupted, that is, it will become out of sync with the primary.

Errors

The `DB->associate()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

If the secondary database handle has already been associated with this or another database handle; the secondary database handle is not open; the primary database has been configured to allow duplicates; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->associate_foreign()

```
#include <db.h>

int
DB->associate_foreign(DB *foreign, DB *secondary,,
   int (*callback)(DB *secondary,
   const DBT *key, DBT *data, const DBT *foreignkey, int *changed),
   u_int32_t flags);
```

The `DB->associate_foreign()` function is used to declare one database a foreign constraint for a secondary database. The DB handle that you call the `associate_foreign()` method from is the foreign database.

After a foreign database has been "associated" with a secondary database, all keys inserted into the secondary must exist in the foreign database. Attempting to add a record with a foreign key that does not exist in the foreign database will cause the put method to fail and return `DB_FOREIGN_CONFLICT`.

Deletions in the foreign database affect the secondary in a manner defined by the flags parameter. See Foreign Indices in the *Berkeley DB Programmer's Reference Guide* for more information.

The `DB->associate_foreign()` method returns a non-zero error value on failure and 0 on success.

Parameters

foreign

The **foreign** parameter should be a database handle for the foreign database.

secondary

The **secondary** parameter should be an open database handle of a database that contains a secondary index who's keys also exist in the **foreign** database.

callback

The **callback** parameter is a callback function that nullifies the foreign key portion of a data DBT.

The callback parameter must be NULL if either DB_FOREIGN_ABORT or DB_FOREIGN_CASCADE is set.

The callback takes four arguments:

- `secondary`

  The **secondary** parameter is the database handle for the secondary.

- `key`

The **key** parameter is a DBT referencing the primary key.

- `data`

  The **data** parameter is a DBT referencing the primary data item to be updated.

- `foreignkey`

  The **foreignkey** parameter is a DBT referencing the foreign key which is being deleted.

- `changed`

  The **changed** parameter is a pointer to a boolean value, indicated whether **data** has changed.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

flags

The **flags** parameter must be set to one of the following values:

- `DB_FOREIGN_ABORT`

  Abort the deletion of a key in the foreign database and return DB_FOREIGN_CONFLICT if that key exists in the secondary database. The deletion should be protected by a transaction to ensure database integrity after the aborted delete.

- `DB_FOREIGN_CASCADE`

  The deletion of a key in the foreign database will also delete that key from the secondary database (and the corresponding entry in the secondary's primary database.)

- `DB_FOREIGN_NULLIFY`

  The deletion of a key in the foreign database will call the nullification function passed to associate_foreign and update the secondary database with the changed data.

Errors

The `DB->associate_foreign()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return

`DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

If the foreign database handle is a secondary index; the foreign database handle has been configured to allow duplicates; the foreign database handle is a renumbering recno database; callback is configured and DB_FOREIGN_NULLIFY is not; DB_FOREIGN_NULLIFY is configured and callback is not.

Class

DB

See Also

Database and Related Methods (page 3)

DB->close()

```
#include <db.h>

int
DB->close(DB *db, u_int32_t flags);
```

The `DB->close()` method flushes cached database information to disk, closes any open cursors, frees allocated resources, and closes underlying files. When the close operation for a cursor fails, the method returns a non-zero error value for the first instance of such an error, and continues to close the rest of the cursors and database handles.

Although closing a database handle will close any open cursors, it is recommended that applications explicitly close all their DBcursor handles before closing the database. The reason why is that when the cursor is explicitly closed, the memory allocated for it is reclaimed; however, this will *not* happen if you close a database while cursors are still opened.

The same rule, for the same reasons, hold true for DB_TXN handles. Simply make sure you close all your transaction handles before closing your database handle.

Because key/data pairs are cached in memory, applications should make a point to always either close database handles or sync their data to disk (using the DB->sync() (page 138) method) before exiting, to ensure that any data cached in main memory are reflected in the underlying file system.

When called on a database that is the primary database for a secondary index, the primary database should be closed only after all secondary indices referencing it have been closed.

When multiple threads are using the DB concurrently, only a single thread may call the `DB->close()` method.

The DB handle may not be accessed again after `DB->close()` is called, regardless of its return.

If you do not close the DB handle explicitly, it will be closed when the environment handle that owns the DB handle is closed.

The `DB->close()` method returns a non-zero error value on failure and 0 on success. The error values that `DB->close()` method returns include the error values of `DBcursor->close()` and the following:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Parameters

flags

The **flags** parameter must be set to 0 or be set to the following value:

- DB_NOSYNC

  Do not flush cached information to disk. This flag is a dangerous option. It should be set only if the application is doing logging (with transactions) so that the database is recoverable after a system or application crash, or if the database is always generated from scratch after any system or application crash.

  It is important to understand that flushing cached information to disk only minimizes the window of opportunity for corrupted data. Although unlikely, it is possible for database corruption to happen if a system or application crash occurs while writing data to the database. To ensure that database corruption never occurs, applications must either: use transactions and logging with automatic recovery; use logging and application-specific recovery; or edit a copy of the database, and once all applications using the database have successfully called `DB->close()`, atomically replace the original database with the updated copy.

  Note that this flag only works when the database has been opened using an environment.

Errors

The `DB->close()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

The error messages returned for the first error encountered when `DB->close()` method closes any open cursors include:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

db_create

```
#include <db.h>

int db_create(DB **dbp, DB_ENV *dbenv, u_int32_t flags);
```

The `db_create()` function creates a DB structure that is the handle for a Berkeley DB database. This function allocates memory for the structure, returning a pointer to the structure in the memory to which **dbp** refers. To release the allocated memory and discard the handle, call the DB->close() (page 13), DB->remove() (page 75), DB->rename() (page 77), or DB->verify() (page 144) methods.

DB handles are free-threaded if the DB_THREAD flag is specified to the DB->open() (page 66) method when the database is opened or if the database environment in which the database is opened is free-threaded. The handle should not be closed while any other handle that refers to the database is in use; for example, database handles must not be closed while cursor handles into the database remain open, or transactions that include operations on the database have not yet been committed or aborted. Once the DB->close() (page 13), DB->remove() (page 75), DB->rename() (page 77), or DB->verify() (page 144) methods are called, the handle may not be accessed again, regardless of the method's return.

The DB handle contains a special field, `app_private`, which is declared as type `void *`. This field is provided for the use of the application program. It is initialized to NULL and is not further used by Berkeley DB in any way.

The `db_create` function returns a non-zero error value on failure and 0 on success.

Parameters

dbp

The **dbp** parameter references the memory into which the returned structure pointer is stored.

dbenv

If the **dbenv** parameter is NULL, the database is standalone; that is, it is not part of any Berkeley DB environment.

If the **dbenv** parameter is not NULL, the database is created within the specified Berkeley DB environment. The database access methods automatically make calls to the other subsystems in Berkeley DB, based on the enclosing environment. For example, if the environment has been configured to use locking, the access methods will automatically acquire the correct locks when reading and writing pages of the database.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `db_create()` function may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->compact()

```
#include <db.h>

int
DB->compact(DB *db, DB_TXN *txnid,
    DBT *start, DBT *stop, DB_COMPACT *c_data, u_int32_t flags, DBT *end);
```

The `DB->compact()` method compacts Btree, Hash, and Recno access method databases, and optionally returns unused Btree, Hash or Recno database pages to the underlying filesystem.

The `DB->compact()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL.

If a transaction handle is supplied to this method, then the operation is performed using that transaction. In this event, large sections of the tree may be locked during the course of the transaction.

If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected using multiple transactions. These transactions will be periodically committed to avoid locking large sections of the tree. Any deadlocks encountered cause the compaction operation to be retried from the point of the last transaction commit.

start

If non-NULL, the **start** parameter is the starting point for compaction. For a Btree or Recno database, compaction will start at the smallest key greater than or equal to the specified key. For a Hash database, the compaction will start in the bucket specified by the integer stored in the key. If NULL, compaction will start at the beginning of the database.

stop

If non-NULL, the **stop** parameter is the stopping point for compaction. For a Btree or Recno database, compaction will stop at the page with the smallest key greater than the specified key. For a Hash database, compaction will stop in the bucket specified by the integer stored in the key. If NULL, compaction will stop at the end of the database.

c_data

If non-NULL, the **c_data** parameter contains additional compaction configuration parameters, and returns compaction operation statistics, in a structure of type `DB_COMPACT`.

The following input configuration fields are available from the `DB_COMPACT` structure:

- `int compact_fillpercent;`

  If non-zero, this provides the goal for filling pages, specified as a percentage between 1 and 100. Any page in the database not at or above this percentage full will be considered for compaction. The default behavior is to consider every page for compaction, regardless of its page fill percentage.

- `int compact_pages;`

  If non-zero, the call will return after the specified number of pages have been freed, or no more pages can be freed.

- `db_timeout_t compact_timeout;`

  If non-zero, and no **txnid** parameter was specified, this parameter identifies the lock timeout used for implicit transactions, in microseconds.

The following output statistics fields are available from the `DB_COMPACT` structure:

- `u_int32_t compact_deadlock;`

  An output statistics parameter: if no **txnid** parameter was specified, the number of deadlocks which occurred.

- `u_int32_t compact_pages_examine;`

  An output statistics parameter: the number of database pages reviewed during the compaction phase.

- `u_int32_t compact_empty_buckets;`

  An output statistics parameter: the number of empty hash buckets that were found the compaction phase.

- `u_int32_t compact_pages_free;`

  An output statistics parameter: the number of database pages freed during the compaction phase.

- `u_int32_t compact_levels;`

  An output statistics parameter: the number of levels removed from the Btree or Recno database during the compaction phase.

- `u_int32_t compact_pages_truncated;`

  An output statistics parameter: the number of database pages returned to the filesystem.

flags

The **flags** parameter must be set to 0 or one of the following values:

- DB_FREELIST_ONLY

  Do no page compaction, only returning pages to the filesystem that are already free and at the end of the file.

- DB_FREE_SPACE

  Return pages to the filesystem when possible. If this flag is not specified, pages emptied as a result of compaction will be placed on the free list for re-use, but never returned to the filesystem.

  Note that only pages at the end of a file can be returned to the filesystem. Because of the one-pass nature of the compaction algorithm, any unemptied page near the end of the file inhibits returning pages to the file system. A repeated call to the DB->compact() method with a low **compact_fillpercent** may be used to return pages in this case.

end

If non-NULL, the **end** parameter will be filled with the database key marking the end of the compaction operation in a Btree or Recno database. This is generally the first key of the page where the operation stopped. For a Hash database, this will hold the integer value representing which bucket the compaction stopped in.

Errors

The DB->compact() method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EACCES

An attempt was made to modify a read-only database.

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->del()

```
#include <db.h>

int
DB->del(DB *db, DB_TXN *txnid, DBT *key, u_int32_t flags);
```

The `DB->del()` method removes key/data pairs from the database. The key/data pair associated with the specified **key** is discarded from the database. In the presence of duplicate key values, all records associated with the designated key will be discarded.

When called on a database that has been made into a secondary index using the DB->associate() (page 6) method, the `DB->del()` method deletes the key/data pair from the primary database and all secondary indices.

The `DB->del()` method will return DB_NOTFOUND if the specified key is not in the database. The `DB->del()` method will return DB_KEYEMPTY if the database is a Queue or Recno database and the specified key exists, but was never explicitly created by the application or was later deleted. Unless otherwise specified, the `DB->del()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

key

The key DBT operated on.

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_CONSUME`

  If the database is of type DB_QUEUE then this flag may be set to force the head of the queue to move to the first non-deleted item in the queue. Normally this is only done if the deleted item is exactly at the head when deleted.

- `DB_MULTIPLE`

  Delete multiple data items using keys from the buffer to which the **key** parameter refers.

  To delete records in bulk by key with the btree or hash access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT (page 182) and

DB_MULTIPLE_WRITE_NEXT (page 183). To delete records in bulk by record number, construct the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT (page 187) and DB_MULTIPLE_RECNO_WRITE_NEXT (page 188) with a data size of zero.

A successful bulk delete operation is logically equivalent to a loop through each key/data pair, performing a DB->del() (page 22) for each one.

See the DBT and Bulk Operations (page 176) for more information on working with bulk updates.

The `DB_MULTIPLE` flag may only be used alone.

- `DB_MULTIPLE_KEY`

  Delete multiple data items using keys and data from the buffer to which the **key** parameter refers.

  To delete records in bulk with the btree or hash access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT (page 182) and DB_MULTIPLE_KEY_WRITE_NEXT (page 185). To delete records in bulk with the recno or hash access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT (page 187) and DB_MULTIPLE_RECNO_WRITE_NEXT (page 188).

  See the DBT and Bulk Operations (page 176) for more information on working with bulk updates.

  The `DB_MULTIPLE_KEY` flag may only be used alone.

Errors

The `DB->del()` method may fail and return one of the following non-zero errors:

DB_FOREIGN_CONFLICT

A foreign key constraint violation has occurred. This can be caused by one of two things:

1. An attempt was made to add a record to a constrained database, and the key used for that record does not exist in the foreign key database.
2. DB_FOREIGN_ABORT (page 11) was declared for a foreign key database, and then subsequently a record was deleted from the foreign key database without first removing it from the constrained secondary database.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DB_SECONDARY_BAD

A secondary index references a nonexistent primary key.

EACCES

An attempt was made to modify a read-only database.

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->err()

```
#include <db.h>

void
DB->err(DB *db, int error, const char *fmt, ...);

void
DB->errx(DB *db, const char *fmt, ...);
```

The DB_ENV->err() (page 204), DB_ENV->errx(), DB->err() and DB->errx() methods provide error-messaging functionality for applications written using the Berkeley DB library.

The DB->err() and DB_ENV->err() (page 204) methods construct an error message consisting of the following elements:

- **An optional prefix string**

  If no error callback function has been set using the DB_ENV->set_errcall() (page 253) method, any prefix string specified using the DB_ENV->set_errpfx() (page 257) method, followed by two separating characters: a colon and a <space> character.

- **An optional printf-style message**

  The supplied message **fmt**, if non-NULL, in which the ANSI C X3.159-1989 (ANSI C) printf function specifies how subsequent parameters are converted for output.

- **A separator**

  Two separating characters: a colon and a <space> character.

- **A standard error string**

  The standard system or Berkeley DB library error string associated with the **error** value, as returned by the db_strerror (page 290) method.

The DB->errx() and DB_ENV->errx() methods are the same as the DB->err() and DB_ENV->err() (page 204) methods, except they do not append the final separator characters and standard error string to the error message.

This constructed error message is then handled as follows:

- If an error callback function has been set (see DB->set_errcall() (page 97) and DB_ENV->set_errcall() (page 253)), that function is called with two parameters: any prefix string specified (see DB->set_errpfx() (page 101) and DB_ENV->set_errpfx() (page 257)) and the error message.

- If a C library FILE * has been set (see DB->set_errfile() (page 99) and DB_ENV->set_errfile() (page 255)), the error message is written to that output stream.

- If none of these output options have been configured, the error message is written to stderr, the standard error output stream.

Parameters

error

The **error** parameter is the error value for which the DB_ENV->err() (page 204) and `DB->err()` methods will display an explanatory string.

fmt

The **fmt** parameter is an optional printf-style message to display.

Class

DB

See Also

Database and Related Methods (page 3)

DB->exists()

```
#include <db.h>

int
DB->exists(DB *db, DB_TXN *txnid, DBT *key, u_int32_t flags);
```

The `DB->exists()` method returns whether the specified key appears in the database.

The `DB->exists()` method will return DB_NOTFOUND if the specified key is not in the database. The `DB->exists()` method will return DB_KEYEMPTY if the database is a Queue or Recno database and the specified key exists, but was never explicitly created by the application or was later deleted.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

key

The key DBT operated on.

flags

The **flags** parameter must be set to zero or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_READ_COMMITTED`

  Configure a transactional read operation to have degree 2 isolation (the read is not repeatable).

- `DB_READ_UNCOMMITTED`

  Configure a transactional read operation to have degree 1 isolation, reading modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

- `DB_RMW`

  Acquire write locks instead of read locks when doing the read, if locking is configured. Setting this flag can eliminate deadlock during a read-modify-write cycle by acquiring the write lock during the read part of the cycle so that another thread of control acquiring a read lock for the same item, in its own read-modify-write cycle, will not result in deadlock.

Because the `DB->exists()` method will not hold locks across Berkeley DB calls in non-transactional operations, the DB_RMW flag to the `DB->exists()` call is meaningful only in the presence of transactions.

Class

DB

See Also

Database and Related Methods (page 3)

DB->fd()

```
#include <db.h>

int
DB->fd(DB *db, int *fdp);
```

The `DB->fd()` method provides access to a file descriptor representative of the underlying database. A file descriptor referring to the same file will be returned to all processes that call DB->open() (page 66) with the same **file** parameter.

This file descriptor may be safely used as a parameter to the **fcntl**(2) and **flock**(2) locking functions.

The `DB->fd()` method only supports a coarse-grained form of locking. Applications should instead use the Berkeley DB lock manager where possible.

The `DB->fd()` method returns a non-zero error value on failure and 0 on success.

Parameters

fdp

The **fdp** parameter references memory into which the current file descriptor is copied.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get()

```
#include <db.h>

int
DB->get(DB *db,
    DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);

int
DB->pget(DB *db,
    DB_TXN *txnid, DBT *key, DBT *pkey, DBT *data, u_int32_t flags);
```

The `DB->get()` method retrieves key/data pairs from the database. The address and length of the data associated with the specified **key** are returned in the structure to which **data** refers.

In the presence of duplicate key values, `DB->get()` will return the first data item for the designated key. Duplicates are sorted by:

- Their sort order, if a duplicate sort function was specified.
- Any explicit cursor designated insertion.
- By insert order. This is the default behavior.

Retrieval of duplicates requires the use of cursor operations. See DBcursor->get() (page 158) for details.

When called on a database that has been made into a secondary index using the DB->associate() (page 6) method, the `DB->get()` and `DB->pget()` methods return the key from the secondary index and the data item from the primary database. In addition, the `DB->pget()` method returns the key from the primary database. In databases that are not secondary indices, the `DB->pget()` method will always fail.

The `DB->get()` method will return DB_NOTFOUND if the specified key is not in the database. The `DB->get()` method will return DB_KEYEMPTY if the database is a Queue or Recno database and the specified key exists, but was never explicitly created by the application or was later deleted. Unless otherwise specified, the `DB->get()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

key

The key DBT operated on.

pkey

The **pkey** parameter is the return key from the primary database.

data

The data DBT operated on.

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_CONSUME`

  Return the record number and data from the available record closest to the head of the queue, and delete the record. The record number will be returned in **key**, as described in DBT. The data will be returned in the **data** parameter. A record is available if it is not deleted and is not currently locked. The underlying database must be of type Queue for DB_CONSUME to be specified.

- `DB_CONSUME_WAIT`

  The DB_CONSUME_WAIT flag is the same as the DB_CONSUME flag, except that if the Queue database is empty, the thread of control will wait until there is data in the queue before returning. The underlying database must be of type Queue for DB_CONSUME_WAIT to be specified.

  If lock or transaction timeouts have been specified, the `DB->get()` method with the DB_CONSUME_WAIT flag may return DB_LOCK_NOTGRANTED. This failure, by itself, does not require the enclosing transaction be aborted.

- `DB_GET_BOTH`

  Retrieve the key/data pair only if both the key and data match the arguments.

  When using a secondary index handle, the `DB_GET_BOTH`: flag causes:

  - the `DB->pget()` version of this method to retun the secondary key/primary key/data tuple only if both the primary and secondary keys match the arguments.
  - the `DB->get()` version of this method to result in an error.

- `DB_SET_RECNO`

  Retrieve the specified numbered key/data pair from a database. Upon return, both the **key** and **data** items will have been filled in.

  The **data** field of the specified **key** must be a pointer to a logical record number (that is, a **db_recno_t**). This record number determines the record to be retrieved.

For DB_SET_RECNO to be specified, the underlying database must be of type Btree, and it must have been created with the DB_RECNUM flag.

In addition, the following flags may be set by bitwise inclusively **OR**'ing them into the **flags** parameter:

- `DB_IGNORE_LEASE`

  Return the data item irrespective of the state of master leases. The item will be returned under all conditions: if master leases are not configured, if the request is made to a client, if the request is made to a master with a valid lease, or if the request is made to a master without a valid lease.

- `DB_MULTIPLE`

  Return multiple data items in the buffer to which the **data** parameter refers.

  In the case of Btree or Hash databases, all of the data items associated with the specified key are entered into the buffer. In the case of Queue or Recno databases, all of the data items in the database, starting at, and subsequent to, the specified key, are entered into the buffer.

  The buffer to which the **data** parameter refers must be provided from user memory (see DB_DBT_USERMEM). The buffer must be at least as large as the page size of the underlying database, aligned for unsigned integer access, and be a multiple of 1024 bytes in size. If the buffer size is insufficient, then upon return from the call the size field of the **data** parameter will have been set to an estimated buffer size, and the error DB_BUFFER_SMALL is returned. (The size is an estimate as the exact size needed may not be known until all entries are read. It is best to initially provide a relatively large buffer, but applications should be prepared to resize the buffer as necessary and repeatedly call the method.)

  The DB_MULTIPLE flag may only be used alone, or with the DB_GET_BOTH and DB_SET_RECNO options. The DB_MULTIPLE flag may not be used when accessing databases made into secondary indices using the DB->associate() (page 6) method.

  See the DBT and Bulk Operations (page 176) for more information on working with bulk get.

- `DB_READ_COMMITTED`

  Configure a transactional get operation to have degree 2 isolation (the read is not repeatable).

- `DB_READ_UNCOMMITTED`

  Configure a transactional get operation to have degree 1 isolation, reading modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

- `DB_RMW`

Acquire write locks instead of read locks when doing the read, if locking is configured. Setting this flag can eliminate deadlock during a read-modify-write cycle by acquiring the write lock during the read part of the cycle so that another thread of control acquiring a read lock for the same item, in its own read-modify-write cycle, will not result in deadlock.

Because the `DB->get()` method will not hold locks across Berkeley DB calls in non-transactional operations, the DB_RMW flag to the `DB->get()` call is meaningful only in the presence of transactions.

Errors

The `DB->get()` method may fail and return one of the following non-zero errors:

DB_BUFFER_SMALL

The requested item could not be returned due to undersized buffer.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_LOCK_NOTGRANTED

The `DB_CONSUME_WAIT` flag was specified, lock or transaction timers were configured and the lock could not be granted before the wait-time expired.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LEASE_EXPIRED

The operation failed because the site's replication master lease has expired.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DB_SECONDARY_BAD

A secondary index references a nonexistent primary key.

EINVAL

If a record number of 0 was specified; the DB_THREAD flag was specified to the DB->open() (page 66) method and none of the DB_DBT_MALLOC, DB_DBT_REALLOC or DB_DBT_USERMEM flags were set in the DBT; the `DB->pget()` method was called with a DB handle that does not refer to a secondary index; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_bt_minkey()

```
#include <db.h>

int
DB->get_bt_minkey(DB *db, u_int32_t *bt_minkeyp);
```

The `DB->get_bt_minkey()` method returns the minimum number of key/data pairs intended to be stored on any single Btree leaf page. This value can be set using the DB->set_bt_minkey() (page 88) method.

The `DB->get_bt_minkey()` method may be called at any time during the life of the application.

The `DB->get_bt_minkey()` method returns a non-zero error value on failure and 0 on success.

Parameters

bt_minkeyp

The `DB->get_bt_minkey()` method returns the minimum number of key/data pairs intended to be stored on any single Btree leaf page in **bt_minkeyp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_bt_minkey() (page 88)

DB->get_byteswapped()

```
#include <db.h>

int
DB->get_byteswapped(DB *db, int *isswapped);
```

The `DB->get_byteswapped()` method returns whether the underlying database files were created on an architecture of the same byte order as the current one, or if they were not (that is, big-endian on a little-endian machine, or vice versa). This information may be used to determine whether application data needs to be adjusted for this architecture or not.

The `DB->get_byteswapped()` method may not be called before the DB->open() (page 66) method is called.

The `DB->get_byteswapped()` method returns a non-zero error value on failure and 0 on success.

Parameters

isswapped

If the underlying database files were created on an architecture of the same byte order as the current one, 0 is stored into the memory location referenced by **isswapped**. If the underlying database files were created on an architecture of a different byte order as the current one, 1 is stored into the memory location referenced by **isswapped**.

Errors

The `DB->get_byteswapped()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called before DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_cachesize()

```
#include <db.h>

int
DB->get_cachesize(DB *db,
    u_int32_t *gbytesp, u_int32_t *bytesp, int *ncachep);
```

The `DB->get_cachesize()` method returns the current size and composition of the cache. These values may be set using the DB->set_cachesize() (page 91) method.

The `DB->get_cachesize()` method may be called at any time during the life of the application.

The `DB->get_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytesp

The **gbytesp** parameter references memory into which the gigabytes of memory in the cache is copied.

bytesp

The **bytesp** parameter references memory into which the additional bytes of memory in the cache is copied.

ncachep

The **ncachep** parameter references memory into which the number of caches is copied.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_cachesize() (page 91)

DB->get_create_dir()

```
#include <db.h>

int
DB->get_create_dir(DB *db, const char **dirp);
```

Determine which directory a database file will be created in or was found in.

The `DB->get_create_dir()` method may be called at any time.

The `DB->get_create_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirp

The **dirp** will be set to the directory specified in the call to DB->set_create_dir() (page 93) method on this handle or to the directory that the database was found in after DB->open() (page 66) has been called.

Errors

The `DB->get_create_dir()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_dbname()

```
#include <db.h>

int
DB->get_dbname(DB *db, const char **filenamep, const char **dbnamep);
```

The `DB->get_dbname()` method returns the filename and database name used by the DB handle.

The `DB->get_dbname()` method returns a non-zero error value on failure and 0 on success.

Parameters

filenamep

The **filenamep** parameter references memory into which a pointer to the current filename is copied.

dbnamep

The **dbnamep** parameter references memory into which a pointer to the current database name is copied.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_encrypt_flags()

```
#include <db.h>

int
DB->get_encrypt_flags(DB *db, u_int32_t *flagsp);
```

The `DB->get_encrypt_flags()` method returns the encryption flags. This flag can be set using the DB->set_encrypt() (page 96) method.

The `DB->get_encrypt_flags()` method may be called at any time during the life of the application.

The `DB->get_encrypt_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB->get_encrypt_flags()` method returns the encryption flags in **flagsp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_encrypt() (page 96)

DB->get_errfile()

```
#include <db.h>

void
DB->get_errfile(DB *db, FILE **errfilep);
```

The `DB->get_errfile()` method returns the `FILE *`, as set by the DB->set_errfile() (page 99) method.

The `DB->get_errfile()` method may be called at any time during the life of the application.

Parameters

errfilep

The `DB->get_errfile()` method returns the FILE * in **errfilep**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_errfile() (page 99)

DB->get_errpfx()

```
#include <db.h>

void DB->get_errpfx(DB *db, const char **errpfxp);
```

The `DB->get_errpfx()` method returns the error prefix.

The `DB->get_errpfx()` method may be called at any time during the life of the application.

Parameters

errpfxp

The `DB->get_errpfx()` method returns a reference to the error prefix in **errpfxp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_errpfx() (page 101)

DB->get_flags()

```
#include <db.h>

int
DB->get_flags(DB *db, u_int32_t *flagsp);
```

The `DB->get_flags()` method returns the current database flags as set by the DB->set_flags() (page 104) method.

The `DB->get_flags()` method may be called at any time during the life of the application.

The `DB->get_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB->get_flags()` method returns the current flags in **flagsp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_flags() (page 104)

DB->get_h_ffactor()

```
#include <db.h>

int
DB->get_h_ffactor(DB *db, u_int32_t *h_ffactorp);
```

The `DB->get_h_ffactor()` method returns the hash table density as set by the DB->set_h_ffactor() (page 112) method. The hash table density is the number of items that Berkeley DB tries to place in a hash bucket before splitting the hash bucket.

The `DB->get_h_ffactor()` method may be called at any time during the life of the application.

The `DB->get_h_ffactor()` method returns a non-zero error value on failure and 0 on success.

Parameters

h_ffactorp

The `DB->get_h_ffactor()` method returns the hash table density in **h_ffactorp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_h_ffactor() (page 112)

DB->get_h_nelem()

```
#include <db.h>

int
DB->get_h_nelem(DB *db, u_int32_t *h_nelemp);
```

The `DB->get_h_nelem()` method returns the estimate of the final size of the hash table as set by the DB->set_h_nelem() (page 114) method.

The `DB->get_h_nelem()` method may be called at any time during the life of the application.

The `DB->get_h_nelem()` method returns a non-zero error value on failure and 0 on success.

Parameters

h_nelemp

The `DB->get_h_nelem()` method returns the estimate of the final size of the hash table in **h_nelemp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_h_nelem() (page 114)

DB->get_lorder()

```
#include <db.h>

int
DB->get_lorder(DB *db, int *lorderp);
```

The `DB->get_lorder()` method returns the database byte order; a byte order of 4,321 indicates a big endian order, and a byte order of 1,234 indicates a little endian order. This value is set using the DB->set_lorder() (page 115) method.

The `DB->get_lorder()` method may be called at any time during the life of the application.

The `DB->get_lorder()` method returns a non-zero error value on failure and 0 on success.

Parameters

lorderp

The `DB->get_lorder()` method returns the database byte order in **lorderp**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_lorder() (page 115)

DB->get_msgfile()

```
#include <db.h>

void
DB->get_msgfile(DB *db, FILE **msgfilep);
```

The `DB->get_msgfile()` method returns the `FILE *` used to output informational or statistical messages. This file handle is configured using the DB->set_msgfile() (page 118) method.

The `DB->get_msgfile()` method may be called at any time during the life of the application.

Parameters

msgfilep

The `DB->get_msgfile()` method returns the `FILE *` in **msgfilep**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_msgfile() (page 118)

DB->get_multiple()

```
#include <db.h>

int
DB->get_multiple(DB *db);
```

This method returns non-zero if the DB handle references a physical file supporting multiple databases, and 0 otherwise.

In this case, the DB handle is a handle on a database whose key values are the names of the databases stored in the physical file and whose data values are opaque objects. No keys or data values may be modified or stored using the database handle.

This method may not be called before the DB->open() (page 66) method is called.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_open_flags()

```
#include <db.h>

int
DB->get_open_flags(DB *db, u_int32_t *flagsp);
```

The `DB->get_open_flags()` method returns the current open method flags. That is, this method returns the flags that were specified when DB->open() (page 66) was called.

The `DB->get_open_flags()` method may not be called before the `DB->open()` method is called.

The `DB->get_open_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB->get_open_flags()` method returns the current open method flags in **flagsp**.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_partition_callback()

```
#include <db.h>

int
DB->get_partition_callback(DB *db, u_int32_t *partsp,
 u_int32_t (**callback_fcn) (DB *dbp, DBT *key);
```

The `DB->get_partition_callback()` method returns the partitioning information as set by the DB->set_partition() (page 120) method.

The `DB->get_partition_callback()` method may be called at any time during the life of the application.

The `DB->get_partition_callback()` method returns a non-zero error value on failure and 0 on success.

Parameters

partsp

The `DB->get_partition_callback()` method returns number of partitions in the **partsp** parameter.

callback_fcn

The **callback_fcn** parameter will be set to the partitioning function.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_partition() (page 120)

DB->get_partition_dirs()

```
#include <db.h>

int
DB->get_partition_dirs(DB *db, const char ***dirsp);
```

Determine which directorise the database partitions files will be created in or were found in.

The `DB->get_partition_dirs()` method may be called at any time.

The `DB->get_partition_dirs()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirsp

The **dirsp** will be set to the array of directories specified in the call to DB->set_partition_dirs() (page 122) method on this handle or to the directoreies that the database partitions were found in after DB->open() (page 66) has been called.

Errors

The `DB->get_partition_dirs()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->get_partition_keys()

```
#include <db.h>

int
DB->get_partition_keys(DB *db, u_int32_t *partsp, DBT *keysp);
```

The `DB->get_partition_keys()` method returns the partitioning information as set by the DB->set_partition() (page 120) method.

The `DB->get_partition_keys()` method may be called at any time during the life of the application.

The `DB->get_partition_keys()` method returns a non-zero error value on failure and 0 on success.

Parameters

partsp

The `DB->get_partition_keys()` method returns number of partitions in the **partsp** parameter.

keysp

The **keysp** parameter will be set to the array of partitioning keys.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_partition() (page 120)

DB->get_pagesize()

```
#include <db.h>

int
DB->get_pagesize(DB *db, u_int32_t *pagesizep);
```

The `DB->get_pagesize()` method returns the database's current page size, as set by the DB->set_pagesize() (page 119) method. Note that if `DB->set_pagesize()` was not called by your application, then the default pagesize is selected based on the underlying filesystem I/O block size. If you call `DB->get_pagesize()` before you have opened the database, the value returned by this method is therefore the underlying filesystem I/O block size.

The `DB->get_pagesize()` method may be called only after the database has been opened.

The `DB->get_pagesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

pagesizep

The `DB->get_pagesize()` method returns the page size in **pagesizep**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_pagesize() (page 119)

DB->get_priority()

```
#include <db.h>

int
DB->get_priority(DB *db, DB_CACHE_PRIORITY *priorityp);
```

The `DB->get_priority()` method returns the cache priority for pages referenced by the DB handle. This priority value is set using the DB->set_priority() (page 123) method.

The `DB->get_priority()` method may be called only after the database has been opened.

The `DB->get_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priorityp

The `DB->get_priority()` method returns a reference to the cache priority in **priorityp**. See DB->set_priority() (page 123) for a list of possible priorities.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_priority() (page 123)

DB->get_q_extentsize()

```
#include <db.h>

int
DB->get_q_extentsize(DB *db, u_int32_t *extentsizep);
```

The `DB->get_q_extentsize()` method returns the number of pages in an extent. This value is used only for Queue databases and is set using the DB->set_q_extentsize() (page 124) method.

The `DB->get_q_extentsize()` method may be called only after the database has been opened.

The `DB->get_q_extentsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

extentsizep

The `DB->get_q_extentsize()` method returns the number of pages in an extent in **extentsizep**. If used on a handle that has not yet been opened, `0` is returned.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_q_extentsize() (page 124)

DB->get_re_delim()

```
#include <db.h>

int
DB->get_re_delim(DB *db, int *delimp);
```

The `DB->get_re_delim()` method returns the delimiting byte, which is used to mark the end of a record in the backing source file for the Recno access method. This value is set using the DB->set_re_delim() (page 125) method.

The `DB->get_re_delim()` method may be called only after the database has been opened.

The `DB->get_re_delim()` method returns a non-zero error value on failure and 0 on success.

Parameters

delimp

The `DB->get_re_delim()` method returns the delimiting byte in **delimp**. If this method is called on a handle that has not yet been opened, then the default delimiting byte is returned. See DB->set_re_delim() (page 125) for details.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_re_delim() (page 125)

DB->get_re_len()

```
#include <db.h>

int
DB->get_re_len(DB *db, u_int32_t *re_lenp);
```

The `DB->get_re_len()` method returns the length of the records held in a Queue access method database. This value can be set using the DB->set_re_len() (page 126) method.

The `DB->get_re_len()` method may be called only after the database has been opened.

The `DB->get_re_len()` method returns a non-zero error value on failure and 0 on success.

Parameters

re_lenp

The `DB->get_re_len()` method returns the record length in **re_lenp**. If the record length has never been set using DB->set_re_len() (page 126), then `0` is returned.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_re_len() (page 126)

DB->get_re_pad()

```
#include <db.h>

int
DB->get_re_pad(DB *db, int *re_padp);
```

The `DB->get_re_pad()` method returns the pad character used for short, fixed-length records used by the Queue and Recno access methods. This character is set using the DB->set_re_pad() (page 127) method.

The `DB->get_re_pad()` method may be called only after the database has been opened.

The `DB->get_re_pad()` method returns a non-zero error value on failure and 0 on success.

Parameters

re_padp

The `DB->get_re_pad()` method returns the pad character in **re_padp**. If used on a handle that has not yet been opened, the default pad character is returned. See the DB->set_re_pad() (page 127) method description for what that default value is.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_re_pad() (page 127)

DB->get_re_source()

```
#include <db.h>

int
DB->get_re_source(DB *db, const char **sourcep);
```

The `DB->get_re_source()` method returns the source file used by the Recno access method. This file is configured for the Recno access method using the DB->set_re_source() (page 128) method.

The `DB->get_re_source()` method may be called only after the database has been opened.

The `DB->get_re_source()` method returns a non-zero error value on failure and 0 on success.

Parameters

sourcep

The `DB->get_re_source()` method returns a reference to the source file in **sourcep**.

Class

DB

See Also

Database and Related Methods (page 3), DB->set_re_source() (page 128)

DB->get_type()

```
#include <db.h>

int
DB->get_type(DB *db, DBTYPE *type);
```

The `DB->get_type()` method returns the type of the underlying access method (and file format). The type value is one of DB_BTREE, DB_HASH, DB_RECNO, or DB_QUEUE. This value may be used to determine the type of the database after a return from DB->open() (page 66) with the **type** parameter set to DB_UNKNOWN.

The `DB->get_type()` method may not be called before the DB->open() (page 66) method is called.

The `DB->get_type()` method returns a non-zero error value on failure and 0 on success.

Parameters

type

The **type** parameter references memory into which the type of the underlying access method is copied.

Errors

The `DB->get_type()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called before DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->join()

```
#include <db.h>

int
DB->join(DB *primary,
    DBC **curslist, DBC **dbcp, u_int32_t flags);
```

The `DB->join()` method creates a specialized join cursor for use in performing equality or natural joins on secondary indices. For information on how to organize your data to use this functionality, see Equality join.

The `DB->join()` method is called using the DB handle of the primary database.

The join cursor supports only the DBcursor->get() (page 158) and DBcursor->close() (page 151) cursor functions:

- **DBcursor->get() (page 158)**

  Iterates over the values associated with the keys to which each item in **curslist** was initialized. Any data value that appears in all items specified by the **curslist** parameter is then used as a key into the **primary**, and the key/data pair found in the **primary** is returned. The **flags** parameter must be set to 0 or the following value:

  - **DB_JOIN_ITEM**

    Do not use the data value found in all the cursors as a lookup key for the **primary**, but simply return it in the key parameter instead. The data parameter is left unchanged.

  In addition, the following flag may be set by bitwise inclusively **OR**'ing it into the **flags** parameter:

  - **DB_READ_UNCOMMITTED**

    Configure a transactional join operation to have degree 1 isolation, reading modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

  - **DB_RMW**

    Acquire write locks instead of read locks when doing the read, if locking is configured. Setting this flag can eliminate deadlock during a read-modify-write cycle by acquiring the write lock during the read part of the cycle so that another thread of control acquiring a read lock for the same item, in its own read-modify-write cycle, will not result in deadlock.

- **DBcursor->close() (page 151)**

  Close the returned cursor and release all resources. (Closing the cursors in **curslist** is the responsibility of the caller.)

The `DB->join()` method returns a non-zero error value on failure and 0 on success.

Parameters

curslist

The **curslist** parameter contains a NULL terminated array of cursors. Each cursor must have been initialized to refer to the key on which the underlying database should be joined. Typically, this initialization is done by a DBcursor->get() (page 158) call with the DB_SET flag specified. Once the cursors have been passed as part of a **curslist**, they should not be accessed or modified until the newly created join cursor has been closed, or else inconsistent results may be returned.

Joined values are retrieved by doing a sequential iteration over the first cursor in the **curslist** parameter, and a nested iteration over each secondary cursor in the order they are specified in the **curslist** parameter. This requires database traversals to search for the current datum in all the cursors after the first. For this reason, the best join performance normally results from sorting the cursors from the one that refers to the least number of data items to the one that refers to the most. By default, `DB->join()` does this sort on behalf of its caller.

For the returned join cursor to be used in a transaction-protected manner, the cursors listed in **curslist** must have been created within the context of the same transaction.

dbcp

The newly created join cursor is returned in the memory location to which **dbcp** refers.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_JOIN_NOSORT`

  Do not sort the cursors based on the number of data items to which they refer. If the data are structured so that cursors with many data items also share many common elements, higher performance will result from listing those cursors before cursors with fewer data items; that is, a sort order other than the default. The DB_JOIN_NOSORT flag permits applications to perform join optimization prior to calling the `DB->join()` method.

Errors

The `DB->join()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DB_SECONDARY_BAD

A secondary index references a nonexistent primary key.

EINVAL

If cursor methods other than DBcursor->get() (page 158) or DBcursor->close() (page 151) were called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->key_range()

```
#include <db.h>

int
DB->key_range(DB *db, DB_TXN *txnid,
    DBT *key, DB_KEY_RANGE *key_range, u_int32_t flags);
```

The `DB->key_range()` method returns an estimate of the proportion of keys that are less than, equal to, and greater than the specified key. The underlying database must be of type Btree.

The `DB->key_range()` method fills in a structure of type DB_KEY_RANGE. The following data fields are available from the DB_KEY_RANGE structure:

- **double less;**

  A value between 0 and 1, the proportion of keys less than the specified key.

- **double equal;**

  A value between 0 and 1, the proportion of keys equal to the specified key.

- **double greater;**

  A value between 0 and 1, the proportion of keys greater than the specified key.

Values are in the range of 0 to 1; for example, if the field **less** is 0.05, 5% of the keys in the database are less than the **key** parameter. The value for **equal** will be zero if there is no matching key, and will be non-zero otherwise.

The `DB->key_range()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected. The `DB->key_range()` method does not retain the locks it acquires for the life of the transaction, so estimates may not be repeatable.

key

The key DBT operated on.

key_range

The estimates are returned in the **key_range** parameter, which contains three elements of type double: **less**, **equal**, and **greater**. Values are in the range of 0 to 1; for example, if the field **less** is 0.05, 5% of the keys in the database are less than the **key** parameter. The value for **equal** will be zero if there is no matching key, and will be non-zero otherwise.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB->key_range()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

If the underlying database was not of type Btree; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->open()

```
#include <db.h>

int
DB->open(DB *db, DB_TXN *txnid, const char *file,
    const char *database, DBTYPE type, u_int32_t flags, int mode);
```

The `DB->open()` method opens the database represented by the **file** and **database**.

The currently supported Berkeley DB file formats (or *access methods*) are Btree, Hash, Queue, and Recno. The Btree format is a representation of a sorted, balanced tree structure. The Hash format is an extensible, dynamic hashing scheme. The Queue format supports fast access to fixed-length records accessed sequentially or by logical record number. The Recno format supports fixed- or variable-length records, accessed sequentially or by logical record number, and optionally backed by a flat text file.

Storage and retrieval for the Berkeley DB access methods are based on key/data pairs; see DBT for more information.

Calling `DB->open()` is a relatively expensive operation, and maintaining a set of open databases will normally be preferable to repeatedly opening and closing the database for each new query.

The `DB->open()` method returns a non-zero error value on failure and 0 on success. If `DB->open()` fails, the DB->close() (page 13) method must be called to discard the DB handle.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the DB_AUTO_COMMIT flag is specified, the operation will be implicitly transaction protected. Note that transactionally protected operations on a DB handle requires the DB handle itself be transactionally protected during its open. Also note that the transaction must be committed before the handle is closed; see Berkeley DB handles for more information.

file

The **file** parameter is used as the name of an underlying file that will be used to back the database; see File naming for more information.

In-memory databases never intended to be preserved on disk may be created by setting the **file** parameter to NULL. Whether other threads of control can access this database is driven entirely by whether the **database** parameter is set to NULL.

When using a Unicode build on Windows (the default), the **file** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

database

The **database** parameter is optional, and allows applications to have multiple databases in a single file. Although no **database** parameter needs to be specified, it is an error to attempt to open a second database in a **file** that was not initially created using a **database** name. Further, the **database** parameter is not supported by the Queue format. Finally, when opening multiple databases in the same physical file, it is important to consider locking and memory cache issues; see Opening multiple databases in a single file for more information.

If both the **database** and **file** parameters are NULL, the database is strictly temporary and cannot be opened by any other thread of control. Thus the database can only be accessed by sharing the single database handle that created it, in circumstances where doing so is safe.

If the **database** parameter is not set to NULL, the database can be opened by other threads of control and will be replicated to client sites in any replication group, regardless of whether the **file** parameter is set to NULL.

type

The **type** parameter is of type DBTYPE, and must be set to one of `DB_BTREE`, `DB_HASH`, `DB_QUEUE`, `DB_RECNO`, or `DB_UNKNOWN`. If **type** is DB_UNKNOWN, the database must already exist and `DB->open()` will automatically determine its type. The DB->get_type() (page 60) method may be used to determine the underlying type of databases opened using DB_UNKNOWN.

It is an error to specify the incorrect **type** for a database that already exists.

flags

The **flags** parameter must be set to zero or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_AUTO_COMMIT`

  Enclose the `DB->open()` call within a transaction. If the call succeeds, the open operation will be recoverable and all subsequent database modification operations based on this handle will be transactionally protected. If the call fails, no database will have been created.

- `DB_CREATE`

  Create the database. If the database does not already exist and the `DB_CREATE` flag is not specified, the `DB->open()` will fail.

- `DB_EXCL`

  Return an error if the database already exists. The `DB_EXCL` flag is only meaningful when specified with the `DB_CREATE`. flag.

- `DB_MULTIVERSION`

  Open the database with support for multiversion concurrency control. This will cause updates to the database to follow a copy-on-write protocol, which is required to support snapshot isolation. The `DB_MULTIVERSION` flag requires that the database be transactionally protected during its open and is not supported by the queue format.

- `DB_NOMMAP`

  Do not map this database into process memory (see the DB_ENV->set_mp_mmapsize() (page 399) method for further information).

- `DB_RDONLY`

  Open the database for reading only. Any attempt to modify items in the database will fail, regardless of the actual permissions of any underlying files.

- `DB_READ_UNCOMMITTED`

  Support transactional read operations with degree 1 isolation. Read operations on the database may request the return of modified but not yet committed data. This flag must be specified on all DB handles used to perform dirty reads or database updates, otherwise requests for dirty reads may not be honored and the read may block.

- `DB_THREAD`

  Cause the DB handle returned by `DB->open()` to be *free-threaded*; that is, concurrently usable by multiple threads in the address space.

- `DB_TRUNCATE`

  Physically truncate the underlying file, discarding all previous databases it might have held. Underlying filesystem primitives are used to implement this flag. For this reason, it is applicable only to the file and cannot be used to discard databases within a file.

  The `DB_TRUNCATE` flag cannot be lock or transaction-protected, and it is an error to specify it in a locking or transaction-protected environment.

mode

On Windows systems, the mode parameter is ignored.

On UNIX systems or in IEEE/ANSI Std 1003.1 (POSIX) environments, files created by the database open are created with mode **mode** (as described in **chmod**(2)) and modified by the process' umask value at the time of creation (see **umask**(2)). Created files are owned by the process owner; the group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB. System shared memory segments created by the database open are created with mode **mode**, unmodified by the process' umask value. If **mode** is 0, the database open will use a default mode of readable and writable by both owner and group.

Environment Variables

If the database was opened within a database environment, the environment variable **DB_HOME** may be used as the path of the database environment home.

DB->open() is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the "set_data_dir" string in the environment's DB_CONFIG file.

- **TMPDIR**

  If the **file** and **dbenv** parameters to DB->open() are NULL, the environment variable **TMPDIR** may be used as a directory in which to create temporary backing files

Errors

The DB->open() method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

ENOENT

The file or directory does not exist.

ENOENT

A nonexistent **re_source** file was specified.

DB_OLD_VERSION

The database cannot be opened without being first upgraded.

EEXIST

DB_CREATE and DB_EXCL were specified and the database exists.

EINVAL

If an unknown database type, page size, hash function, pad byte, byte order, or a flag value or parameter that is incompatible with the specified database was specified; the DB_THREAD flag was specified and fast mutexes are not available for this architecture; the DB_THREAD flag was specified to DB->open(), but was not specified to the DB_ENV->open() call for the environment in which the DB handle was created; a backing flat text file was specified with either the DB_THREAD flag or the provided database environment supports transaction processing; or if an invalid flag value or parameter was specified.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

Class

DB

See Also

Database and Related Methods (page 3)

DB->put()

```
#include <db.h>

int
DB->put(DB *db,
    DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
```

The `DB->put()` method stores key/data pairs in the database. The default behavior of the `DB->put()` function is to enter the new key/data pair, replacing any previously existing key if duplicates are disallowed, or adding a duplicate data item if duplicates are allowed. If the database supports duplicates, the `DB->put()` method adds the new data value at the end of the duplicate set. If the database supports sorted duplicates, the new data value is inserted at the correct sorted location.

Unless otherwise specified, the `DB->put()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

key

The key DBT operated on.

data

The data DBT operated on.

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_APPEND`

  Append the key/data pair to the end of the database. For the DB_APPEND flag to be specified, the underlying database must be a Queue or Recno database. The record number allocated to the record is returned in the specified **key**.

  There is a minor behavioral difference between the Recno and Queue access methods for the DB_APPEND flag. If a transaction enclosing a `DB->put()` operation with the DB_APPEND flag aborts, the record number may be reallocated in a subsequent `DB_APPEND` operation if you are using the Recno access method, but it will not be reallocated if you are using the Queue access method.

- DB_NODUPDATA

  In the case of the Btree and Hash access methods, enter the new key/data pair only if it does not already appear in the database.

  The DB_NODUPDATA flag may only be specified if the underlying database has been configured to support sorted duplicates. The DB_NODUPDATA flag may not be specified to the Queue or Recno access methods.

  The DB->put() method will return DB_KEYEXIST (page 169) if DB_NODUPDATA is set and the key/data pair already appears in the database.

- DB_NOOVERWRITE

  Enter the new key/data pair only if the key does not already appear in the database. The DB->put() method call with the DB_NOOVERWRITE flag set will fail if the key already exists in the database, even if the database supports duplicates.

  The DB->put() method will return DB_KEYEXIST (page 169) if DB_NOOVERWRITE is set and the key already appears in the database.

  This enforcement of uniqueness of keys applies only to the primary key. The behavior of insertions into secondary databases is not affected by the DB_NOOVERWRITE flag. In particular, the insertion of a record that would result in the creation of a duplicate key in a secondary database that allows duplicates would not be prevented by the use of this flag.

- DB_MULTIPLE

  Put multiple data items using keys from the buffer to which the **key** parameter refers and data values from the buffer to which the **data** parameter refers.

  To put records in bulk with the btree or hash access methods, construct bulk buffers in the **key** and **data** DBT using DB_MULTIPLE_WRITE_INIT (page 182) and DB_MULTIPLE_WRITE_NEXT (page 183). To put records in bulk with the recno or queue access methods, construct bulk buffers in the **data** DBT as before, but construct the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT (page 187) and DB_MULTIPLE_RECNO_WRITE_NEXT (page 188) with a data size of zero.

  A successful bulk operation is logically equivalent to a loop through each key/data pair, performing a DB->put() (page 71) for each one.

  See DBT and Bulk Operations (page 176) for more information on working with bulk updates.

  The DB_MULTIPLE flag may only be used alone, or with the DB_OVERWRITE_DUP option.

- DB_MULTIPLE_KEY

Put multiple data items using keys and data from the buffer to which the **key** parameter refers.

To put records in bulk with the btree or hash access methods, construct a single bulk buffer in the **key** DBT using DB_MULTIPLE_WRITE_INIT (page 182) and DB_MULTIPLE_KEY_WRITE_NEXT (page 185). To put records in bulk with the recno or queue access methods, construct a bulk buffer in the **key** DBT using DB_MULTIPLE_RECNO_WRITE_INIT (page 187) and DB_MULTIPLE_RECNO_WRITE_NEXT (page 188).

See DBT and Bulk Operations (page 176) for more information on working with bulk updates.

The `DB_MULTIPLE_KEY` flag may only be used alone, or with the `DB_OVERWRITE_DUP` option.

- `DB_OVERWRITE_DUP`

  Ignore duplicate records when overwriting records in a database configured for sorted duplicates.

  Normally, if a database is configured for sorted duplicates, an attempt to put a record that compares identically to a record already existing in the database will fail. Using this flag causes the put to silently proceed, without failure.

  This flag is extremely useful when performing bulk puts (using the `DB_MULTIPLE` or `DB_MULTIPLE_KEY` flags). Depending on the number of records you are writing to the database with a bulk put, you may not want the operation to fail in the event that a duplicate record is encountered. Using this flag along with the `DB_MULTIPLE` or `DB_MULTIPLE_KEY` flags allows the bulk put to complete, even if a duplicate record is encountered.

  This flag is also useful if you are using a custom comparison function that compares only part of the data portion of a record. In this case, two records can compare equally when, in fact, they are not equal. This flag allows the put to complete, even if your custom comparison routine claims the two records are equal.

Errors

The `DB->put()` method may fail and return one of the following non-zero errors:

DB_FOREIGN_CONFLICT

A foreign key constraint violation has occurred. This can be caused by one of two things:

1. An attempt was made to add a record to a constrained database, and the key used for that record does not exist in the foreign key database.
2. DB_FOREIGN_ABORT (page 11) was declared for a foreign key database, and then subsequently a record was deleted from the foreign key database without first removing it from the constrained secondary database.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EACCES

An attempt was made to modify a read-only database.

EINVAL

If a record number of 0 was specified; an attempt was made to add a record to a fixed-length database that was too large to fit; an attempt was made to do a partial put; an attempt was made to add a record to a secondary index; or if an invalid flag value or parameter was specified.

ENOSPC

A btree exceeded the maximum btree depth (255).

Class

DB

See Also

Database and Related Methods (page 3)

DB->remove()

```
#include <db.h>

int
DB->remove(DB *db,
    const char *file, const char *database, u_int32_t flags);
```

The `DB->remove()` method removes the database specified by the **file** and **database** parameters. If no **database** is specified, the underlying file represented by **file** is removed, incidentally removing all of the databases it contained.

Applications should never remove databases with open DB handles, or in the case of removing a file, when any database in the file has an open handle. For example, some architectures do not permit the removal of files with open system handles. On these architectures, attempts to remove databases currently in use by any thread of control in the system may fail.

The `DB->remove()` method should not be called if the remove is intended to be transactionally safe; the DB_ENV->dbremove() (page 200) method should be used instead.

The `DB->remove()` method may not be called after calling the DB->open() (page 66) method on any DB handle. If the DB->open() (page 66) method has already been called on a DB handle, close the existing handle and create a new one before calling `DB->remove. ()`

The DB handle may not be accessed again after `DB->remove()` is called, regardless of its return.

The `DB->remove()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The **file** parameter is the physical file which contains the database(s) to be removed.

database

The **database** parameter is the database to be removed.

flags

The **flags** parameter is currently unused, and must be set to 0.

Environment Variables

If the database was opened within a database environment, the environment variable `DB_HOME` may be used as the path of the database environment home.

`DB->remove()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the "set_data_dir" string in the environment's DB_CONFIG file.

Errors

The `DB->remove()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

ENOENT

The file or directory does not exist.

Class

DB

See Also

Database and Related Methods (page 3)

DB->rename()

```
#include <db.h>

int
DB->rename(DB *db, const char *file,
    const char *database, const char *newname, u_int32_t flags);
```

The `DB->rename()` method renames the database specified by the **file** and **database** parameters to **newname**. If no **database** is specified, the underlying file represented by **file** is renamed, incidentally renaming all of the databases it contained.

Applications should not rename databases that are currently in use. If an underlying file is being renamed and logging is currently enabled in the database environment, no database in the file may be open when the `DB->rename()` method is called. In particular, some architectures do not permit renaming files with open handles. On these architectures, attempts to rename databases that are currently in use by any thread of control in the system may fail.

The `DB->rename()` method should not be called if the rename is intended to be transactionally safe; the DB_ENV->dbrename() (page 202) method should be used instead.

The `DB->rename()` method may not be called after calling the DB->open() (page 66) method on any DB handle. If the DB->open() (page 66) method has already been called on a DB handle, close the existing handle and create a new one before calling `DB->rename()`.

The DB handle may not be accessed again after `DB->rename()` is called, regardless of its return.

The `DB->rename()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The **file** parameter is the physical file which contains the database(s) to be renamed.

When using a Unicode build on Windows (the default), the **file** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

database

The **database** parameter is the database to be renamed.

newname

The **newname** parameter is the new name of the database or file.

flags

The **flags** parameter is currently unused, and must be set to 0.

Environment Variables

If the database was opened within a database environment, the environment variable `DB_HOME` may be used as the path of the database environment home.

`DB->rename()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the "set_data_dir" string in the environment's DB_CONFIG file.

Errors

The `DB->rename()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

ENOENT

The file or directory does not exist.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_alloc()

```
#include <db.h>

int
DB->set_alloc(DB *db,
    void *(*app_malloc)(size_t),
    void *(*app_realloc)(void *, size_t),
    void (*app_free)(void *));
```

Set the allocation functions used by the DB_ENV and DB methods to allocate or free memory owned by the application.

There are a number of interfaces in Berkeley DB where memory is allocated by the library and then given to the application. For example, the DB_DBT_MALLOC flag, when specified in the DBT object, will cause the DB methods to allocate and reallocate memory which then becomes the responsibility of the calling application. (See DBT for more information.) Other examples are the Berkeley DB interfaces which return statistical information to the application: DB->stat() (page 130), DB_ENV->lock_stat() (page 319), DB_ENV->log_archive() (page 337), DB_ENV->log_stat() (page 351), DB_ENV->memp_stat() (page 383), and DB_ENV->txn_stat() (page 549). There is one method in Berkeley DB where memory is allocated by the application and then given to the library: DB->associate() (page 6).

On systems in which there may be multiple library versions of the standard allocation routines (notably Windows NT), transferring memory between the library and the application will fail because the Berkeley DB library allocates memory from a different heap than the application uses to free it. To avoid this problem, the DB_ENV->set_alloc() (page 239) and `DB->set_alloc()` methods can be used to pass Berkeley DB references to the application's allocation routines.

It is not an error to specify only one or two of the possible allocation function parameters to these interfaces; however, in that case the specified interfaces must be compatible with the standard library interfaces, as they will be used together. The functions specified must match the calling conventions of the ANSI C X3.159-1989 (ANSI C) library routines of the same name.

Because databases opened within Berkeley DB environments use the allocation interfaces specified to the environment, it is an error to attempt to set those interfaces in a database created within an environment.

The `DB->set_alloc()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_alloc()` method returns a non-zero error value on failure and 0 on success.

Errors

The `DB->set_alloc()` method may fail and return one of the following non-zero errors:

EINVAL

If called in a database environment, or called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_append_recno()

```
#include <db.h>

int
DB->set_append_recno(DB *,
    int (*db_append_recno_fcn)(DB *dbp, DBT *data, db_recno_t recno));
```

When using the DB_APPEND option of the DB->put() (page 71) method, it may be useful to modify the stored data based on the generated key. If a callback function is specified using the DB->set_append_recno() method, it will be called after the record number has been selected, but before the data has been stored.

The DB->set_append_recno() method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The DB->set_append_recno() method may not be called after the DB->open() (page 66) method is called.

The DB->set_append_recno() method returns a non-zero error value on failure and 0 on success.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_append_recno_fcn

The **db_append_recno_fcn** parameter is a function to call after the record number has been selected but before the data has been stored into the database. The function takes three parameters:

- dbp

  The **dbp** parameter is the enclosing database handle.

- data

  The **data** parameter is the data DBT to be stored.

- recno

  The **recno** parameter is the generated record number.

The called function may modify the data DBT. If the function needs to allocate memory for the **data** field, the **flags** field of the returned DBT should be set to DB_DBT_APPMALLOC, which indicates that Berkeley DB should free the memory when it is done with it.

The callback function must return 0 on success and **errno** or a value outside of the Berkeley DB error name space on failure.

Errors

The `DB->set_append_recno()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_bt_compare()

```
#include <db.h>

int
DB->set_bt_compare(DB *db,
    int (*bt_compare_fcn)(DB *db, const DBT *dbt1, const DBT *dbt2));
```

Set the Btree key comparison function. The comparison function is called whenever it is necessary to compare a key specified by the application with a key currently stored in the tree.

If no comparison function is specified, the keys are compared lexically, with shorter keys collating before longer keys.

The `DB->set_bt_compare()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_bt_compare()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_bt_compare()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_bt_compare()` method returns a non-zero error value on failure and 0 on success.

Parameters

bt_compare_fcn

The **bt_compare_fcn** function is the application-specified Btree comparison function. The comparison function takes three parameters:

- `db`

  The **db** parameter is the enclosing database handle.

- `dbt1`

  The **dbt1** parameter is the DBT representing the application supplied key.

- `dbt2`

  The **dbt2** parameter is the DBT representing the current tree's key.

The **bt_compare_fcn** function must return an integer value less than, equal to, or greater than zero if the first key parameter is considered to be respectively less than, equal to, or greater than the second key parameter. In addition, the comparison function must cause the keys in the database to be *well-ordered*. The comparison function must correctly handle any key values used by the application (possibly including zero-length keys). In addition, when Btree key prefix comparison is being performed (see DB->set_bt_prefix() (page 89) for more information), the comparison routine may be passed a prefix of any database key. The **data** and **size** fields of the DBT are the only fields that may be used for the purposes of this comparison, and no particular alignment of the memory to which by the **data** field refers may be assumed.

Errors

The `DB->set_bt_compare()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_bt_compress()

```
#include <db.h>

int
DB->set_bt_compress(DB *db,
    int (*bt_compress_fcn)(DB *db, const DBT *prevKey,
        const DBT *prevData, const DBT *key, const DBT *data, DBT *dest),
    int (*bt_decompress_fcn)(DB *db, const DBT *prevKey,
        const DBT *prevData, DBT *compressed, DBT *destKey,
        DBT *destData));
```

Set the Btree compression and decompression functions. The compression function is called whenever a key/data pair is added to the tree and the decompression function is called whenever data is requested from the tree.

If NULL function pointers are specified, then default compression and decompression functions are used. Berkeley DB's default compression function performs prefix compression on all keys and prefix compression on data values for duplicate keys. If using default compression, both the default compression and decompression functions must be used.

The `DB->set_bt_compress()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_bt_compress()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_bt_compress()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_bt_compress()` method returns a non-zero error value on failure and 0 on success.

Parameters

bt_compress_fcn

The **bt_compress_fcn** function is the application-specified Btree compression function. The compression function takes six parameters:

- `db`

  The **db** parameter is the enclosing database handle.

- `prevKey`

  The **prevKey** parameter is the DBT representing the key immediately preceding the application supplied key.

- `prevData`

  The **prevData** parameter is the DBT representing the data associated with **prevKey**.

- `key`

  The **key** parameter is the DBT representing the application supplied key.

- `data`

  The **data** parameter is the DBT representing the application supplied data.

- `dest`

  The **dest** parameter is the DBT representing the data stored in the tree, where the function should write the compressed data.

The **bt_compress_fcn** function must return 0 on success and a non-zero value on failure. If the compressed data cannot fit in **dest->data** (the size of which is stored in **dest->ulen**), the function should identify the required buffer size in **dest->size** and return `DB_BUFFER_SMALL`.

bt_decompress_fcn

The **bt_decompress_fcn** function is the application-specified Btree decompression function. The decompression function takes six parameters:

- `db`

  The **db** parameter is the enclosing database handle.

- `prevKey`

  The **prevKey** parameter is the DBT representing the key immediately preceding the key being decompressed.

- `prevData`

  The **prevData** parameter is the DBT representing the data associated with **prevKey**.

- `compressed`

  The **compressed** parameter is the DBT representing the data stored in the tree, that is, the compressed data.

- `destKey`

  The **key** parameter is the DBT where the decompression function should store the decompressed key.

- `destData`

  The **data** parameter is the DBT where the decompression function should store the decompressed key.

The **bt_decompress_fcn** function must return 0 on success and a non-zero value on failure. If the decompressed data cannot fit in **key->data** or **data->data** (the size of which is available in the DBT's **ulen** field), the function should identify the required buffer size using the DBT's **size** field and return `DB_BUFFER_SMALL`.

Errors

The `DB->set_bt_compress()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_bt_minkey()

```
#include <db.h>

int
DB->set_bt_minkey(DB *db, u_int32_t bt_minkey);
```

Set the minimum number of key/data pairs intended to be stored on any single Btree leaf page.

This value is used to determine if key or data items will be stored on overflow pages instead of Btree leaf pages. For more information on the specific algorithm used, see Minimum keys per page. The **bt_minkey** value specified must be at least 2; if **bt_minkey** is not explicitly set, a value of 2 is used.

The `DB->set_bt_minkey()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_bt_minkey()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_bt_minkey()` will be ignored.

The `DB->set_bt_minkey()` method returns a non-zero error value on failure and 0 on success.

Parameters

bt_minkey

The **bt_minkey** parameter is the minimum number of key/data pairs intended to be stored on any single Btree leaf page.

Errors

The `DB->set_bt_minkey()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_bt_prefix()

```
#include <db.h>

int
DB->set_bt_prefix(DB *db,
    size_t (*bt_prefix_fcn)(DB *, const *dbt1, const *dbt2));
```

Set the Btree prefix function. The prefix function is used to determine the amount by which keys stored on the Btree internal pages can be safely truncated without losing their uniqueness. See the Btree prefix comparison section of the Berkeley DB Reference Guide for more details about how this works. The usefulness of this is data-dependent, but can produce significantly reduced tree sizes and search times in some data sets.

If no prefix function or key comparison function is specified by the application, a default lexical comparison function is used as the prefix function. If no prefix function is specified and a key comparison function is specified, no prefix function is used. It is an error to specify a prefix function without also specifying a Btree key comparison function.

The `DB->set_bt_prefix()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_bt_prefix()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_bt_prefix()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_bt_prefix()` method returns a non-zero error value on failure and 0 on success.

Parameters

bt_prefix_fcn

The **bt_prefix_fcn** function is the application-specific Btree prefix function. The prefix function takes three parameters:

- `db`

  The **db** parameter is the enclosing database handle.

- `dbt1`

  The **dbt1** parameter is a DBT representing a database key.

- `dbt2`

  The **dbt2** parameter is a DBT representing a database key.

The **bt_prefix_fcn** function must return the number of bytes of the second key parameter that would be required by the Btree key comparison function to determine the second key parameter's ordering relationship with respect to the first key parameter. If the two keys are equal, the key length should be returned. The prefix function must correctly handle any key values used by the application (possibly including zero-length keys). The **data** and **size** fields of the DBT are the only fields that may be used for the purposes of this determination, and no particular alignment of the memory to which the **data** field refers may be assumed.

Errors

The `DB->set_bt_prefix()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_cachesize()

```
#include <db.h>

int
DB->set_cachesize(DB *db,
    u_int32_t gbytes, u_int32_t bytes, int ncache);
```

Set the size of the shared memory buffer pool -- that is, the cache. The cache should be the size of the normal working data set of the application, with some small amount of additional memory for unusual situations. (Note: the working set is not the same as the number of pages accessed simultaneously, and is usually much larger.)

The default cache size is 256KB, and may not be specified as less than 20KB. Any cache size less than 500MB is automatically increased by 25% to account for buffer pool overhead; cache sizes larger than 500MB are used as specified. The maximum size of a single cache is 4GB on 32-bit systems and 10TB on 64-bit systems. (All sizes are in powers-of-two, that is, 256KB is 2^18 not 256,000.) For information on tuning the Berkeley DB cache size, see Selecting a cache size.

It is possible to specify caches to Berkeley DB large enough they cannot be allocated contiguously on some architectures. For example, some releases of Solaris limit the amount of memory that may be allocated contiguously by a process. If **ncache** is 0 or 1, the cache will be allocated contiguously in memory. If it is greater than 1, the cache will be split across **ncache** separate regions, where the **region size** is equal to the initial cache size divided by **ncache**.

Because databases opened within Berkeley DB environments use the cache specified to the environment, it is an error to attempt to set a cache in a database created within an environment.

The `DB->set_cachesize()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytes

The size of the cache is set to **gbytes** gigabytes plus **bytes**.

bytes

The size of the cache is set to **gbytes** gigabytes plus **bytes**.

ncache

The **ncache** parameter is the number of caches to create.

Errors

The `DB->set_cachesize()` method may fail and return one of the following non-zero errors:

EINVAL

If the specified cache size was impossibly small; the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_create_dir()

```
#include <db.h>

int
DB->set_create_dir(DB *db, const char *dir);
```

Specify which directory a database should be created in or looked for.

The `DB->set_create_dir()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_create_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** will be used to create or locate the database file specified in the DB->open() (page 66) method call. The directory must be one of the directories in the environment list specified by DB_ENV->add_data_dir() (page 195).

Errors

The `DB->set_create_dir()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_dup_compare()

```
#include <db.h>

int
DB->set_dup_compare(DB *db,
    int (*dup_compare_fcn)(DB *db, const DBT *dbt1, const DBT *dbt2));
```

Set the duplicate data item comparison function. The comparison function is called whenever it is necessary to compare a data item specified by the application with a data item currently stored in the database. Calling `DB->set_dup_compare()` implies calling DB->set_flags() (page 104) with the DB_DUPSORT flag.

If no comparison function is specified, the data items are compared lexically, with shorter data items collating before longer data items.

The `DB->set_dup_compare()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_dup_compare()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_dup_compare()` method returns a non-zero error value on failure and 0 on success.

Parameters

dup_compare_fcn

The **dup_compare_fcn** function is the application-specified duplicate data item comparison function. The function takes three arguments:

- `db`

  The **db** parameter is the enclosing database handle.

- `dbt1`

  The **dbt1** parameter is a DBT representing the application supplied data item.

- `dbt2`

  The **dbt2** parameter is a DBT representing the current tree's data item.

The **dup_compare_fcn** function must return an integer value less than, equal to, or greater than zero if the first data item parameter is considered to be respectively less than, equal to, or greater than the second data item parameter. In addition, the comparison function must cause the data items in the set to be *well-ordered*. The comparison function must correctly handle any data item values used by the application (possibly including zero-length data items). The **data** and **size** fields of the DBT are the only fields that may be used for the purposes of this comparison, and no particular alignment of the memory to which the **data** field refers may be assumed.

Errors

The `DB->set_dup_compare()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_encrypt()

```
#include <db.h>

int
DB->set_encrypt(DB *db, const char *passwd, u_int32_t flags);
```

Set the password used by the Berkeley DB library to perform encryption and decryption.

Because databases opened within Berkeley DB environments use the password specified to the environment, it is an error to attempt to set a password in a database created within an environment.

The `DB->set_encrypt()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_encrypt()` method returns a non-zero error value on failure and 0 on success.

Parameters

passwd

The **passwd** parameter is the password used to perform encryption and decryption.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_ENCRYPT_AES`

  Use the Rijndael/AES (also known as the Advanced Encryption Standard and Federal Information Processing Standard (FIPS) 197) algorithm for encryption or decryption.

Errors

The `DB->set_encrypt()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

EOPNOTSUPP

Cryptography is not available in this Berkeley DB release.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_errcall()

```
#include <db.h>

void
DB->set_errcall(DB *, void (*db_errcall_fcn)
    (const DB_ENV *dbenv, const char *errpfx, const char *msg));
```

When an error occurs in the Berkeley DB library, a Berkeley DB error or an error return value is returned by the interface. In some cases, however, the **errno** value may be insufficient to completely describe the cause of the error, especially during initial application debugging.

The DB_ENV->set_errcall() (page 253) and `DB->set_errcall()` methods are used to enhance the mechanism for reporting error messages to the application. In some cases, when an error occurs, Berkeley DB will call **db_errcall_fcn()** with additional error information. It is up to the **db_errcall_fcn()** function to display the error message in an appropriate manner.

Setting **db_errcall_fcn** to NULL unconfigures the callback interface.

Alternatively, you can use the DB->set_errfile() (page 99) or DB->set_errfile() (page 255) methods to display the additional information via a C library `FILE *`.

This error-logging enhancement does not slow performance or significantly increase application size, and may be run during normal operation as well as during application debugging.

For DB handles opened inside of Berkeley DB environments, calling the `DB->set_errcall()` method affects the entire environment and is equivalent to calling the DB_ENV->set_errcall() (page 253) method.

When used on a database that was *not* opened in an environment, the `DB->set_errcall()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_errcall()` method may be called at any time during the life of the application.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_errcall_fcn

The **db_errcall_fcn** parameter is the application-specified error reporting function. The function takes three parameters:

- `dbenv`

The **dbenv** parameter is the enclosing database environment.

- `errpfx`

  The **errpfx** parameter is the prefix string (as previously set by DB->set_errpfx() (page 101) or DB_ENV->set_errpfx() (page 257)).

- `msg`

  The **msg** parameter is the error message string.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_errfile()

```
#include <db.h>

void
DB->set_errfile(DB *db, FILE *errfile);
```

When an error occurs in the Berkeley DB library, a Berkeley DB error or an error return value is returned by the interface. In some cases, however, the **errno** value may be insufficient to completely describe the cause of the error, especially during initial application debugging.

The DB_ENV->set_errfile() (page 255) and `DB->set_errfile()` methods are used to enhance the mechanism for reporting error messages to the application by setting a C library FILE * to be used for displaying additional Berkeley DB error messages. In some cases, when an error occurs, Berkeley DB will output an additional error message to the specified file reference.

Alternatively, you can use the DB_ENV->set_errcall() (page 253) or DB->set_errcall() (page 97) methods to capture the additional error information in a way that does not use C library FILE *'s.

The error message will consist of the prefix string and a colon ("**:**") (if a prefix string was previously specified using DB->set_errpfx() (page 101) or DB_ENV->set_errpfx() (page 257)), an error string, and a trailing <newline> character.

The default configuration when applications first create DB or DB_ENV handles is as if the DB_ENV->set_errfile() (page 255) or `DB->set_errfile()` methods were called with the standard error output (stderr) specified as the FILE * argument. Applications wanting no output at all can turn off this default configuration by calling the DB_ENV->set_errfile() (page 255) or `DB->set_errfile()` methods with NULL as the FILE * argument. Additionally, explicitly configuring the error output channel using any of the following methods will also turn off this default output for the application:

- `DB->set_errfile()`
- DB_ENV->set_errfile() (page 255)
- DB_ENV->set_errcall() (page 253)
- DB->set_errcall() (page 97)

This error logging enhancement does not slow performance or significantly increase application size, and may be run during normal operation as well as during application debugging.

For DB handles opened inside of Berkeley DB environments, calling the `DB->set_errfile()` method affects the entire environment and is equivalent to calling the DB_ENV->set_errfile() (page 255) method.

When used on a database that was *not* opened in an environment, the `DB->set_errfile()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_errfile()` method may be called at any time during the life of the application.

Parameters

errfile

The **errfile** parameter is a C library `FILE *` to be used for displaying additional Berkeley DB error information.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_errpfx()

```
#include <db.h>

void
DB->set_errpfx(DB *db, const char *errpfx);
```

Set the prefix string that appears before error messages issued by Berkeley DB.

The `DB->set_errpfx()` and DB_ENV->set_errpfx() (page 257) methods do not copy the memory to which the **errpfx** parameter refers; rather, they maintain a reference to it. Although this allows applications to modify the error message prefix at any time (without repeatedly calling the interfaces), it means the memory must be maintained until the handle is closed.

For DB handles opened inside of Berkeley DB environments, calling the `DB->set_errpfx()` method affects the entire environment and is equivalent to calling the DB_ENV->set_errpfx() (page 257) method.

The `DB->set_errpfx()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_errpfx()` method may be called at any time during the life of the application.

Parameters

errpfx

The **errpfx** parameter is the application-specified error prefix for additional error messages.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_feedback()

```
#include <db.h>

int
DB->set_feedback(DB *,
    void (*db_feedback_fcn)(DB *dbp, int opcode, int percent));
```

Some operations performed by the Berkeley DB library can take non-trivial amounts of time. The `DB->set_feedback()` method can be used by applications to monitor progress within these operations. When an operation is likely to take a long time, Berkeley DB will call the specified callback function with progress information.

It is up to the callback function to display this information in an appropriate manner.

The `DB->set_feedback()` method may be called at any time during the life of the application.

The `DB->set_feedback()` method returns a non-zero error value on failure and 0 on success.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_feedback_fcn

The **db_feedback_fcn** parameter is the application-specified feedback function called to report Berkeley DB operation progress. The callback function must take three parameters:

- `dbp`

  The **dbp** parameter is a reference to the enclosing database.

- `opcode`

  The **opcode** parameter is an operation code. The **opcode** parameter may take on any of the following values:

  - `DB_UPGRADE`

    The underlying database is being upgraded.

  - `DB_VERIFY`

    The underlying database is being verified.

- `percent`

The **percent** parameter is the percent of the operation that has been completed, specified as an integer value between 0 and 100.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_flags()

```
#include <db.h>

int
DB->set_flags(DB *db, u_int32_t flags);
```

Configure a database. Calling `DB->set_flags()` is additive; there is no way to clear flags.

The `DB->set_flags()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

General

The following flags may be specified for any Berkeley DB access method:

- `DB_CHKSUM`

  Do checksum verification of pages read into the cache from the backing filestore. Berkeley DB uses the SHA1 Secure Hash Algorithm if encryption is configured and a general hash algorithm if it is not.

  Calling `DB->set_flags()` with the DB_CHKSUM flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

  If the database already exists when DB->open() (page 66) is called, the DB_CHKSUM flag will be ignored.

- `DB_ENCRYPT`

  Encrypt the database using the cryptographic password specified to the DB_ENV->set_encrypt() (page 247) or DB->set_encrypt() (page 96) methods.

  Calling `DB->set_flags()` with the DB_ENCRYPT flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

  If the database already exists when DB->open() (page 66) is called, the DB_ENCRYPT flag must be the same as the existing database or an error will be returned.

  Encrypted databases are not portable between machines of different byte orders, that is, encrypted databases created on big-endian machines cannot be read on little-endian machines, and vice versa.

- `DB_TXN_NOT_DURABLE`

If set, Berkeley DB will not write log records for this database. This means that updates of this database exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the application or system fails, integrity will not persist. The database file must be verified and/or restored from backup after a failure. In order to ensure integrity after application shut down, the database handles must be closed without specifying DB_NOSYNC, or all database changes must be flushed from the database environment cache using either the DB_ENV->txn_checkpoint() (page 547) or DB_ENV->memp_sync() (page 390) methods. All database handles for a single physical file must set DB_TXN_NOT_DURABLE, including database handles for different databases in a physical file.

Calling `DB->set_flags()` with the DB_TXN_NOT_DURABLE flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

Btree

The following flags may be specified for the Btree access method:

- `DB_DUP`

  Permit duplicate data items in the database; that is, insertion when the key of the key/data pair being inserted already exists in the database will be successful. The ordering of duplicates in the database is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor operation or a duplicate sort function.

  The DB_DUPSORT flag is preferred to DB_DUP for performance reasons. The DB_DUP flag should only be used by applications wanting to order duplicate data items manually.

  Calling `DB->set_flags()` with the DB_DUP flag affects the database, including all threads of control accessing the database.

  If the database already exists when DB->open() (page 66) is called, the DB_DUP flag must be the same as the existing database or an error will be returned.

  It is an error to specify both DB_DUP and DB_RECNUM.

- `DB_DUPSORT`

  Permit duplicate data items in the database; that is, insertion when the key of the key/data pair being inserted already exists in the database will be successful. The ordering of duplicates in the database is determined by the duplicate comparison function. If the application does not specify a comparison function using the DB->set_dup_compare() (page 94) method, a default lexical comparison will be used. It is an error to specify both DB_DUPSORT and DB_RECNUM.

  Calling `DB->set_flags()` with the DB_DUPSORT flag affects the database, including all threads of control accessing the database.

  If the database already exists when DB->open() (page 66) is called, the DB_DUPSORT flag must be the same as the existing database or an error will be returned.

- DB_RECNUM

  Support retrieval from the Btree using record numbers. For more information, see the DB_SET_RECNO flag to the DB->get() (page 30) and DBcursor->get() (page 158) methods.

  Logical record numbers in Btree databases are mutable in the face of record insertion or deletion. See the DB_RENUMBER flag in the Recno access method information for further discussion.

  Maintaining record counts within a Btree introduces a serious point of contention, namely the page locations where the record counts are stored. In addition, the entire database must be locked during both insertions and deletions, effectively single-threading the database for those operations. Specifying DB_RECNUM can result in serious performance degradation for some applications and data sets.

  It is an error to specify both DB_DUP and DB_RECNUM.

  Calling DB->set_flags() with the DB_RECNUM flag affects the database, including all threads of control accessing the database.

  If the database already exists when DB->open() (page 66) is called, the DB_RECNUM flag must be the same as the existing database or an error will be returned.

- DB_REVSPLITOFF

  Turn off reverse splitting in the Btree. As pages are emptied in a database, the Berkeley DB Btree implementation attempts to coalesce empty pages into higher-level pages in order to keep the database as small as possible and minimize search time. This can hurt performance in applications with cyclical data demands; that is, applications where the database grows and shrinks repeatedly. For example, because Berkeley DB does page-level locking, the maximum level of concurrency in a database of two pages is far smaller than that in a database of 100 pages, so a database that has shrunk to a minimal size can cause severe deadlocking when a new cycle of data insertion begins.

  Calling DB->set_flags() with the DB_REVSPLITOFF flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

Hash

The following flags may be specified for the Hash access method:

- DB_DUP

  Permit duplicate data items in the database; that is, insertion when the key of the key/data pair being inserted already exists in the database will be successful. The ordering of duplicates in the database is determined by the order of insertion, unless the ordering is otherwise specified by use of a cursor operation.

  The DB_DUPSORT flag is preferred to DB_DUP for performance reasons. The DB_DUP flag should only be used by applications wanting to order duplicate data items manually.

Calling `DB->set_flags()` with the DB_DUP flag affects the database, including all threads of control accessing the database.

If the database already exists when DB->open() (page 66) is called, the DB_DUP flag must be the same as the existing database or an error will be returned.

- `DB_DUPSORT`

  Permit duplicate data items in the database; that is, insertion when the key of the key/data pair being inserted already exists in the database will be successful. The ordering of duplicates in the database is determined by the duplicate comparison function. If the application does not specify a comparison function using the DB->set_dup_compare() (page 94) method, a default lexical comparison will be used.

  Calling `DB->set_flags()` with the DB_DUPSORT flag affects the database, including all threads of control accessing the database.

  If the database already exists when DB->open() (page 66) is called, the DB_DUPSORT flag must be the same as the existing database or an error will be returned.

- `DB_REVSPLITOFF`

  Turns off hash bucket compaction. When a hash bucket is emptied, the Berkeley DB Hash implementation will decrease the hash table size, coalescing buckets. This will decrease the number of pages in the database. This can hurt performance in applications with cyclical data demands — that is, applications where the database grows and shrinks repeatedly — because of the cost of resplitting buckets when they grow again.

  Calling `DB->set_flags()` with the DB_REVSPLITOFF flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

Queue

The following flags may be specified for the Queue access method:

- `DB_INORDER`

  The DB_INORDER flag modifies the operation of the DB_CONSUME or DB_CONSUME_WAIT flags to DB->get() (page 30) to return key/data pairs in order. That is, they will always return the key/data item from the head of the queue.

  The default behavior of queue databases is optimized for multiple readers, and does not guarantee that record will be retrieved in the order they are added to the queue. Specifically, if a writing thread adds multiple records to an empty queue, reading threads may skip some of the initial records when the next DB->get() (page 30) call returns.

  This flag modifies the DB->get() (page 30) call to verify that the record being returned is in fact the head of the queue. This will increase contention and reduce concurrency when there are many reading threads.

Calling `DB->set_flags()` with the DB_INORDER flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

Recno

The following flags may be specified for the Recno access method:

- `DB_RENUMBER`

  Specifying the DB_RENUMBER flag causes the logical record numbers to be mutable, and change as records are added to and deleted from the database.

  Using the DB->put() (page 71) or DBcursor->put() (page 167) interfaces to create new records will cause the creation of multiple records if the record number is more than one greater than the largest record currently in the database. For example, creating record 28, when record 25 was previously the last record in the database, will create records 26 and 27 as well as 28. Attempts to retrieve records that were created in this manner will result in an error return of DB_KEYEMPTY.

  If a created record is not at the end of the database, all records following the new record will be automatically renumbered upward by one. For example, the creation of a new record numbered 8 causes records numbered 8 and greater to be renumbered upward by one. If a cursor was positioned to record number 8 or greater before the insertion, it will be shifted upward one logical record, continuing to refer to the same record as it did before.

  If a deleted record is not at the end of the database, all records following the removed record will be automatically renumbered downward by one. For example, deleting the record numbered 8 causes records numbered 9 and greater to be renumbered downward by one. If a cursor was positioned to record number 9 or greater before the removal, it will be shifted downward one logical record, continuing to refer to the same record as it did before.

  If a record is deleted, all cursors that were positioned on that record prior to the removal will no longer be positioned on a valid entry. This includes cursors used to delete an item. For example, if a cursor was positioned to record number 8 before the removal of that record, subsequent calls to DBcursor->get() (page 158) with flags of DB_CURRENT will result in an error return of DB_KEYEMPTY until the cursor is moved to another record. A call to DBcursor->get() (page 158) with flags of DB_NEXT will return the new record numbered 8 - which is the record that was numbered 9 prior to the delete (if such a record existed).

  For these reasons, concurrent access to a Recno database with the DB_RENUMBER flag specified may be largely meaningless, although it is supported.

  Calling `DB->set_flags()` with the DB_RENUMBER flag affects the database, including all threads of control accessing the database.

  If the database already exists when DB->open() (page 66) is called, the DB_RENUMBER flag must be the same as the existing database or an error will be returned.

- `DB_SNAPSHOT`

This flag specifies that any specified **re_source** file be read in its entirety when DB->open() (page 66) is called. If this flag is not specified, the **re_source** file may be read lazily.

See the DB->set_re_source() (page 128) method for information on the **re_source** file.

Calling `DB->set_flags()` with the DB_SNAPSHOT flag only affects the specified DB handle (and any other Berkeley DB handles opened within the scope of that handle).

Errors

The `DB->set_flags()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_h_compare()

```
#include <db.h>

int
DB->set_h_compare(DB *db,
    int (*compare_fcn)(DB *db, const DBT *dbt1, const DBT *dbt2));
```

Set the Hash key comparison function. The comparison function is called whenever it is necessary to compare a key specified by the application with a key currently stored in the database.

If no comparison function is specified, a byte-by-byte comparison is performed.

The `DB->set_h_compare()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_h_compare()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_h_compare()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_h_compare()` method returns a non-zero error value on failure and 0 on success.

Parameters

compare_fcn

The **compare_fcn** function is the application-specified Hash comparison function. The comparison function takes three parameters:

- `db`

  The **db** parameter is the enclosing database handle.

- `dbt1`

  The **dbt1** parameter is the DBT representing the application supplied key.

- `dbt2`

  The **dbt2** parameter is the DBT representing the current database's key.

The **compare_fcn** function must return an integer value less than, equal to, or greater than zero if the first key parameter is considered to be respectively less than, equal to, or greater than the second key parameter. The comparison function must correctly handle any key values used by the application (possibly including zero-length keys). The **data** and **size** fields of the DBT are the only fields that may be used for the purposes of this comparison, and no particular alignment of the memory to which by the **data** field refers may be assumed.

Errors

The `DB->set_h_compare()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_h_ffactor()

```
#include <db.h>

int
DB->set_h_ffactor(DB *db, u_int32_t h_ffactor);
```

Set the desired density within the hash table. If no value is specified, the fill factor will be selected dynamically as pages are filled.

The density is an approximation of the number of keys allowed to accumulate in any one bucket, determining when the hash table grows or shrinks. If you know the average sizes of the keys and data in your data set, setting the fill factor can enhance performance. A reasonable rule computing fill factor is to set it to the following:

```
              (pagesize - 32) / (average_key_size + average_data_size + 8)
```

The `DB->set_h_ffactor()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_h_ffactor()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_h_ffactor()` will be ignored.

The `DB->set_h_ffactor()` method returns a non-zero error value on failure and 0 on success.

Parameters

h_ffactor

The **h_ffactor** parameter is the desired density within the hash table.

Errors

The `DB->set_h_ffactor()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_h_hash()

```
#include <db.h>

int
DB->set_h_hash(DB *db,
    u_int32_t (*h_hash_fcn)(DB *dbp, const void *bytes,
    u_int32_t length));
```

Set a user-defined hash function; if no hash function is specified, a default hash function is used. Because no hash function performs equally well on all possible data, the user may find that the built-in hash function performs poorly with a particular data set.

The `DB->set_h_hash()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_h_hash()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_h_hash()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_h_hash()` method returns a non-zero error value on failure and 0 on success.

Parameters

h_hash_fcn

The **h_hash_fcn** parameter is the application-specified hash function.

Application-specified hash functions take a pointer to a byte string and a length as parameters, and return a value of type **u_int32_t**. The hash function must handle any key values used by the application (possibly including zero-length keys).

Errors

The `DB->set_h_hash()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_h_nelem()

```
#include <db.h>

int
DB->set_h_nelem(DB *db, u_int32_t h_nelem);
```

Set an estimate of the final size of the hash table.

In order for the estimate to be used when creating the database, the DB->set_h_ffactor() (page 112) method must also be called. If the estimate or fill factor are not set or are set too low, hash tables will still expand gracefully as keys are entered, although a slight performance degradation may be noticed.

The `DB->set_h_nelem()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_h_nelem()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_h_nelem()` will be ignored.

The `DB->set_h_nelem()` method returns a non-zero error value on failure and 0 on success.

Parameters

h_nelem

The **h_nelem** parameter is an estimate of the final size of the hash table.

Errors

The `DB->set_h_nelem()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_lorder()

```
#include <db.h>

int
DB->set_lorder(DB *db, int lorder);
```

Set the byte order for integers in the stored database metadata. The host byte order of the machine where the Berkeley DB library was compiled will be used if no byte order is set.

The access methods provide no guarantees about the byte ordering of the application data stored in the database, and applications are responsible for maintaining any necessary ordering.

The `DB->set_lorder()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_lorder()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_lorder()` will be ignored.

If creating additional databases in a single physical file, information specified to `DB->set_lorder()` will be ignored and the byte order of the existing databases will be used.

The `DB->set_lorder()` method returns a non-zero error value on failure and 0 on success.

Parameters

lorder

The **lorder** parameter should represent the byte order as an integer; for example, big endian order is the number 4,321, and little endian order is the number 1,234.

Errors

The `DB->set_lorder()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_msgcall()

```
#include <db.h>

void
DB->set_msgcall(DB *,
    void (*db_msgcall_fcn)(const DB_ENV *dbenv, char *msg));
```

There are interfaces in the Berkeley DB library which either directly output informational messages or statistical information, or configure the library to output such messages when performing other operations, for example, DB_ENV->set_verbose() (page 286) and DB_ENV->stat_print() (page 289).

The DB_ENV->set_msgcall() (page 271) and `DB->set_msgcall()` methods are used to pass these messages to the application, and Berkeley DB will call **db_msgcall_fcn** with each message. It is up to the **db_msgcall_fcn** function to display the message in an appropriate manner.

Setting **db_msgcall_fcn** to NULL unconfigures the callback interface.

Alternatively, you can use the DB->set_msgfile() (page 118) or DB->set_msgfile() (page 273) methods to display the messages via a C library FILE *.

For DB handles opened inside of Berkeley DB environments, calling the `DB->set_msgcall()` method affects the entire environment and is equivalent to calling the `DB_ENV->set_msgcall()` method.

The `DB->set_msgcall()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_msgcall()` method may be called at any time during the life of the application.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_msgcall_fcn

The **db_msgcall_fcn** parameter is the application-specified message reporting function. The function takes two parameters:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment.

- `msg`

The **msg** parameter is the message string.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_msgfile()

```
#include <db.h>

void
DB->set_msgfile(DB *db, FILE *msgfile);
```

There are interfaces in the Berkeley DB library which either directly output informational messages or statistical information, or configure the library to output such messages when performing other operations, for example, DB_ENV->set_verbose() (page 286) and DB_ENV->stat_print() (page 289).

The DB_ENV->set_msgfile() (page 273) and `DB->set_msgfile()` methods are used to display these messages for the application. In this case the message will include a trailing <newline> character.

Setting **msgfile** to NULL unconfigures the interface.

Alternatively, you can use the DB_ENV->set_msgcall() (page 271) or DB->set_msgcall() (page 116) methods to capture the additional error information in a way that does not use C library FILE *'s.

For DB handles opened inside of Berkeley DB environments, calling the `DB->set_msgfile()` method affects the entire environment and is equivalent to calling the DB_ENV->set_msgfile() (page 273) method.

The `DB->set_msgfile()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_msgfile()` method may be called at any time during the life of the application.

Parameters

msgfile

The **msgfile** parameter is a C library FILE * to be used for displaying messages.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_pagesize()

```
#include <db.h>

int
DB->set_pagesize(DB *db, u_int32_t pagesize);
```

Set the size of the pages used to hold items in the database, in bytes. The minimum page size is 512 bytes, the maximum page size is 64K bytes, and the page size must be a power-of-two. If the page size is not explicitly set, one is selected based on the underlying filesystem I/O block size. The automatically selected size has a lower limit of 512 bytes and an upper limit of 16K bytes.

For information on tuning the Berkeley DB page size, see Selecting a page size.

The `DB->set_pagesize()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_pagesize()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_pagesize()` will be ignored.

If creating additional databases in a single physical file, information specified to `DB->set_pagesize()` will be ignored and the page size of the existing databases will be used.

The `DB->set_pagesize()` method returns a non-zero error value on failure and 0 on success.

The `DB->set_pagesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

pagesize

The **pagesize** parameter sets the database page size.

Errors

The `DB->set_pagesize()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_partition()

```
#include <db.h>

int
DB->set_partition(DB * db, u_int32_t parts,  DBT *keys,
 u_int32_t (*db_partition_fcn) (DB *db, DBT *key));
```

Set up partitioning for a database. Partitioning may be used on either BTREE or HASH databases. Partitions may be specified by either a set of keys specifying a range of values in each partition or with a callback function that returns the number of the partition to put a specific key. Partition range keys may only be specified for BTREE databases.

Partitions are implimented as separate database files and can help reduce contention within a logical database. Contention can come from multiple threads of control accessing database pages simultaniously. Typically these pages are the root of a btree and the metadata page which contains allocation information in both BTREE and HASH databases. Each partition has its own metadata and root pages.

Parameters

Exactly one of the parameters **keys** and **partition_fcn** must be NULL.

parts

The **parts** parameter is the number of partitions to create. The value must be 2 or greater.

keys

The **keys** parameter is an array of DBT structures containing the keys that specify the range of key values to be stored in each partition. Each key specifies the minimum value that may be stored in the corresponding partition. The number of keys must be one less than the number of partitions specified by the **parts** parameter since the first partition will hold any key less than the first key in the array.

db_partition_fcn

The **db_partition_fcn** parameter is the application-specified partitioning function. The function returns an integer which will be used modulo the number of partitions specified by the **parts** parameter. The function will be called with two parameters:

- db

  The **db** parameter is the database handle.

- key

  The **key** parameter is the key for which a partition number should be returned.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_partition_dirs()

```
#include <db.h>

int
DB->set_partition_dirs(DB *db, const char **dirs);
```

Specify which directories the database extents should be created in or looked for. If the number of directories is less than the number of partitions, the directories will be used in a round robin fashion.

The `DB->set_partition_dirs()` method may not be called after the DB->open() (page 66) method is called.

The `DB->set_partition_dirs()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirs

The **dirs** points to an array of directories that will be used to create or locate the database extent files specified in the DB->open() (page 66) method call. The directories must be included in the environment list specified by DB_ENV->add_data_dir() (page 195).

Errors

The `DB->set_partition_dirs()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_priority()

```
#include <db.h>

int
DB->set_priority(DB *db, DB_CACHE_PRIORITY priority);
```

Set the cache priority for pages referenced by the DB handle.

The priority of a page biases the replacement algorithm to be more or less likely to discard a page when space is needed in the buffer pool. The bias is temporary, and pages will eventually be discarded if they are not referenced again. The `DB->set_priority()` method is only advisory, and does not guarantee pages will be treated in a specific way.

The `DB->set_priority()` method may be called at any time during the life of the application.

The `DB->set_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priority

The **priority** parameter must be set to one of the following values:

- `DB_PRIORITY_VERY_LOW`

  The lowest priority: pages are the most likely to be discarded.

- `DB_PRIORITY_LOW`

  The next lowest priority.

- `DB_PRIORITY_DEFAULT`

  The default priority.

- `DB_PRIORITY_HIGH`

  The next highest priority.

- `DB_PRIORITY_VERY_HIGH`

  The highest priority: pages are the least likely to be discarded.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_q_extentsize()

```
#include <db.h>

int
DB->set_q_extentsize(DB *db, u_int32_t extentsize);
```

Set the size of the extents used to hold pages in a Queue database, specified as a number of pages. Each extent is created as a separate physical file. If no extent size is set, the default behavior is to create only a single underlying database file.

For information on tuning the extent size, see Selecting a extent size.

The `DB->set_q_extentsize()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_q_extentsize()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_q_extentsize()` will be ignored.

The `DB->set_q_extentsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

extentsize

The **extentsize** parameter is the number of pages in a Queue database extent.

Errors

The `DB->set_q_extentsize()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_re_delim()

```
#include <db.h>

int
DB->set_re_delim(DB *db, int *re_delim);
```

Set the delimiting byte used to mark the end of a record in the backing source file for the Recno access method.

This byte is used for variable length records if the **re_source** file is specified using the DB->set_re_source() (page 128) method. If the **re_source** file is specified and no delimiting byte was specified, <newline> characters (that is, ASCII 0x0a) are interpreted as end-of-record markers.

The `DB->set_re_delim()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_re_delim()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_re_delim()` will be ignored.

The `DB->set_re_delim()` method returns a non-zero error value on failure and 0 on success.

Parameters

re_delim

The **re_delim** parameter is the delimiting byte used to mark the end of a record.

Errors

The `DB->set_re_delim()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_re_len()

```
#include <db.h>

int
DB->set_re_len(DB *db, u_int32_t re_len);
```

For the Queue access method, specify that the records are of length **re_len**. For the Queue access method, the record length must be enough smaller than the database's page size that at least one record plus the database page's metadata information can fit on each database page.

For the Recno access method, specify that the records are fixed-length, not byte-delimited, and are of length **re_len**.

Any records added to the database that are less than **re_len** bytes long are automatically padded (see DB->set_re_pad() (page 127) for more information).

Any attempt to insert records into the database that are greater than **re_len** bytes long will cause the call to fail immediately and return an error.

The `DB->set_re_len()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_re_len()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_re_len()` will be ignored.

The `DB->set_re_len()` method returns a non-zero error value on failure and 0 on success.

Parameters

re_len

The **re_len** parameter is the length of a Queue or Recno database record, in bytes.

Errors

The `DB->set_re_len()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_re_pad()

```
#include <db.h>

int
DB->set_re_pad(DB *db, int re_pad);
```

Set the padding character for short, fixed-length records for the Queue and Recno access methods.

If no pad character is specified, <space> characters (that is, ASCII 0x20) are used for padding.

The `DB->set_re_pad()` method configures a database, not only operations performed using the specified DB handle.

The `DB->set_re_pad()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_re_pad()` will be ignored.

The `DB->set_re_pad()` method returns a non-zero error value on failure and 0 on success.

Parameters

re_pad

The **re_pad** parameter is the pad character for fixed-length records for the Queue and Recno access methods.

Errors

The `DB->set_re_pad()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->set_re_source()

```
#include <db.h>

int
DB->set_re_source(DB *db, char *source);
```

Set the underlying source file for the Recno access method. The purpose of the **source** value is to provide fast access and modification to databases that are normally stored as flat text files.

The **source** parameter specifies an underlying flat text database file that is read to initialize a transient record number index. In the case of variable length records, the records are separated, as specified by DB->set_re_delim() (page 125). For example, standard UNIX byte stream files can be interpreted as a sequence of variable length records separated by <newline> characters.

In addition, when cached data would normally be written back to the underlying database file (for example, the DB->close() (page 13) or DB->sync() (page 138) methods are called), the in-memory copy of the database will be written back to the **source** file.

By default, the backing source file is read lazily; that is, records are not read from the file until they are requested by the application. **If multiple processes (not threads) are accessing a Recno database concurrently, and are either inserting or deleting records, the backing source file must be read in its entirety before more than a single process accesses the database, and only that process should specify the backing source file as part of the DB->open() (page 66) call. See the DB_SNAPSHOT flag for more information.**

Reading and writing the backing source file specified by source cannot be transaction-protected because it involves filesystem operations that are not part of the Db transaction methodology. For this reason, if a temporary database is used to hold the records, it is possible to lose the contents of the **source** file, for example, if the system crashes at the right instant. If a file is used to hold the database, normal database recovery on that file can be used to prevent information loss, although it is still possible that the contents of **source** will be lost if the system crashes.

The **source** file must already exist (but may be zero-length) when DB->open() (page 66) is called.

It is not an error to specify a read-only **source** file when creating a database, nor is it an error to modify the resulting database. However, any attempt to write the changes to the backing source file using either the DB->sync() (page 138) or DB->close() (page 13) methods will fail, of course. Specify the DB_NOSYNC flag to the DB->close() (page 13) method to stop it from attempting to write the changes to the backing file; instead, they will be silently discarded.

For all of the previous reasons, the **source** field is generally used to specify databases that are read-only for Berkeley DB applications; and that are either generated on the fly by software tools or modified using a different mechanism — for example, a text editor.

The `DB->set_re_source()` method configures operations performed using the specified DB handle, not all operations performed on the underlying database.

The `DB->set_re_source()` method may not be called after the DB->open() (page 66) method is called. If the database already exists when DB->open() (page 66) is called, the information specified to `DB->set_re_source()` must be the same as that historically used to create the database or corruption can occur.

The `DB->set_re_source()` method returns a non-zero error value on failure and 0 on success.

Parameters

source

The backing flat text database file for a Recno database.

When using a Unicode build on Windows (the default), the **source** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB->set_re_source()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->stat()

```
#include <db.h>

int
DB->stat(DB *db, DB_TXN *txnid, void *sp, u_int32_t flags);
```

The `DB->stat()` method creates a statistical structure and copies a pointer to it into user-specified memory locations. Specifically, if **sp** is non-NULL, a pointer to the statistics for the database are copied into the memory location to which it refers.

The `DB->stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_FAST_STAT`

  Return only the values which do not require traversal of the database. Among other things, this flag makes it possible for applications to request key and record counts without incurring the performance penalty of traversing the entire database.

- `DB_READ_COMMITTED`

  Database items read during a transactional call will have degree 2 isolation. This ensures the stability of the data items read during the stat operation but permits that data to be modified or deleted by other transactions prior to the commit of the specified transaction.

- `DB_READ_UNCOMMITTED`

  Database items read during a transactional call will have degree 1 isolation, including modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

Statistical Structure

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

If the DB_FAST_STAT flag has not been specified, the `DB->stat()` method will access some of or all the pages in the database, incurring a severe performance penalty as well as possibly flushing the underlying buffer pool.

In the presence of multiple threads or processes accessing an active database, the information returned by DB->stat may be out-of-date.

If the database was not opened read-only and the DB_FAST_STAT flag was not specified, the cached key and record numbers will be updated after the statistical information has been gathered.

The `DB->stat()` method may not be called before the DB->open() (page 66) method is called.

The `DB->stat()` method returns a non-zero error value on failure and 0 on success.

Hash Statistics

In the case of a Hash database, the statistics are stored in a structure of type `DB_HASH_STAT`. The following fields will be filled in:

- `u_int32_t hash_magic;`

  Magic number that identifies the file as a Hash file. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_version;`

  The version of the Hash database. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_nkeys;`

  The number of unique keys in the database. If DB_FAST_STAT was specified the count will be the last saved value unless it has never been calculated, in which case it will be 0. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_ndata;`

  The number of key/data pairs in the database. If DB_FAST_STAT was specified the count will be the last saved value unless it has never been calculated, in which case it will be 0. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_pagecnt;`

  The number of pages in the database. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_pagesize;`

  The underlying database page (and bucket) size, in bytes. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_ffactor;`

The desired fill factor (number of items per bucket) specified at database-creation time. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_buckets;`

  The number of hash buckets. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_free;`

  The number of pages on the free list.

- `uintmax_t hash_bfree;`

  The number of bytes free on bucket pages.

- `u_int32_t hash_bigpages;`

  The number of big key/data pages.

- `uintmax_t hash_big_bfree;`

  The number of bytes free on big item pages.

- `u_int32_t hash_overflows;`

  The number of overflow pages (overflow pages are pages that contain items that did not fit in the main bucket page).

- `uintmax_t hash_ovfl_free;`

  The number of bytes free on overflow pages.

- `u_int32_t hash_dup;`

  The number of duplicate pages.

- `uintmax_t hash_dup_free;`

  The number of bytes free on duplicate pages.

Btree and Recno Statistics

In the case of a Btree or Recno database, the statistics are stored in a structure of type `DB_BTREE_STAT`. The following fields will be filled in:

- `u_int32_t bt_magic;`

  Magic number that identifies the file as a Btree database. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_version;`

  The version of the Btree database. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_nkeys;`

  For the Btree Access Method, the number of keys in the database. If the DB_FAST_STAT flag is not specified or the database was configured to support record numbers (see DB_RECNUM), the count will be exact. Otherwise, the count will be the last saved value unless it has never been calculated, in which case it will be 0.

  For the Recno Access Method, the number of records in the database. If the database was configured with mutable record numbers (see DB_RENUMBER), the count will be exact. Otherwise, if the DB_FAST_STAT flag is specified the count will be exact but will include deleted and implicitly created records; if the DB_FAST_STAT flag is not specified, the count will be exact and will not include deleted or implicitly created records.

  Returned if DB_FAST_STAT is set.

- `u_int32_t bt_ndata;`

  For the Btree Access Method, the number of key/data pairs in the database. If the DB_FAST_STAT flag is not specified, the count will be exact. Otherwise, the count will be the last saved value unless it has never been calculated, in which case it will be 0.

  For the Recno Access Method, the number of records in the database. If the database was configured with mutable record numbers (see DB_RENUMBER), the count will be exact. Otherwise, if the DB_FAST_STAT flag is specified the count will be exact but will include deleted and implicitly created records; if the DB_FAST_STAT flag is not specified, the count will be exact and will not include deleted or implicitly created records.

  Returned if DB_FAST_STAT is set.

- `u_int32_t bt_pagecnt;`

  The number of pages in the database. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_pagesize;`

  The underlying database page size, in bytes. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_minkey;`

  The minimum keys per page. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_re_len;`

  The length of fixed-length records. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_re_pad;`

  The padding byte value for fixed-length records. Returned if DB_FAST_STAT is set.

- `u_int32_t bt_levels;`

  Number of levels in the database.

- `u_int32_t bt_int_pg;`

  Number of database internal pages.

- `u_int32_t bt_leaf_pg;`

  Number of database leaf pages.

- `u_int32_t bt_dup_pg;`

  Number of database duplicate pages.

- `u_int32_t bt_over_pg;`

  Number of database overflow pages.

- `u_int32_t bt_empty_pg;`

  Number of empty database pages.

- `u_int32_t bt_free;`

  Number of pages on the free list.

- `uintmax_t bt_int_pgfree;`

  Number of bytes free in database internal pages.

- `uintmax_t bt_leaf_pgfree;`

  Number of bytes free in database leaf pages.

- `uintmax_t bt_dup_pgfree;`

  Number of bytes free in database duplicate pages.

- `uintmax_t bt_over_pgfree;`

  Number of bytes free in database overflow pages.

Queue Statistics

In the case of a Queue database, the statistics are stored in a structure of type `DB_QUEUE_STAT`. The following fields will be filled in:

- `u_int32_t qs_magic;`

  Magic number that identifies the file as a Queue file. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_version;`

  The version of the Queue file type. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_nkeys;`

  The number of records in the database. If DB_FAST_STAT was specified the count will be the last saved value unless it has never been calculated, in which case it will be 0. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_ndata;`

  The number of records in the database. If DB_FAST_STAT was specified the count will be the last saved value unless it has never been calculated, in which case it will be 0. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_pagesize;`

  Underlying database page size, in bytes. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_extentsize;`

  Underlying database extent size, in pages. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_pages;`

  Number of pages in the database.

- `u_int32_t qs_re_len;`

  The length of the records. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_re_pad;`

  The padding byte value for the records. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_pgfree;`

  Number of bytes free in database pages.

- `u_int32_t qs_first_recno;`

  First undeleted record in the database. Returned if DB_FAST_STAT is set.

- `u_int32_t qs_cur_recno;`

  Next available record number. Returned if DB_FAST_STAT is set.

Errors

The `DB->stat()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->stat_print()

```
#include <db.h>

int
DB->stat_print(DB *db, u_int32_t flags);
```

The `DB->stat_print()` method displays the database statistical information, as described for the `DB->stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB->stat_print()` method may not be called before the DB->open() (page 66) method is called.

The `DB->stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_FAST_STAT`

  Return only the values which do not require traversal of the database. Among other things, this flag makes it possible for applications to request key and record counts without incurring the performance penalty of traversing the entire database.

- `DB_STAT_ALL`

  Display all available information.

Class

DB

See Also

Database and Related Methods (page 3)

DB->sync()

```
#include <db.h>

int
DB->sync(DB *db, u_int32_t flags);
```

The `DB->sync()` method flushes any cached information to disk.

If the database is in memory only, the `DB->sync()` method has no effect and will always succeed.

It is important to understand that flushing cached information to disk only minimizes the window of opportunity for corrupted data. Although unlikely, it is possible for database corruption to happen if a system or application crash occurs while writing data to the database. To ensure that database corruption never occurs, applications must either: use transactions and logging with automatic recovery; use logging and application-specific recovery; or edit a copy of the database, and once all applications using the database have successfully called DB->close() (page 13), atomically replace the original database with the updated copy.

The `DB->sync()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB->sync()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->truncate()

```
#include <db.h>

int
DB->truncate(DB *db,
    DB_TXN *txnid, u_int32_t *countp, u_int32_t flags);
```

The `DB->truncate()` method empties the database, discarding all records it contains. The number of records discarded from the database is returned in **countp**.

When called on a database configured with secondary indices using the DB->associate() (page 6) method, the `DB->truncate()` method truncates the primary database and all secondary indices. A count of the records discarded from the primary database is returned.

It is an error to call the `DB->truncate()` method on a database with open cursors.

The `DB->truncate()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

countp

The **countp** parameter references memory into which the number of records discarded from the database is copied.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB->truncate()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If there are open cursors in the database; or if an invalid flag value or parameter was specified.

Class

DB

See Also

Database and Related Methods (page 3)

DB->upgrade()

```
#include <db.h>

int
DB->upgrade(DB *db, const char *file, u_int32_t flags);
```

The `DB->upgrade()` method upgrades all of the databases included in the file **file**, if necessary. If no upgrade is necessary, `DB->upgrade()` always returns success.

Database upgrades are done in place and are destructive. For example, if pages need to be allocated and no disk space is available, the database may be left corrupted. Backups should be made before databases are upgraded. See Upgrading databases for more information.

Unlike all other database operations, `DB->upgrade()` may only be done on a system with the same byte-order as the database.

The `DB->upgrade()` method returns a non-zero error value on failure and 0 on success.

The `DB->upgrade()` method is the underlying method used by the db_upgrade utility. See the db_upgrade utility source code for an example of using `DB->upgrade()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

Parameters

file

The **file** parameter is the physical file containing the databases to be upgraded.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_DUPSORT`

  This flag is only meaningful when upgrading databases from releases before the Berkeley DB 3.1 release.

  As part of the upgrade from the Berkeley DB 3.0 release to the 3.1 release, the on-disk format of duplicate data items changed. To correctly upgrade the format requires applications to specify whether duplicate data items in the database are sorted or not. Specifying the DB_DUPSORT flag informs `DB->upgrade()` that the duplicates are sorted; otherwise they are assumed to be unsorted. Incorrectly specifying the value of this flag may lead to database corruption.

  Further, because the `DB->upgrade()` method upgrades a physical file (including all the databases it contains), it is not possible to use `DB->upgrade()` to upgrade files in which some of the databases it includes have sorted duplicate data items, and some of the databases it includes have unsorted duplicate data items. If the file does not have more than a single database, if the databases do not support duplicate data items, or if all of the databases that support duplicate data items support the same style of duplicates (either sorted or unsorted), `DB->upgrade()` will work correctly as long as the DB_DUPSORT flag is correctly specified. Otherwise, the file cannot be upgraded using `DB->upgrade;()` it must be upgraded manually by dumping and reloading the databases.

Environment Variables

If the database was opened within a database environment, the environment variable `DB_HOME` may be used as the path of the database environment home.

`DB->upgrade()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the "set_data_dir" string in the environment's DB_CONFIG file.

Errors

The `DB->upgrade()` method may fail and return one of the following non-zero errors:

DB_OLD_VERSION

The database cannot be upgraded by this version of the Berkeley DB software.

Class

DB

See Also

Database and Related Methods (page 3)

DB->verify()

```
#include <db.h>

int
DB->verify(DB *db, const char *file,
    const char *database, FILE *outfile, u_int32_t flags);
```

The `DB->verify()` method verifies the integrity of all databases in the file specified by the **file** parameter, and optionally outputs the databases' key/data pairs to the file stream specified by the **outfile** parameter.

The `DB->verify()` method does not perform any locking, even in Berkeley DB environments that are configured with a locking subsystem. As such, it should only be used on files that are not being modified by another thread of control.

The `DB->verify()` method may not be called after the DB->open() (page 66) method is called.

The DB handle may not be accessed again after `DB->verify()` is called, regardless of its return.

The `DB->verify()` method is the underlying method used by the db_verify utility. See the db_verify utility source code for an example of using `DB->verify()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The `DB->verify()` method will return DB_VERIFY_BAD if a database is corrupted. When the DB_SALVAGE flag is specified, the DB_VERIFY_BAD return means that all key/data pairs in the file may not have been successfully output. Unless otherwise specified, the `DB->verify()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The **file** parameter is the physical file in which the databases to be verified are found.

database

The **database** parameter is the database in **file** on which the database checks for btree and duplicate sort order and for hashing are to be performed. See the DB_ORDERCHKONLY flag for more information.

The database parameter must be set to NULL except when the DB_ORDERCHKONLY flag is set.

outfile

The **outfile** parameter is an optional file stream to which the databases' key/data pairs are written.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_SALVAGE`

  Write the key/data pairs from all databases in the file to the file stream named in the **outfile** parameter. Key values are written for Btree, Hash and Queue databases, but not for Recno databases.

  The output format is the same as that specified for the db_dump utility, and can be used as input for the db_load utility.

  Because the key/data pairs are output in page order as opposed to the sort order used by db_dump, using `DB->verify()` to dump key/data pairs normally produces less than optimal loads for Btree databases.

In addition, the following flags may be set by bitwise inclusively **OR**'ing them into the **flags** parameter:

- `DB_AGGRESSIVE`

  Output **all** the key/data pairs in the file that can be found. By default, `DB->verify()` does not assume corruption. For example, if a key/data pair on a page is marked as deleted, it is not then written to the output file. When DB_AGGRESSIVE is specified, corruption is assumed, and any key/data pair that can be found is written. In this case, key/data pairs that are corrupted or have been deleted may appear in the output (even if the file being salvaged is in no way corrupt), and the output will almost certainly require editing before being loaded into a database.

- `DB_PRINTABLE`

  When using the DB_SALVAGE flag, if characters in either the key or data items are printing characters (as defined by **isprint**(3)), use printing characters to represent them. This flag permits users to use standard text editors and tools to modify the contents of databases or selectively remove data from salvager output.

  Note: different systems may have different notions about what characters are considered *printing characters*, and databases dumped in this manner may be less portable to external systems.

- `DB_NOORDERCHK`

  Skip the database checks for btree and duplicate sort order and for hashing.

  The `DB->verify()` method normally verifies that btree keys and duplicate items are correctly sorted, and hash keys are correctly hashed. If the file being verified contains multiple databases using differing sorting or hashing algorithms, some of them must necessarily fail database verification because only one sort order or hash function can be specified before `DB->verify()` is called. To verify files with multiple databases having differing sorting orders or hashing functions, first perform verification of the file as a whole by using the DB_NOORDERCHK flag, and then individually verify the sort order and hashing function for each database in the file using the DB_ORDERCHKONLY flag.

- `DB_ORDERCHKONLY`

Perform the database checks for btree and duplicate sort order and for hashing, skipped by DB_NOORDERCHK.

When this flag is specified, a **database** parameter should also be specified, indicating the database in the physical file which is to be checked. This flag is only safe to use on databases that have already successfully been verified using `DB->verify()` with the DB_NOORDERCHK flag set.

Environment Variables

If the database was opened within a database environment, the environment variable `DB_HOME` may be used as the path of the database environment home.

`DB->verify()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the "set_data_dir" string in the environment's DB_CONFIG file.

Errors

The `DB->verify()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB->open() (page 66) was called; or if an invalid flag value or parameter was specified.

ENOENT

The file or directory does not exist.

Class

DB

See Also

Database and Related Methods (page 3)

Chapter 3. The DBcursor Handle

A DBcursor object is a handle for a cursor into a Berkeley DB database.

DBcursor handles are not free-threaded. Cursor handles may be shared by multiple threads if access is serialized by the application.

You create a DBcursor using the DB->cursor() (page 149) method.

If the cursor is to be used to perform operations on behalf of a transaction, the cursor must be opened and closed within the context of that single transaction.

Once DBcursor->close() (page 151) has been called, the handle may not be accessed again, regardless of the method's return.

Database Cursors and Related Methods

Database Cursors and Related Methods	Description
DB->cursor()	Create a cursor handle
DBcursor->close()	Close a cursor handle
DBcursor->cmp()	Compare two cursors for equality.
DBcursor->count()	Return count of duplicates for current key
DBcursor->del()	Delete current key/data pair
DBcursor->dup()	Duplicate the cursor handle
DBcursor->get()	Retrieve by cursor
DBcursor->put()	Store by cursor
DBcursor->set_priority(), DBcursor->get_priority()	Set/get the cursor's cache priority

DB->cursor()

```
#include <db.h>

int
DB->cursor(DB *db, DB_TXN *txnid, DBC **cursorp, u_int32_t flags);
```

The `DB->cursor()` method returns a created database cursor.

Cursors may span threads, but only serially, that is, the application must serialize access to the cursor handle.

The `DB->cursor()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

To transaction-protect cursor operations, cursors must be opened and closed within the context of a transaction. The **txnid** parameter specifies the transaction context in which the cursor may be used.

Cursor operations are not automatically transaction-protected, even if the DB_AUTO_COMMIT flag is specified to the DB_ENV->set_flags() (page 260) or DB->open() (page 66) methods. If cursor operations are to be transaction-protected, the **txnid** parameter must be a transaction handle returned from DB_ENV->txn_begin() (page 544); otherwise, NULL.

cursorp

The **cursorp** parameter references memory into which a pointer to the allocated cursor is copied.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_BULK`

  Configure a cursor to optimize for bulk operations. Each successive operation on a cursor configured with this flag attempts to continue on the same database page as the previous operation, falling back to a search if a different page is required. This avoids searching if there is a high degree of locality between cursor operations. This flag is currently only effective with the btree access method: for other access methods it is ignored.

- `DB_READ_COMMITTED`

  Configure a transactional cursor to have degree 2 isolation. This ensures the stability of the current data item read by this cursor but permits data read by this cursor to be modified or deleted prior to the commit of the transaction for this cursor.

- `DB_READ_UNCOMMITTED`

Configure a transactional cursor to have degree 1 isolation. Read operations performed by the cursor may return modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

- `DB_WRITECURSOR`

  Specify that the cursor will be used to update the database. The underlying database environment must have been opened using the DB_INIT_CDB flag.

- `DB_TXN_SNAPSHOT`

  Configure a transactional cursor to operate with read-only snapshot isolation. For databases with the DB_MULTIVERSION flag set, data values will be read as they are when the cursor is opened, without taking read locks.

  This flag implicitly begins a transaction that is committed when the cursor is closed.

  This flag is silently ignored if DB_MULTIVERSION is not set on the underlying database or if a transaction is supplied in the **txnid** parameter.

Errors

The `DB->cursor()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

An invalid flag value or parameter was specified.

Class

DB

See Also

Database Cursors and Related Methods (page 148)

DBcursor->close()

```
#include <db.h>

int
DBcursor->close(DBC *DBcursor);
```

The `DBcursor->close()` method discards the cursor.

It is possible for the `DBcursor->close()` method to return DB_LOCK_DEADLOCK, signaling that any enclosing transaction should be aborted. If the application is already intending to abort the transaction, this error should be ignored, and the application should proceed.

After the `DBcursor->close()` method has been called, regardless of its return value, you can not use the cursor handle again.

It is not required to close the cursor explicitly before closing the database handle or the transaction handle that owns this cursor because, closing a database handle or a transaction handle closes those open cursors.

However, it is recommended that you always close all cursor handles immediately after their use to promote concurrency and to release resources such as page locks.

The `DBcursor->close()` method returns a non-zero error value on failure and 0 on success.

Errors

The `DBcursor->close()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->cmp()

```
#include <db.h>

int
DBcursor->cmp(DBC *DBcursor,
    DBC *other_cursor, int *result, u_int32_t flags);
```

The `DBcursor->cmp()` method compares two cursors for equality. Two cursors are equal if and only if they are positioned on the same item in the same database.

The `DBcursor->cmp()` method returns a non-zero error value on failure and 0 on success.

Parameters

other_cursor

The **other_cursor** parameter references another cursor handle that will be used as the comparator.

result

If the call is successful and both cursors are positioned on the same item, result is set to zero. If the call is successful but the cursors are not positioned on the same item, result is set to a non-zero value. If the call is unsuccessful, the value of result should be ignored.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DBcursor->cmp()` method may fail and return one of the following non-zero errors:

EINVAL

- If either of the cursors are already closed.
- If the cursors have been opened against different databases.
- If either of the cursors have not been positioned.
- If the other_dbc parameter is NULL.
- If the result parameter is NULL.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->count()

```
#include <db.h>

int
DBcursor->count(DBC *DBcursor, db_recno_t *countp, u_int32_t flags);
```

The `DBcursor->count()` method returns a count of the number of data items for the key to which the cursor refers.

The `DBcursor->count()` method returns a non-zero error value on failure and 0 on success.

Parameters

countp

The **countp** parameter references memory into which the count of the number of duplicate data items is copied.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DBcursor->count()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

If the cursor has not been initialized; or if an invalid flag value or parameter was specified.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->del()

```
#include <db.h>

int
DBcursor->del(DBC *DBcursor, u_int32_t flags);
```

The `DBcursor->del()` method deletes the key/data pair to which the cursor refers.

When called on a cursor opened on a database that has been made into a secondary index using the DB->associate() (page 6) method, the DB->del() (page 22) method deletes the key/data pair from the primary database and all secondary indices.

The cursor position is unchanged after a delete, and subsequent calls to cursor functions expecting the cursor to refer to an existing key will fail.

The `DBcursor->del()` method will return DB_KEYEMPTY if the element has already been deleted. The `DBcursor->del()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_CONSUME`

  If the database is of type DB_QUEUE then this flag may be set to force the head of the queue to move to the first non-deleted item in the queue. Normally this is only done if the deleted item is exactly at the head when deleted.

Errors

The `DBcursor->del()` method may fail and return one of the following non-zero errors:

DB_FOREIGN_CONFLICT

A foreign key constraint violation has occurred. This can be caused by one of two things:

1. An attempt was made to add a record to a constrained database, and the key used for that record does not exist in the foreign key database.

2. DB_FOREIGN_ABORT (page 11) was declared for a foreign key database, and then subsequently a record was deleted from the foreign key database without first removing it from the constrained secondary database.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DB_SECONDARY_BAD

A secondary index references a nonexistent primary key.

EACCES

An attempt was made to modify a read-only database.

EINVAL

If the cursor has not been initialized; or if an invalid flag value or parameter was specified.

EPERM

Write attempted on read-only cursor when the DB_INIT_CDB flag was specified to DB_ENV->open() (page 231).

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->dup()

```
#include <db.h>

int
DBcursor->dup(DBC *DBcursor, DBC **cursorp, u_int32_t flags);
```

The `DBcursor->dup()` method creates a new cursor that uses the same transaction and locker ID as the original cursor. This is useful when an application is using locking and requires two or more cursors in the same thread of control.

The `DBcursor->dup()` method returns a non-zero error value on failure and 0 on success.

Parameters

cursorp

The `DBcursor->dup()` method returns the newly created cursor in **cursorp**.

flags

The **flags** parameter must be set to 0 or the following flag:

- `DB_POSITION`

  The newly created cursor is initialized to refer to the same position in the database as the original cursor (if any) and hold the same locks (if any). If the DB_POSITION flag is not specified, or the original cursor does not hold a database position and locks, the created cursor is uninitialized and will behave like a cursor newly created using the DB->cursor() (page 149) method.

Errors

The `DBcursor->dup()` method may fail and return one of the following non-zero errors:

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EINVAL

An invalid flag value or parameter was specified.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->get()

```
#include <db.h>

int
DBcursor->get(DBC *DBcursor,
    DBT *key, DBT *data, u_int32_t flags);

int
DBcursor->pget(DBC *DBcursor,
    DBT *key, DBT *pkey, DBT *data, u_int32_t flags);
```

The `DBcursor->get()` method retrieves key/data pairs from the database. The address and length of the key are returned in the object to which **key** refers (except for the case of the DB_SET flag, in which the **key** object is unchanged), and the address and length of the data are returned in the object to which **data** refers.

When called on a cursor opened on a database that has been made into a secondary index using the DB->associate() (page 6) method, the `DBcursor->get()` and `DBcursor->pget()` methods return the key from the secondary index and the data item from the primary database. In addition, the `DBcursor->pget()` method returns the key from the primary database. In databases that are not secondary indices, the `DBcursor->pget()` method will always fail.

Modifications to the database during a sequential scan will be reflected in the scan; that is, records inserted behind a cursor will not be returned while records inserted in front of a cursor will be returned.

In Queue and Recno databases, missing entries (that is, entries that were never explicitly created or that were created and then deleted) will be skipped during a sequential scan.

Unless otherwise specified, the `DBcursor->get()` method returns a non-zero error value on failure and 0 on success.

If `DBcursor->get()` fails for any reason, the state of the cursor will be unchanged.

Parameters

key

The key DBT operated on.

pkey

The secondary index key DBT operated on.

data

The data DBT operated on.

flags

The **flags** parameter must be set to one of the following values:

- DB_CURRENT

  Return the key/data pair to which the cursor refers.

  The DBcursor->get() method will return DB_KEYEMPTY if DB_CURRENT is set and the cursor key/data pair was deleted.

- DB_FIRST

  The cursor is set to refer to the first key/data pair of the database, and that pair is returned. If the first key has duplicate values, the first data item in the set of duplicates is returned.

  If the database is a Queue or Recno database, DBcursor->get() using the DB_FIRST flag will ignore any keys that exist but were never explicitly created by the application, or were created and later deleted.

  The DBcursor->get() method will return DB_NOTFOUND if DB_FIRST is set and the database is empty.

- DB_GET_BOTH

  Move the cursor to the specified key/data pair of the database. The cursor is positioned to a key/data pair if both the key and data match the values provided on the key and data parameters.

  In all other ways, this flag is identical to the DB_SET flag.

  When used with DBcursor->pget() on a secondary index handle, both the secondary and primary keys must be matched by the secondary and primary key item in the database. It is an error to use the DB_GET_BOTH flag with the DBcursor->get() version of this method and a cursor that has been opened on a secondary index handle.

- DB_GET_BOTH_RANGE

  Move the cursor to the specified key/data pair of the database. The key parameter must be an exact match with a key in the database. The data item retrieved is the item in a duplicate set that is the smallest value which is greater than or equal to the value provided by the data parameter (as determined by the comparison function). If this flag is specified on a database configured without sorted duplicate support, the behavior is identical to the DB_GET_BOTH flag. Returns the datum associated with the given key/data pair.

  In all other ways, this flag is identical to the DB_GET_BOTH flag.

- DB_GET_RECNO

  Return the record number associated with the cursor. The record number will be returned in **data**, as described in DBT. The **key** parameter is ignored.

  For DB_GET_RECNO to be specified, the underlying database must be of type Btree, and it must have been created with the DB_RECNUM flag.

When called on a cursor opened on a database that has been made into a secondary index, the `DBcursor->get()` and `DBcursor->pget()` methods return the record number of the primary database in **data**. In addition, the `DBcursor->pget()` method returns the record number of the secondary index in **pkey**. If either underlying database is not of type Btree or is not created with the DB_RECNUM flag, the out-of-band record number of 0 is returned.

- `DB_JOIN_ITEM`

  Do not use the data value found in all of the cursors as a lookup key for the primary database, but simply return it in the key parameter instead. The data parameter is left unchanged.

  For DB_JOIN_ITEM to be specified, the underlying cursor must have been returned from the DB->join() (page 61) method.

- `DB_LAST`

  The cursor is set to refer to the last key/data pair of the database, and that pair is returned. If the last key has duplicate values, the last data item in the set of duplicates is returned.

  If the database is a Queue or Recno database, `DBcursor->get()` using the DB_LAST flag will ignore any keys that exist but were never explicitly created by the application, or were created and later deleted.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_LAST is set and the database is empty.

- `DB_NEXT`

  If the cursor is not yet initialized, DB_NEXT is identical to DB_FIRST. Otherwise, the cursor is moved to the next key/data pair of the database, and that pair is returned. In the presence of duplicate key values, the value of the key may not change.

  If the database is a Queue or Recno database, `DBcursor->get()` using the DB_NEXT flag will skip any keys that exist but were never explicitly created by the application, or those that were created and later deleted.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_NEXT is set and the cursor is already on the last record in the database.

- `DB_NEXT_DUP`

  If the next key/data pair of the database is a duplicate data record for the current key/data pair, the cursor is moved to the next key/data pair of the database, and that pair is returned.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_NEXT_DUP is set and the next key/data pair of the database is not a duplicate data record for the current key/data pair.

- `DB_NEXT_NODUP`

  If the cursor is not yet initialized, DB_NEXT_NODUP is identical to DB_FIRST. Otherwise, the cursor is moved to the next non-duplicate key of the database, and that key/data pair is returned.

  If the database is a Queue or Recno database, `DBcursor->get()` using the DB_NEXT_NODUP flag will ignore any keys that exist but were never explicitly created by the application, or those that were created and later deleted.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_NEXT_NODUP is set and no non-duplicate key/data pairs exist after the cursor position in the database.

- `DB_PREV`

  If the cursor is not yet initialized, DB_PREV is identical to DB_LAST. Otherwise, the cursor is moved to the previous key/data pair of the database, and that pair is returned. In the presence of duplicate key values, the value of the key may not change.

  If the database is a Queue or Recno database, `DBcursor->get()` using the DB_PREV flag will skip any keys that exist but were never explicitly created by the application, or those that were created and later deleted.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_PREV is set and the cursor is already on the first record in the database.

- `DB_PREV_DUP`

  If the previous key/data pair of the database is a duplicate data record for the current key/data pair, the cursor is moved to the previous key/data pair of the database, and that pair is returned.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_PREV_DUP is set and the previous key/data pair of the database is not a duplicate data record for the current key/data pair.

- `DB_PREV_NODUP`

  If the cursor is not yet initialized, DB_PREV_NODUP is identical to DB_LAST. Otherwise, the cursor is moved to the previous non-duplicate key of the database, and that key/data pair is returned.

  If the database is a Queue or Recno database, `DBcursor->get()` using the DB_PREV_NODUP flag will ignore any keys that exist but were never explicitly created by the application, or those that were created and later deleted.

  The `DBcursor->get()` method will return DB_NOTFOUND if DB_PREV_NODUP is set and no non-duplicate key/data pairs exist before the cursor position in the database.

- `DB_SET`

Move the cursor to the specified key/data pair of the database, and return the datum associated with the given key.

The `DBcursor->get()` method will return DB_NOTFOUND if DB_SET is set and no matching keys are found. The `DBcursor->get()` method will return DB_KEYEMPTY if DB_SET is set and the database is a Queue or Recno database, and the specified key exists, but was never explicitly created by the application or was later deleted. In the presence of duplicate key values, `DBcursor->get()` will return the first data item for the given key.

- `DB_SET_RANGE`

  Move the cursor to the specified key/data pair of the database. In the case of the Btree access method, the key is returned as well as the data item and the returned key/data pair is the smallest key greater than or equal to the specified key (as determined by the Btree comparison function), permitting partial key matches and range searches.

  In all other ways the behavior of this flag is the same as the DB_SET flag.

- `DB_SET_RECNO`

  Move the cursor to the specific numbered record of the database, and return the associated key/data pair. The **data** field of the specified **key** must be a pointer to a memory location from which a `db_recno_t` may be read, as described in DBT. This memory location will be read to determine the record to be retrieved.

  For DB_SET_RECNO to be specified, the underlying database must be of type Btree, and it must have been created with the DB_RECNUM flag.

In addition, the following flags may be set by bitwise inclusively **OR**'ing them into the **flags** parameter:

- `DB_IGNORE_LEASE`

  This flag is relevant only when using a replicated environment.

  Return the data item irrespective of the state of master leases. The item will be returned under all conditions: if master leases are not configured, if the request is made to a client, if the request is made to a master with a valid lease, or if the request is made to a master without a valid lease.

- `DB_READ_COMMITTED`

  Configure a transactional get operation to have degree 2 isolation (the read is not repeatable).

- `DB_READ_UNCOMMITTED`

  Database items read during a transactional call will have degree 1 isolation, including modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

- DB_MULTIPLE

  Return multiple data items in the **data** parameter.

  In the case of Btree or Hash databases, duplicate data items for the current key, starting at the current cursor position, are entered into the buffer. Subsequent calls with both the DB_NEXT_DUP and DB_MULTIPLE flags specified will return additional duplicate data items associated with the current key or DB_NOTFOUND if there are no additional duplicate data items to return. Subsequent calls with both the DB_NEXT and DB_MULTIPLE flags specified will return additional duplicate data items associated with the current key or if there are no additional duplicate data items will return the next key and its data items or DB_NOTFOUND if there are no additional keys in the database.

  In the case of Queue or Recno databases, data items starting at the current cursor position are entered into the buffer. The record number of the first record will be returned in the **key** parameter. The record number of each subsequent returned record must be calculated from this value. Subsequent calls with the DB_MULTIPLE flag specified will return additional data items or DB_NOTFOUND if there are no additional data items to return.

  The buffer to which the **data** parameter refers must be provided from user memory (see DB_DBT_USERMEM). The buffer must be at least as large as the page size of the underlying database, aligned for unsigned integer access, and be a multiple of 1024 bytes in size. If the buffer size is insufficient, then upon return from the call the size field of the **data** parameter will have been set to an estimated buffer size, and the error DB_BUFFER_SMALL is returned. (The size is an estimate as the exact size needed may not be known until all entries are read. It is best to initially provide a relatively large buffer, but applications should be prepared to resize the buffer as necessary and repeatedly call the method.)

  The multiple data items can be iterated over using the DB_MULTIPLE_NEXT (page 178) macro.

  The DB_MULTIPLE flag may only be used with the DB_CURRENT, DB_FIRST, DB_GET_BOTH, DB_GET_BOTH_RANGE, DB_NEXT, DB_NEXT_DUP, DB_NEXT_NODUP, DB_SET, DB_SET_RANGE, and DB_SET_RECNO options. The DB_MULTIPLE flag may not be used when accessing databases made into secondary indices using the DB->associate() (page 6) method.

- DB_MULTIPLE_KEY

  Return multiple key and data pairs in the **data** parameter.

  Key and data pairs, starting at the current cursor position, are entered into the buffer. Subsequent calls with both the DB_NEXT and DB_MULTIPLE_KEY flags specified will return additional key and data pairs or DB_NOTFOUND if there are no additional key and data items to return.

  In the case of Btree or Hash databases, the multiple key and data pairs can be iterated over using the DB_MULTIPLE_KEY_NEXT (page 179) macro.

  In the case of Queue or Recno databases, the multiple record number and data pairs can be iterated over using the DB_MULTIPLE_RECNO_NEXT (page 181) macro.

The buffer to which the **data** parameter refers must be provided from user memory (see DB_DBT_USERMEM). The buffer must be at least as large as the page size of the underlying database, aligned for unsigned integer access, and be a multiple of 1024 bytes in size. If the buffer size is insufficient, then upon return from the call the size field of the **data** parameter will have been set to an estimated buffer size, and the error DB_BUFFER_SMALL is returned. (The size is an estimate as the exact size needed may not be known until all entries are read. It is best to initially provide a relatively large buffer, but applications should be prepared to resize the buffer as necessary and repeatedly call the method.)

The DB_MULTIPLE_KEY flag may only be used with the DB_CURRENT, DB_FIRST, DB_GET_BOTH, DB_GET_BOTH_RANGE, DB_NEXT, DB_NEXT_DUP, DB_NEXT_NODUP, DB_SET, DB_SET_RANGE, and DB_SET_RECNO options. The DB_MULTIPLE_KEY flag may not be used when accessing databases made into secondary indices using the DB->associate() (page 6) method.

- `DB_RMW`

  Acquire write locks instead of read locks when doing the read, if locking is configured. Setting this flag can eliminate deadlock during a read-modify-write cycle by acquiring the write lock during the read part of the cycle so that another thread of control acquiring a read lock for the same item, in its own read-modify-write cycle, will not result in deadlock.

Errors

The `DBcursor->get()` method may fail and return one of the following non-zero errors:

DB_BUFFER_SMALL

The requested item could not be returned due to undersized buffer.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LEASE_EXPIRED

The operation failed because the site's replication master lease has expired.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DB_SECONDARY_BAD

A secondary index references a nonexistent primary key.

EINVAL

If the DB_CURRENT, DB_NEXT_DUP or DB_PREV_DUP flags were specified and the cursor has not been initialized; the `DBcursor->pget()` method was called with a cursor that does not refer to a secondary index; or if an invalid flag value or parameter was specified.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->get_priority()

```
#include <db.h>

int
DBcursor->get_priority(DBC *DbCursor, DB_CACHE_PRIORITY *priorityp);
```

The `DBcursor->get_priority()` method returns the cache priority for pages referenced by the DBcursor handle.

The `DBcursor->get_priority()` method may be called at any time during the life of the application.

The `DBcursor->get_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priorityp

The `DBcursor->get_priority()` method returns a reference to the cache priority for pages referenced by the DBcursor handle in **priorityp**.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->put()

```
#include <db.h>

int
DBcursor->put(DBC *DBcursor, DBT *key, DBT *data, u_int32_t flags);
```

The `DBcursor->put()` method stores key/data pairs into the database.

Unless otherwise specified, the `DBcursor->put()` method returns a non-zero error value on failure and 0 on success.

If `DBcursor->put()` fails for any reason, the state of the cursor will be unchanged. If `DBcursor->put()` succeeds and an item is inserted into the database, the cursor is always positioned to refer to the newly inserted item.

Parameters

key

The key DBT operated on.

data

The data DBT operated on.

flags

The **flags** parameter must be set to one of the following values:

- `DB_AFTER`

  In the case of the Btree and Hash access methods, insert the data element as a duplicate element of the key to which the cursor refers. The new element appears immediately after the current cursor position. It is an error to specify DB_AFTER if the underlying Btree or Hash database is not configured for unsorted duplicate data items. The **key** parameter is ignored.

  In the case of the Recno access method, it is an error to specify DB_AFTER if the underlying Recno database was not created with the DB_RENUMBER flag. If the DB_RENUMBER flag was specified, a new key is created, all records after the inserted item are automatically renumbered, and the key of the new record is returned in the structure to which the **key** parameter refers. The initial value of the **key** parameter is ignored. See DB->open() (page 66) for more information.

  The DB_AFTER flag may not be specified to the Queue access method.

  The `DBcursor->put()` method will return DB_NOTFOUND if the current cursor record has already been deleted and the underlying access method is Hash.

- `DB_BEFORE`

In the case of the Btree and Hash access methods, insert the data element as a duplicate element of the key to which the cursor refers. The new element appears immediately before the current cursor position. It is an error to specify DB_AFTER if the underlying Btree or Hash database is not configured for unsorted duplicate data items. The **key** parameter is ignored.

In the case of the Recno access method, it is an error to specify DB_BEFORE if the underlying Recno database was not created with the DB_RENUMBER flag. If the DB_RENUMBER flag was specified, a new key is created, the current record and all records after it are automatically renumbered, and the key of the new record is returned in the structure to which the **key** parameter refers. The initial value of the **key** parameter is ignored. See DB->open() (page 66) for more information.

The DB_BEFORE flag may not be specified to the Queue access method.

The `DBcursor->put()` method will return DB_NOTFOUND if the current cursor record has already been deleted and the underlying access method is Hash.

- `DB_CURRENT`

  Overwrite the data of the key/data pair to which the cursor refers with the specified data item. The **key** parameter is ignored.

  The `DBcursor->put()` method will return DB_NOTFOUND if the current cursor record has already been deleted.

- `DB_KEYFIRST`

  Insert the specified key/data pair into the database.

  If the underlying database supports duplicate data items, and if the key already exists in the database and a duplicate sort function has been specified, the inserted data item is added in its sorted location. If the key already exists in the database and no duplicate sort function has been specified, the inserted data item is added as the first of the data items for that key.

- `DB_KEYLAST`

  Insert the specified key/data pair into the database.

  If the underlying database supports duplicate data items, and if the key already exists in the database and a duplicate sort function has been specified, the inserted data item is added in its sorted location. If the key already exists in the database, and no duplicate sort function has been specified, the inserted data item is added as the last of the data items for that key.

- `DB_NODUPDATA`

  In the case of the Btree and Hash access methods, insert the specified key/data pair into the database, unless a key/data pair comparing equally to it already exists in the database.

If a matching key/data pair already exists in the database, DB_KEYEXIST (page 169) is returned. The DB_NODUPDATA flag may only be specified if the underlying database has been configured to support sorted duplicate data items.

The DB_NODUPDATA flag may not be specified to the Queue or Recno access methods.

Errors

The `DBcursor->put()` method may fail and return one of the following non-zero errors:

DB_KEYEXIST

An attempt was made to insert a duplicate key into a database not configured for duplicate data.

DB_FOREIGN_CONFLICT

A foreign key constraint violation has occurred. This can be caused by one of two things:

1. An attempt was made to add a record to a constrained database, and the key used for that record does not exist in the foreign key database.
2. DB_FOREIGN_ABORT (page 11) was declared for a foreign key database, and then subsequently a record was deleted from the foreign key database without first removing it from the constrained secondary database.

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_REP_HANDLE_DEAD

When a client synchronizes with the master, it is possible for committed transactions to be rolled back. This invalidates all the database and cursor handles opened in the replication environment. Once this occurs, an attempt to use such a handle will return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

EACCES

An attempt was made to modify a read-only database.

EINVAL

If the DB_AFTER, DB_BEFORE or DB_CURRENT flags were specified and the cursor has not been initialized; the DB_AFTER or DB_BEFORE flags were specified and a duplicate sort function has been specified; the DB_CURRENT flag was specified, a duplicate sort function has been specified, and the data item of the referenced key/data pair does not compare equally to the **data** parameter; the DB_AFTER or DB_BEFORE flags were specified, and the underlying access method is Queue; an attempt was made to add a record to a fixed-length database that was too large to fit; an attempt was made to add a record to a secondary index; or if an invalid flag value or parameter was specified.

EPERM

Write attempted on read-only cursor when the DB_INIT_CDB flag was specified to DB_ENV->open() (page 231).

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

DBcursor->set_priority()

```
#include <db.h>

int
DBcursor->set_priority(DBC *DbCursor, DB_CACHE_PRIORITY priority);
```

Set the cache priority for pages referenced by the DBcursor handle.

The priority of a page biases the replacement algorithm to be more or less likely to discard a page when space is needed in the buffer pool. The bias is temporary, and pages will eventually be discarded if they are not referenced again. The `DBcursor->set_priority()` method is only advisory, and does not guarantee pages will be treated in a specific way.

The `DBcursor->set_priority()` method may be called at any time during the life of the application.

The `DBcursor->set_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priority

The **priority** parameter must be set to one of the following values:

- `DB_PRIORITY_VERY_LOW`

  The lowest priority: pages are the most likely to be discarded.

- `DB_PRIORITY_LOW`

  The next lowest priority.

- `DB_PRIORITY_DEFAULT`

  The default priority.

- `DB_PRIORITY_HIGH`

  The next highest priority.

- `DB_PRIORITY_VERY_HIGH`

  The highest priority: pages are the least likely to be discarded.

Class

DBcursor

See Also

Database Cursors and Related Methods (page 148)

Chapter 4. The DBT Handle

```
#include <db.h>

    typedef struct {
        void *data;
        u_int32_t size;
        u_int32_t ulen;
        u_int32_t dlen;
        u_int32_t doff;
        u_int32_t flags;
    } DBT;
```

Storage and retrieval for the DB access methods are based on key/data pairs. Both key and data items are represented by the DBT data structure. (The name DBT is a mnemonic for data base thang, and was used because no one could think of a reasonable name that wasn't already in use somewhere else.) Key and data byte strings may refer to strings of zero length up to strings of essentially unlimited length. See Database limits for more information.

All fields of the DBT structure that are not explicitly set should be initialized to nul bytes before the first time the structure is used. Do this by declaring the structure external or static, or by calling the C library routine **memset**(3).

By default, the **flags** structure element is expected to be set to 0. In this default case, when the application is providing Berkeley DB a key or data item to store into the database, Berkeley DB expects the **data** structure element to point to a byte string of **size** bytes. When returning a key/data item to the application, Berkeley DB will store into the **data** structure element a pointer to a byte string of **size** bytes, and the memory to which the pointer refers will be allocated and managed by Berkeley DB. Note that using the default flags for returned `DBT`s is only compatible with single threaded usage of Berkeley DB.

The elements of the DBT structure are defined as follows:

- **void *data;**

  A pointer to a byte string.

- **u_int32_t size;**

  The length of **data**, in bytes.

- **u_int32_t ulen;**

  The size of the user's buffer (to which **data** refers), in bytes. This location is not written by the Berkeley DB functions.

  Set the byte size of the user-specified buffer.

Note that applications can determine the length of a record by setting the `ulen` field to 0 and checking the return value in the **size** field. See the `DB_DBT_USERMEM` flag for more information.

- **u_int32_t dlen;**

  The length of the partial record being read or written by the application, in bytes. See the `DB_DBT_PARTIAL` flag for more information.

- **u_int32_t doff;**

  The offset of the partial record being read or written by the application, in bytes. See the `DB_DBT_PARTIAL` flag for more information.

- **u_int32_t flags;**

  The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

  - `DB_DBT_MALLOC`

    When this flag is set, Berkeley DB will allocate memory for the returned key or data item (using **malloc**(3), or the user-specified malloc function), and return a pointer to it in the **data** field of the key or data `DBT` structure. Because any allocated memory becomes the responsibility of the calling application, the caller must determine whether memory was allocated using the returned value of the **data** field.

    It is an error to specify more than one of `DB_DBT_MALLOC`, `DB_DBT_REALLOC`, and `DB_DBT_USERMEM`.

  - `DB_DBT_REALLOC`

    When this flag is set Berkeley DB will allocate memory for the returned key or data item (using **realloc**(3), or the user-specified realloc function), and return a pointer to it in the **data** field of the key or data DBT structure. Because any allocated memory becomes the responsibility of the calling application, the caller must determine whether memory was allocated using the returned value of the **data** field.

    The difference between `DB_DBT_MALLOC` and `DB_DBT_REALLOC` is that the latter will call **realloc**(3) instead of **malloc**(3), so the allocated memory will be grown as necessary instead of the application doing repeated free/malloc calls.

    It is an error to specify more than one of `DB_DBT_MALLOC`, `DB_DBT_REALLOC`, and `DB_DBT_USERMEM`.

  - `DB_DBT_USERMEM`

    The *data* field of the key or data structure must refer to memory that is at least *ulen* bytes in length. If the length of the requested item is less than or equal to that number of bytes, the item is copied into the memory to which the *data* field refers.

Otherwise, the *size* field is set to the length needed for the requested item, and the error DB_BUFFER_SMALL is returned.

It is an error to specify more than one of DB_DBT_MALLOC, DB_DBT_REALLOC, and DB_DBT_USERMEM.

- DB_DBT_PARTIAL

  Do partial retrieval or storage of an item. If the calling application is doing a get, the **dlen** bytes starting **doff** bytes from the beginning of the retrieved data record are returned as if they comprised the entire record. If any or all of the specified bytes do not exist in the record, the get is successful, and any existing bytes are returned.

  For example, if the data portion of a retrieved record was 100 bytes, and a partial retrieval was done using a DBT having a **dlen** field of 20 and a **doff** field of 85, the get call would succeed, the **data** field would refer to the last 15 bytes of the record, and the **size** field would be set to 15.

  If the calling application is doing a put, the **dlen** bytes starting **doff** bytes from the beginning of the specified key's data record are replaced by the data specified by the **data** and **size** structure elements. If **dlen** is smaller than **size** the record will grow; if **dlen** is larger than **size** the record will shrink. If the specified bytes do not exist, the record will be extended using nul bytes as necessary, and the put call will succeed.

  It is an error to attempt a partial put using the DB->put() (page 71) method in a database that supports duplicate records. Partial puts in databases supporting duplicate records must be done using a DBcursor->put() (page 167) method.

  It is an error to attempt a partial put with differing **dlen** and **size** values in Queue or Recno databases with fixed-length records.

  For example, if the data portion of a retrieved record was 100 bytes, and a partial put was done using a DBT having a **dlen** field of 20, a **doff** field of 85, and a **size** field of 30, the resulting record would be 115 bytes in length, where the last 30 bytes would be those specified by the put call.

- DB_DBT_APPMALLOC

  After an application-supplied callback routine passed to either DB->associate() (page 6) or DB->set_append_recno() (page 81) is executed, the **data** field of a DBT may refer to memory allocated with **malloc**(3) or **realloc**(3). In that case, the callback sets the DB_DBT_APPMALLOC flag in the DBT so that Berkeley DB will call **free**(3) to deallocate the memory when it is no longer required.

- DB_DBT_MULTIPLE

  Set in a secondary key creation callback routine passed to DB->associate() (page 6) to indicate that multiple secondary keys should be associated with the given primary key/data pair. If set, the **size** field indicates the number of secondary keys and the **data** field refers to an array of that number of DBT structures.

The DB_DBT_APPMALLOC flag may be set on any of the DBT structures to indicate that their **data** field needs to be freed.

DBT and Bulk Operations

DBT and Bulk Operations	Description
DB_MULTIPLE_INIT	Initialize bulk get retrieval
DB_MULTIPLE_NEXT	Next bulk get retrieval
DB_MULTIPLE_KEY_NEXT	Next bulk get retrieval
DB_MULTIPLE_RECNO_NEXT	Next bulk get retrieval
DB_MULTIPLE_WRITE_INIT	Initialize a bulk buffer to hold key/data pairs
DB_MULTIPLE_WRITE_NEXT	Append a data item to a bulk buffer
DB_MULTIPLE_RESERVE_NEXT	Reserve space for a data item in a bulk buffer
DB_MULTIPLE_KEY_WRITE_NEXT	Append a key / data pair to a bulk buffer
DB_MULTIPLE_KEY_RESERVE_NEXT	Reserve space for a key / data pair in a bulk buffer
DB_MULTIPLE_RECNO_WRITE_INIT	Initialize a bulk buffer to hold recno/data pairs
DB_MULTIPLE_RECNO_WRITE_NEXT	Append a record number / data pair to a bulk buffer
DB_MULTIPLE_RECNO_RESERVE_NEXT	Reserve space for a record number / data pair in a bulk buffer

DB_MULTIPLE_INIT

```
#include <db.h>

DB_MULTIPLE_INIT(void *pointer, DBT *data);
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the DB->get() (page 30) or DBcursor->get() (page 158) methods, the data DBT returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through the DB_MULTIPLE_* macros.

This macro initializes a variable used for bulk retrieval.

Parameters

pointer

The **pointer** parameter is a variable to be initialized.

data

The **data** parameter is a DBT structure returned from a successful call to DB->get() (page 30) or DBcursor->get() (page 158) for which one of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_NEXT

```
#include <db.h>

DB_MULTIPLE_NEXT(void *pointer, DBT *data, void *retdata,
    size_t retdlen);
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the DB->get() (page 30) or DBcursor->get() (page 158) methods, the data DBT returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through the DB_MULTIPLE_* macros.

Returns the next DBT in the bulk retrieval set.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

This parameter is set to NULL if there are no more key/data pairs in the returned set.

data

The **data** parameter is a DBT structure returned from a successful call to DB->get() (page 30) or DBcursor->get() (page 158) for which he DB_MULTIPLE flag was specified.

The **data** parameter must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

retdata

The **retdata** is set to the next data element in the returned set.

retdlen

The **retdlen** parameter is set to the length, in bytes, of that data element. When used with the Queue and Recno access methods, **retdata** parameter will be set to NULL for deleted records.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_KEY_NEXT

```
#include <db.h>

DB_MULTIPLE_KEY_NEXT(void *pointer, DBT *data,
    void *retkey, size_t retklen, void *retdata, size_t retdlen);
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the DB->get() (page 30) or DBcursor->get() (page 158) methods, the data DBT returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through the DB_MULTIPLE_* macros.

Returns the next DBT in the bulk retrieval set. Use this macro with DBT structures obtained from a database that uses the Btree or Hash access methods.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

This parameter is set to NULL if there are no more key/data pairs in the returned set.

data

The **data** parameter is a DBT structure returned from a successful call to DBcursor->get() (page 158) with the Btree or Hash access methods for which the DB_MULTIPLE_KEY flag was specified.

The **data** parameter must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

retkey

The **retkey** parameter is set to the next key element in the returned set.

retklen

The **retklen** parameter is set to the length, in bytes, of the next key element.

retdata

The **retdata** parameter is set to the next data element in the returned set.

retdlen

The **retdlen** parameter is set to the length, in bytes, of the next data element.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_RECNO_NEXT

```
#include <db.h>

DB_MULTIPLE_RECNO_NEXT(void *pointer, DBT *data,
    db_recno_t recno, void * retdata, size_t retdlen);
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the DB->get() (page 30) or DBcursor->get() (page 158) methods, the data DBT returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through the DB_MULTIPLE_* macros.

Returns the next DBT in the bulk retrieval set. Use this macro with DBT structures obtained from a database that uses the Queue or Recno access methods.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

This parameter is set to NULL if there are no more key/data pairs in the returned set.

data

The **data** parameter is a DBT structure returned from a successful call to DBcursor->get() (page 158) with the Queue or Recno access methods for which the DB_MULTIPLE_KEY flag was specified.

The **data** parameter must have been initialized by a call to DB_MULTIPLE_INIT (page 177).

recno

The **recno** parameter is set to the record number of the next record in the returned set.

retdata

The **retdata** parameter is set to the next data element in the returned set. Deleted records are not included in the results.

retdlen

The **retdlen** parameter is set to the length, in bytes, of the next data element.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_WRITE_INIT

```
#include <db.h>

DB_MULTIPLE_WRITE_INIT(void *pointer, DBT *data);
```

Initialize a DBT containing a bulk buffer for use with the DB_MULTIPLE or DB_MULTIPLE_KEY flags to the DB->put() (page 71) or DB->del() (page 22) methods.

This macro initializes an opaque pointer variable used for adding records to a bulk buffer. Use this macro for buffers that will contain either a data item per record (for use with DB_MULTIPLE), or key/data pairs, where the key is not a record number. For record number keys, use DB_MULTIPLE_RECNO_WRITE_INIT.

Parameters

pointer

The **pointer** parameter is an opaque pointer variable to be initialized.

data

The **data** parameter is a DBT structure that has been initialized by the application with a buffer to hold multiple records. The **ulen** field must be set to the size of the buffer allocated by the application, and must be a multiple of 4.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_WRITE_NEXT

```
#include <db.h>

DB_MULTIPLE_WRITE_NEXT(void *pointer, DBT *dbt, void *data,
    size_t dlen);
```

Appends a data item to the bulk buffer.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_WRITE_INIT (page 182).

This parameter is set to NULL if the data item does not fit in the buffer.

dbt

The **dbt** parameter is a DBT structure initialized with DB_MULTIPLE_WRITE_INIT (page 182).

data

A pointer to the bytes to be copied into the bulk buffer.

dlen

The number of bytes to be copied.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_RESERVE_NEXT

```
#include <db.h>

DB_MULTIPLE_RESERVE_NEXT(void *pointer, DBT *dbt,
    void *ddest, size_t dlen);
```

Reserves space for a data item in a bulk buffer.

Parameters

dbt

The **dbt** parameter is a DBT structure initialized with DB_MULTIPLE_WRITE_INIT (page 182).

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_WRITE_INIT (page 182).

ddest

The **ddest** parameter is set to the location reserved in the bulk buffer for the data item.

This parameter is set to NULL if the data item does not fit in the buffer.

dlen

The number of bytes to be reserved for the data item.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_KEY_WRITE_NEXT

```
#include <db.h>

DB_MULTIPLE_KEY_WRITE_NEXT(void *pointer, DBT *dbt,
    void *key, size_t klen, void *data, size_t dlen);
```

Appends a key / data pair to the bulk buffer.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_WRITE_INIT (page 182).

This parameter is set to NULL if the data item does not fit in the buffer.

dbt

The **dbt** parameter is a DBT structure initialized with DB_MULTIPLE_WRITE_INIT (page 182).

key

A pointer to the bytes for the key to be copied into the bulk buffer.

klen

The number of bytes to be copied for the key.

data

A pointer to the bytes for the data item to be copied into the bulk buffer.

dlen

The number of bytes to be copied for the data item.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_KEY_RESERVE_NEXT

```
#include <db.h>

DB_MULTIPLE_KEY_RESERVE_NEXT(void *pointer, DBT *dbt,
    void *kdest, size_t klen, void *ddest, size_t dlen);
```

Reserves space for a key / data pair in a bulk buffer.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_WRITE_INIT (page 182).

kdest

The **kdest** parameter is set to the location reserved in the bulk buffer for the key.

This parameter is set to NULL if the data item does not fit in the buffer.

klen

The number of bytes to be reserved for the key.

ddest

The **ddest** parameter is set to the location reserved in the bulk buffer for the data item.

This parameter is set to NULL if the data item does not fit in the buffer.

dlen

The number of bytes to be reserved for the data item.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_RECNO_WRITE_INIT

```
#include <db.h>

DB_MULTIPLE_RECNO_WRITE_INIT(void *pointer, DBT *data);
```

Initialize a DBT containing a bulk buffer for use with the DB_MULTIPLE or DB_MULTIPLE_KEY flags to the DB->put() (page 71) or DB->del() (page 22) methods, if the buffer will contain record number keys.

This macro initializes an opaque pointer variable used for adding records to a bulk buffer. Use this macro for buffers that will contain either a list of record numbers (for use with DB_MULTIPLE), or key/data pairs, where the key is a record number.

Parameters

pointer

The **pointer** parameter is an opaque pointer variable to be initialized.

data

The **data** parameter is a DBT structure that has been initialized by the application with a buffer to hold multiple records. The **ulen** field must be set to the size of the buffer allocated by the application, which must be a multiple of 4.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_RECNO_WRITE_NEXT

```
#include <db.h>

DB_MULTIPLE_RECNO_WRITE_NEXT(void *pointer, DBT *dbt,
    db_recno_t recno, void *data, size_t dlen);
```

Appends a record number / data pair to the bulk buffer.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_RECNO_WRITE_INIT (page 187).

This parameter is set to NULL if the data item does not fit in the buffer.

dbt

The **dbt** parameter is a DBT structure initialized with

```
#include <db.h>

DB_MULTIPLE_WRITE_INIT(void *pointer, DBT *data);
```

Initialize a DBT containing a bulk buffer for use with the DB_MULTIPLE or DB_MULTIPLE_KEY flags to the DB->put() (page 71) or DB->del() (page 22) methods.

This macro initializes an opaque pointer variable used for adding records to a bulk buffer. Use this macro for buffers that will contain either a data item per record (for use with DB_MULTIPLE), or key/data pairs, where the key is not a record number. For record number keys, use DB_MULTIPLE_RECNO_WRITE_INIT.

Parameters

pointer

The **pointer** parameter is an opaque pointer variable to be initialized.

data

The **data** parameter is a DBT structure that has been initialized by the application With a buffer to hold multiple DBT records. The ulen field must be set to the size of the buffer allocated by the application, and must be a multiple of 4.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

(page 182).

recno

The record number to be copied into the bulk buffer.

data

A pointer to the bytes to be copied into the bulk buffer.

dlen

The number of bytes to be copied.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

DB_MULTIPLE_RECNO_RESERVE_NEXT

```
#include <db.h>

DB_MULTIPLE_RECNO_RESERVE_NEXT(void *pointer, DBT *dbt, db_recno_t recno,
    void *ddest, size_t dlen);
```

Reserves space for a record number / data pair in a bulk buffer.

Parameters

pointer

The **pointer** parameter is a variable that must have been initialized by a call to DB_MULTIPLE_RECNO_WRITE_INIT (page 187).

dbt

The **dbt** parameter is a DBT structure initialized with DB_MULTIPLE_RECNO_WRITE_INIT (page 187).

recno

The record number to be copied into the bulk buffer.

This parameter is set to 0 if the data item does not fit in the buffer.

ddest

The **ddest** parameter is set to the location reserved in the bulk buffer for the data item.

This parameter is set to NULL if the data item does not fit in the buffer.

dlen

The number of bytes to be reserved.

Class

DBT

See Also

DBT and Bulk Operations (page 176)

Chapter 5. The DB_ENV Handle

The DB_ENV object is the handle for a Berkeley DB environment — a collection including support for some or all of caching, locking, logging and transaction subsystems, as well as databases and log files. Methods of the DB_ENV handle are used to configure the environment as well as to operate on subsystems and databases in the environment.

DB_ENV handles are created using the db_env_create (page 199) method, and are opened using the DB_ENV->open() (page 231) method.

When you are done using your environment, close it using the DB_ENV->close() (page 197) method. Before closing your environment, make sure all open database handles are closed first. See the DB->close() (page 13) method for more information.

Database Environments and Related Methods

Database Environment Operations	Description
DB->get_env()	Return the DB's underlying DB_ENV handle
DB_ENV->close()	Close an environment
db_env_create	Create an environment handle
DB_ENV->dbremove()	Remove a database
DB_ENV->dbrename()	Rename a database
DB_ENV->err()	Error message
DB_ENV->failchk()	Check for thread failure
DB_ENV->fileid_reset()	Reset database file IDs
db_full_version	Return full version information
DB_ENV->get_home()	Return environment's home directory
DB_ENV->get_open_flags()	Return flags with which the environment was opened
DB_ENV->log_verify()	Verify log files of an environment.
DB_ENV->lsn_reset()	Reset database file LSNs
DB_ENV->open()	Open an environment
DB_ENV->remove()	Remove an environment
DB_ENV->stat_print()	Environment statistics
db_strerror	Error strings
db_version	Return version information
Environment Configuration	
DB_ENV->add_data_dir()	add an environment data directory
DB_ENV->set_alloc()	Set local space allocation functions
DB_ENV->set_app_dispatch()	Configure application recovery callback
DB_ENV->set_data_dir(), DB_ENV->get_data_dirs()	Set/get the environment data directory
DB_ENV->set_create_dir(), DB_ENV->get_create_dir()	add an environment data directory
DB_ENV->set_encrypt(), DB_ENV->get_encrypt_flags()	Set/get the environment cryptographic key
DB_ENV->set_event_notify()	Set event notification callback
DB_ENV->set_errcall()	Set error message callbacks
DB_ENV->set_errfile(), DB_ENV->get_errfile()	Set/get error message FILE
DB_ENV->set_errpfx(), DB_ENV->get_errpfx()	Set/get error message prefix
DB_ENV->set_feedback()	Set feedback callback

Database Environment Operations	Description
DB_ENV->set_flags(), DB_ENV->get_flags()	Environment configuration
DB_ENV->set_intermediate_dir_mode(), DB_ENV->get_intermediate_dir_mode()	Set/get intermediate directory creation mode
DB_ENV->set_isalive()	Set thread is-alive callback
DB_ENV->set_msgcall()	Set informational message callback
DB_ENV->set_msgfile(), DB_ENV->get_msgfile()	Set/get informational message FILE
DB_ENV->set_shm_key(), DB_ENV->get_shm_key()	Set/get system memory shared segment ID
DB_ENV->set_thread_count(), DB_ENV->get_thread_count()	Set/get approximate thread count
DB_ENV->set_thread_id()	Set thread of control ID function
DB_ENV->set_thread_id_string()	Set thread of control ID format function
DB_ENV->set_timeout(), DB_ENV->get_timeout()	Set/get lock and transaction timeout
DB_ENV->set_tmp_dir(), DB_ENV->get_tmp_dir()	Set/get the environment temporary file directory
DB_ENV->set_verbose(), DB_ENV->get_verbose()	Set/get verbose messages
DB_ENV->set_cachesize(), DB_ENV->get_cachesize()	Set/get the environment cache size

DB->get_env()

```
#include <db.h>

DB_ENV *
DB->get_env(DB *db);
```

The `DB->get_env()` method returns the handle for the database environment underlying the database.

The `DB->get_env()` method may be called at any time during the life of the application.

Class

DB

See Also

Database and Related Methods (page 3)

DB_ENV->add_data_dir()

```
#include <db.h>

int
DB_ENV->add_data_dir(DB_ENV *dbenv, const char *dir);
```

Add the path of a directory to be used as the location of the access method database files. Paths specified to the DB->open() (page 66) function will be searched relative to this path. Paths set using this method are additive, and specifying more than one will result in each specified directory being searched for database files.

If no database directories are specified, database files must be named either by absolute paths or relative to the environment home directory. See Berkeley DB File Naming for more information.

The database environment's data directories may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "add_data_dir", one or more whitespace characters, and the directory name.

The `DB_ENV->add_data_dir()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->add_data_dir()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->add_data_dir()` must be consistent with the existing environment or corruption can occur.

The `DB_ENV->add_data_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** parameter is a directory to be used as a location for database files.

When using a Unicode build on Windows (the default), this argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB_ENV->add_data_dir()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->close()

```
#include <db.h>

int
DB_ENV->close(DB_ENV *dbenv, u_int32_t flags);
```

The `DB_ENV->close()` method closes the Berkeley DB environment, freeing any allocated resources and closing all database handles opened with this environment handle, as well as closing any underlying subsystems.

When you call the `DB_ENV->close()` method, all open DB handles and DBcursor handles are closed automatically by this function. And, when you close a database handle, all cursors opened with it are closed automatically.

In multiple threads of control, each thread of control opens a database environment and the database handles within it. When you close each database handle using the `DB_ENV->close()` method, by default, the database is not synchronized and is similar to calling the `DB->close(DB_NOSYNC)` method. This is to avoid unncessary database synchronization when there are multiple environment handles open. To ensure all open database handles are synchronized when you close the last environment handle, set the flag parameter value of the `DB_ENV->close()` method to DB_FORCESYNC. This is similar to calling the `DB->close(0)` method to close each database handle.

If a database close operation fails, the method returns a non-zero error value for the first instance of such an error, and continues to close the rest of the database and environment handles.

The DB_ENV handle should not be closed while any other handle that refers to it is not yet closed; for example, database environment handles must not be closed while transactions in the environment have not yet been committed or aborted. Specifically, this includes the DB_TXN, DB_LOGC and DB_MPOOLFILE handles.

Where the environment was initialized with the DB_INIT_LOCK flag, calling `DB_ENV->close()` does not release any locks still held by the closing process, providing functionality for long-lived locks. Processes that want to have all their locks released can do so by issuing the appropriate DB_ENV->lock_vec() (page 326) call.

Where the environment was initialized with the DB_INIT_MPOOL flag, calling `DB_ENV->close()` implies calls to DB_MPOOLFILE->close() (page 403) for any remaining open files in the memory pool that were returned to this process by calls to DB_MPOOLFILE->open() (page 407). It does not imply a call to DB_MPOOLFILE->sync() (page 411) for those files.

Where the environment was initialized with the DB_INIT_TXN flag, calling `DB_ENV->close()` aborts any unresolved transactions. Applications should not depend on this behavior for transactions involving Berkeley DB databases; all such transactions should be explicitly resolved. The problem with depending on this semantic is that aborting an unresolved transaction involving database operations requires a database handle. Because the database handles should have been closed before calling `DB_ENV->close()`, it will not be possible to abort the transaction, and recovery will have to be run on the Berkeley DB environment before further operations are done.

Where log cursors were created using the DB_ENV->log_cursor() (page 340) method, calling `DB_ENV->close()` does not imply closing those cursors.

In multithreaded applications, only a single thread may call the `DB_ENV->close()` method.

After `DB_ENV->close()` has been called, regardless of its return, the Berkeley DB environment handle may not be accessed again.

The `DB_ENV->close()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or be set to the following value:

- `DB_FORCESYNC`

  When closing each database handle internally, synchronize the database. If this flag is not specified, the database handle is closed without synchronizing the database.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

db_env_create

```
#include <db.h>

int
db_env_create(DB_ENV **dbenvp, u_int32_t flags);
```

The `db_env_create()` function creates a `DB_ENV` structure that is the handle for a Berkeley DB environment. This function allocates memory for the structure, returning a pointer to the structure in the memory to which **dbenvp** refers. To release the allocated memory and discard the handle, call the DB_ENV->close() (page 197) or DB_ENV->remove() (page 237) methods.

`DB_ENV` handles are free-threaded if the DB_THREAD flag is specified to the DB_ENV->open() (page 231) method when the environment is opened. The `DB_ENV` handle should not be closed while any other handle remains open that is using it as a reference (for example, DB or DB_TXN). Once either the DB_ENV->close() (page 197) or DB_ENV->remove() (page 237) methods are called, the handle may not be accessed again, regardless of the method's return.

Before the handle may be used, you must open it using the DB_ENV->open() (page 231) method.

The DB_ENV handle contains a special field, `app_private`, which is declared as type `void *`. This field is provided for the use of the application program. It is initialized to NULL and is not further used by Berkeley DB in any way.

The `db_env_create()` method returns a non-zero error value on failure and 0 on success.

The **flags** parameter must be set to 0.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->dbremove()

```
#include <db.h>

int
DB_ENV->dbremove(DB_ENV *dbenv, DB_TXN *txnid,
    const char *file, const char *database, u_int32_t flags);
```

The `DB_ENV->dbremove()` method removes the database specified by the **file** and **database** parameters. If no **database** is specified, the underlying file represented by **file** is removed, incidentally removing all of the databases it contained.

Applications should never remove databases with open DB handles, or in the case of removing a file, when any database in the file has an open handle.

The `DB_ENV->dbremove()` method returns a non-zero error value on failure and 0 on success.

`DB_ENV->dbremove()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the `set_data_dir` string in the environment's DB_CONFIG file.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the `DB_AUTO_COMMIT` flag is specified to either this method or the environment handle, the operation will be implicitly transaction protected.

file

The **file** parameter is the physical file which contains the database(s) to be removed.

database

The **database** parameter is the database to be removed.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_AUTO_COMMIT`

  Enclose the `DB_ENV->dbremove()` call within a transaction. If the call succeeds, changes made by the operation will be recoverable. If the call fails, the operation will have made no changes.

Environment Variables

The environment variable `DB_HOME` may be used as the path of the database environment home.

Errors

The `DB_ENV->dbremove()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the method was called before DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

ENOENT

The file or directory does not exist.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->dbrename()

```
#include <db.h>

int
DB_ENV->dbrename(DB_ENV *dbenv, DB_TXN *txnid, const char *file,
    const char *database, const char *newname, u_int32_t flags);
```

The `DB_ENV->dbrename()` method renames the database specified by the **file** and **database** parameters to **newname**. If no **database** is specified, the underlying file represented by **file** is renamed using the value supplied to **newname**, incidentally renaming all of the databases it contained.

Applications should not rename databases that are currently in use. If an underlying file is being renamed and logging is currently enabled in the database environment, no database in the file may be open when the `DB_ENV->dbrename()` method is called.

The `DB_ENV->dbrename()` method returns a non-zero error value on failure and 0 on success.

`DB_ENV->dbrename()` is affected by any database directory specified using the DB_ENV->set_data_dir() (page 243) method, or by setting the `set_data_dir` string in the environment's DB_CONFIG file.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the `DB_AUTO_COMMIT` flag is specified to either this method or the environment handle, the operation will be implicitly transaction protected.

file

The **file** parameter is the physical file which contains the database(s) to be renamed.

When using a Unicode build on Windows (the default), the **file** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

database

The **database** parameter is the database to be renamed.

newname

The **newname** parameter is the new name of the database or file.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_AUTO_COMMIT`

  Enclose the `DB_ENV->dbrename()` call within a transaction. If the call succeeds, changes made by the operation will be recoverable. If the call fails, the operation will have made no changes.

Environment Variables

The environment variable `DB_HOME` may be used as the path of the database environment home.

Errors

The `DB_ENV->dbrename()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the method was called before DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

ENOENT

The file or directory does not exist.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->err()

```
#include <db.h>

void
DB_ENV->err(DB_ENV *dbenv, int error, const char *fmt, ...);

void
DB_ENV->errx(DB_ENV *dbenv, const char *fmt, ...);
```

The `DB_ENV->err()`, `DB_ENV->errx,()`, DB->err() (page 25) and `DB->errx()` methods provide error-messaging functionality for applications written using the Berkeley DB library.

The DB->err() (page 25) and DB_ENV->err() (page 204) methods constructs an error message consisting of the following elements:

- **An optional prefix string**

  If no error callback function has been set using the DB_ENV->set_errcall() (page 253) method, any prefix string specified using the DB_ENV->set_errpfx() (page 257) method, followed by two separating characters: a colon and a <space> character.

- **An optional printf-style message**

  The supplied message **fmt**, if non-NULL, in which the ANSI C X3.159-1989 (ANSI C) printf function specifies how subsequent parameters are converted for output.

- **A separator**

  Two separating characters: a colon and a <space> character.

- **A standard error string**

  The standard system or Berkeley DB library error string associated with the **error** value, as returned by the db_strerror (page 290) method.

This constructed error message is then handled as follows:

- If an error callback function has been set (see DB->set_errcall() (page 97) and DB_ENV->set_errcall() (page 253)), that function is called with two parameters: any prefix string specified (see DB->set_errpfx() (page 101) and DB_ENV->set_errpfx() (page 257)) and the error message.

- If a C library FILE * has been set (see DB->set_errfile() (page 99) and DB_ENV->set_errfile() (page 255)), the error message is written to that output stream.

- If none of these output options have been configured, the error message is written to stderr, the standard error output stream.

Parameters

error

The **error** parameter is the error value for which the `DB_ENV->err()` and DB->err() (page 25) methods will display a explanatory string.

fmt

The **fmt** parameter is an optional printf-style message to display.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->failchk()

```
#include <db.h>

int
DB_ENV->failchk(DB_ENV *dbenv, u_int32_t flags);
```

The `DB_ENV->failchk()` method checks for threads of control (either a true thread or a process) that have exited while manipulating Berkeley DB library data structures, while holding a logical database lock, or with an unresolved transaction (that is, a transaction that was never aborted or committed). For more information, see Architecting Data Store and Concurrent Data Store applications, and Architecting Transactional Data Store applications, both in the *Berkeley DB Programmer's Reference Guide*.

The `DB_ENV->failchk()` method is based on the `thread_id` and `is_alive` functions specified to the DB_ENV->set_thread_id() (page 278) and DB_ENV->set_isalive() (page 269) methods. Applications calling the `DB_ENV->failchk()` method must have already called the DB_ENV->set_isalive() (page 269) method, on the same DB_ENV, and must have configured their database environment using the DB_ENV->set_thread_count() (page 276) method.

If `DB_ENV->failchk()` determines a thread of control exited while holding database read locks, it will release those locks. If `DB_ENV->failchk()` determines a thread of control exited with an unresolved transaction, the transaction will be aborted. In either of these cases, `DB_ENV->failchk()` will return 0 and the application may continue to use the database environment.

In either of these cases, the `DB_ENV->failchk()` method will also report the process and thread IDs associated with any released locks or aborted transactions. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

If `DB_ENV->failchk()` determines a thread of control has exited such that database environment recovery is required, it will return DB_RUNRECOVERY. In this case, the application should not continue to use the database environment. For a further description as to the actions the application should take when this failure occurs, see Handling failure in Data Store and Concurrent Data Store applications, and Handling failure in Transactional Data Store applications, both in the *Berkeley DB Programmer's Reference Guide*.

In multiprocess applications, it is recommended that the DB_ENV handle used to invoke the `DB_ENV->failchk()` method not be shared and therefore not *free-threaded*.

The `DB_ENV->failchk()` method may not be called by the application before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->failchk()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_ENV->failchk()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->fileid_reset()

```
#include <db.h>

int
DB_ENV->fileid_reset(DB_ENV *dbenv, const char *file, u_int32_t flags);
```

The `DB_ENV->fileid_reset()` method allows database files to be copied, and then the copy used in the same database environment as the original.

All databases contain an ID string used to identify the database in the database environment cache. If a physical database file is copied, and used in the same environment as another file with the same ID strings, corruption can occur. The `DB_ENV->fileid_reset()` method creates new ID strings for all of the databases in the physical file.

The `DB_ENV->fileid_reset()` method modifies the physical file, in-place. Applications should not reset IDs in files that are currently in use.

The `DB_ENV->fileid_reset()` method may be called at any time during the life of the application.

The `DB_ENV->fileid_reset()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The name of the physical file in which new file IDs are to be created.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_ENCRYPT`

  The file contains encrypted databases.

Errors

The `DB_ENV->fileid_reset()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

db_full_version

```
#include <db.h>

char *
db_full_version(int *family, int *release, int *major, int *minor,
                int *patch);
```

The `db_full_version()` method returns a pointer to a string, suitable for display, containing Berkeley DB version information. The string includes Oracle family and release numbers, as well as Berkeley DB's traditional major, minor, and patch numbers.

Parameters

family

If **family** is non-NULL, the Oracle family number of the Berkeley DB release is copied to the memory to which it refers.

release

If **release** is non-NULL, the Oracle release number of the Berkeley DB release is copied to the memory to which it refers.

major

If **major** is non-NULL, the major version of the Berkeley DB release is copied to the memory to which it refers.

minor

If **minor** is non-NULL, the minor version of the Berkeley DB release is copied to the memory to which it refers.

patch

If **patch** is non-NULL, the patch version of the Berkeley DB release is copied to the memory to which it refers.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_create_dir()

```
#include <db.h>

int
DB_ENV->get_create_dir(DB_ENV *dbenv, const char **dirp);
```

The `DB_ENV->get_create_dir()` method returns a pointer to the name of the directory to create databases in.

The `DB_ENV->get_create_dir()` method may be called at any time during the life of the application.

The `DB_ENV->get_create_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirp

The `DB_ENV->get_create_dir()` method returns a ponter to the name of the directory in **dirp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_data_dirs()

```
#include <db.h>

int
DB_ENV->get_data_dirs(DB_ENV *dbenv, const char ***dirpp);
```

The `DB_ENV->get_data_dirs()` method returns the NULL-terminated array of directories.

The `DB_ENV->get_data_dirs()` method may be called at any time during the life of the application.

The `DB_ENV->get_data_dirs()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirpp

The `DB_ENV->get_data_dirs()` method returns a reference to the NULL-terminated array of directories in **dirpp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_encrypt_flags()

```
#include <db.h>

int
DB_ENV->get_encrypt_flags(DB_ENV *dbenv, u_int32_t *flagsp);
```

The `DB_ENV->get_encrypt_flags()` method returns the encryption flags.

The `DB_ENV->get_encrypt_flags()` method may be called at any time during the life of the application.

The `DB_ENV->get_encrypt_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB_ENV->get_encrypt_flags()` method returns the encryption flags in **flagsp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_errfile()

```
#include <db.h>

void
DB_ENV->get_errfile(DB_ENV *dbenv, FILE **errfilep);
```

The `DB_ENV->get_errfile()` method returns the FILE * used for displaying additional Berkeley DB error messages. This C library is set using the DB_ENV->set_errfile() (page 255) method.

The `DB_ENV->get_errfile()` method may be called at any time during the life of the application.

Parameters

errfilep

The `DB_ENV->get_errfile()` method returns the FILE * in **errfilep**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_errpfx()

```
#include <db.h>

void
DB_ENV->get_errpfx(DB_ENV *dbenv, const char **errpfxp);
```

The `DB_ENV->get_errpfx()` method returns the error prefix that appears before error messages issued by Berkeley DB. This error prefix is set using the DB_ENV->set_errpfx() (page 257) method.

The `DB_ENV->get_errpfx()` method may be called at any time during the life of the application.

Parameters

errpfxp

The `DB_ENV->get_errpfx()` method returns a reference to the error prefix in **errpfxp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_flags()

```
#include <db.h>

int
DB_ENV->get_flags(DB_ENV *dbenv, u_int32_t *flagsp);
```

The `DB_ENV->get_flags()` method returns the configuration flags set for a DB_ENV handle. These flags are set using the DB_ENV->set_flags() (page 260) method.

The `DB_ENV->get_flags()` method may be called at any time during the life of the application.

The `DB_ENV->get_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB_ENV->get_flags()` method returns the configuration flags in **flagsp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_home()

```
#include <db.h>

int
DB_ENV->get_home(DB_ENV *dbenv, const char **homep);
```

The DB_ENV->get_home() method returns the database environment home directory. This directory is normally identified when the DB_ENV->open() (page 231) method is called.

The DB_ENV->get_home() method may be called at any time during the life of the application.

The DB_ENV->get_home() method returns a non-zero error value on failure and 0 on success.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_intermediate_dir_mode()

```
#include <db.h>

int
DB_ENV->get_intermediate_dir_mode(DB_ENV *dbenv, const char **modep);
```

The `DB_ENV->get_intermediate_dir_mode()` method returns the intermediate directory permissions.

Intermediate directories are directories needed for recovery. Normally, Berkeley DB does not create these directories and will do so only if the DB_ENV->set_intermediate_dir_mode() (page 267) method is called.

The `DB_ENV->get_intermediate_dir_mode()` method may be called at any time during the life of the application.

The `DB_ENV->get_intermediate_dir_mode()` method returns a non-zero error value on failure and 0 on success.

Parameters

modep

The `DB_ENV->get_intermediate_dir_mode()` method returns a reference to the intermediate directory permissions in **modep**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->get_msgfile()

```
#include <db.h>

void
DB_ENV->get_msgfile(DB_ENV *dbenv, FILE **msgfilep);
```

The `DB_ENV->get_msgfile()` method returns the `FILE *` used for displaying messages. This is set using the DB_ENV->set_msgfile() (page 273) method.

The `DB_ENV->get_msgfile()` method may be called at any time during the life of the application.

Parameters

msgfilep

The `DB_ENV->get_msgfile()` method returns the FILE * in **msgfilep**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_msgfile() (page 273)

DB_ENV->get_open_flags()

```
#include <db.h>

int
DB_ENV->get_open_flags(DB_ENV *dbenv, u_int32_t *flagsp);
```

The `DB_ENV->get_open_flags()` method returns the open method flags originally used to create the database environment.

The `DB_ENV->get_open_flags()` method may not be called before the `DB_ENV->open()` method is called.

The `DB_ENV->get_open_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB_ENV->get_open_flags()` method returns the open method flags originally used to create the database environment in **flagsp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->open() (page 231)

DB_ENV->get_shm_key()

```
#include <db.h>

int
DB_ENV->get_shm_key(DB_ENV *dbenv, long *shm_keyp);
```

The `DB_ENV->get_shm_key()` method returns the base segment ID. This is used for Berkeley DB environment shared memory regions created in system memory on VxWorks or systems supporting X/Open-style shared memory interfaces. It may be specified using the DB_ENV->set_shm_key() (page 274) method.

The `DB_ENV->get_shm_key()` method may be called at any time during the life of the application.

The `DB_ENV->get_shm_key()` method returns a non-zero error value on failure and 0 on success.

Parameters

shm_keyp

The `DB_ENV->get_shm_key()` method returns the base segment ID in **shm_keyp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_shm_key() (page 274)

DB_ENV->get_thread_count()

```
#include <db.h>

int
DB_ENV->get_thread_count(DB_ENV *dbenv, u_int32_t *countp);
```

The `DB_ENV->get_thread_count()` method returns the thread count as set by the DB_ENV->set_thread_count() (page 276) method.

The `DB_ENV->get_thread_count()` method may be called at any time during the life of the application.

The `DB_ENV->get_thread_count()` method returns a non-zero error value on failure and 0 on success.

Parameters

countp

The `DB_ENV->get_thread_count()` method returns the thread count in **countp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_thread_count() (page 276)

DB_ENV->get_timeout()

```
#include <db.h>

int
DB_ENV->get_timeout(DB_ENV *dbenv, db_timeout_t *timeoutp,
    u_int32_t flag);
```

The `DB_ENV->get_timeout()` method returns a value, in microseconds, representing either lock or transaction timeouts. These values are set using the DB_ENV->set_timeout() (page 282) method.

The `DB_ENV->get_timeout()` method may be called at any time during the life of the application.

The `DB_ENV->get_timeout()` method returns a non-zero error value on failure and 0 on success.

Parameters

timeoutp

The **timeoutp** parameter references memory into which the timeout value of the specified **flag** parameter is copied.

flag

The **flags** parameter must be set to one of the following values:

- `DB_SET_LOCK_TIMEOUT`

  Return the timeout value for locks in this database environment.

- `DB_SET_TXN_TIMEOUT`

  Return the timeout value for transactions in this database environment.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_timeout() (page 282)

DB_ENV->get_tmp_dir()

```
#include <db.h>

int
DB_ENV->get_tmp_dir(DB_ENV *dbenv, const char **dirp);
```

The `DB_ENV->get_tmp_dir()` method returns the database environment temporary file directory.

The `DB_ENV->get_tmp_dir()` method may be called at any time during the life of the application.

The `DB_ENV->get_tmp_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirp

The `DB_ENV->get_tmp_dir()` method returns a reference to the database environment temporary file directory in **dirp**.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_tmp_dir() (page 284)

DB_ENV->get_verbose()

```
#include <db.h>

int
DB_ENV->get_verbose(DB_ENV *dbenv, u_int32_t which, int *onoffp);
```

The `DB_ENV->get_verbose()` method returns whether the specified **which** parameter is currently set or not. These parameters are set using the DB_ENV->set_verbose() (page 286) method.

The `DB_ENV->get_verbose()` method may be called at any time during the life of the application.

The `DB_ENV->get_verbose()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter is the message value for which configuration is being checked. Must be set to one of the following values:

- `DB_VERB_DEADLOCK`

  Display additional information when doing deadlock detection.

- `DB_VERB_FILEOPS`

  Display additional information when performing filesystem operations such as open, close or rename. May not be available on all platforms.

- `DB_VERB_FILEOPS_ALL`

  Display additional information when performing all filesystem operations, including read and write. May not be available on all platforms.

- `DB_VERB_RECOVERY`

  Display additional information when performing recovery.

- `DB_VERB_REGISTER`

  Display additional information concerning support for the DB_REGISTER flag to the DB_ENV->open() (page 231) method.

- `DB_VERB_REPLICATION`

  Display all detailed information about replication. This includes the information displayed by all of the other DB_VERB_REP_* and DB_VERB_REPMGR_* values.

- `DB_VERB_REP_ELECT`

  Display detailed information about replication elections.

- `DB_VERB_REP_LEASE`

  Display detailed information about replication master leases.

- `DB_VERB_REP_MISC`

  Display detailed information about general replication processing not covered by the other DB_VERB_REP_* values.

- `DB_VERB_REP_MSGS`

  Display detailed information about replication message processing.

- `DB_VERB_REP_SYNC`

  Display detailed information about replication client synchronization.

- `DB_VERB_REP_SYSTEM`

  Saves replication system information to a system-owned file. This value is on by default.

- `DB_VERB_REPMGR_CONNFAIL`

  Display detailed information about Replication Manager connection failures.

- `DB_VERB_REPMGR_MISC`

  Display detailed information about general Replication Manager processing.

- `DB_VERB_WAITSFOR`

  Display the waits-for table when doing deadlock detection.

onoffp

The **onoffp** parameter references memory into which the configuration of the specified **which** parameter is copied.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->log_verify()

```
#include <db.h>
int
DB_ENV->log_verify(const DB_LOG_VERIFY_CONFIG *config);
```

The `DB_ENV->log_verify()` method verifies the integrity of the log records of an environment and writes both error and normal messages to the error/message output facility of the database environment handle.

The `DB_ENV->log_verify()` method does not perform the locking function, even in Berkeley DB environments that are configured with a locking subsystem. Because this function does not access any database files, you can call it even when the environment has other threads of control attached and running.

The `DB_ENV->log_verify()` method is the underlying method used by the `DB_ENV-> db_log_verify` utility. See the `DB_ENV-> db_log_verify` utility source code for an example of using `DB_ENV->log_verify()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The `DB_ENV->log_verify()` method returns DB_LOG_VERIFY_BAD when either log errors are detected or the internal data storage layer does not work. It returns EINVAL if you specify wrong configurations. Unless otherwise specified, the `DB_ENV->log_verify()` method returns a non-zero error value on failure and 0 on success.

Parameters

config

The configuration parameter of type DB_LOG_VERIFY_CONFIG is for the verification of log files. A struct variable of this type must be memset to 0 before setting any configurations to it.

DB_LOG_VERIFY_CONFIG members

temp_envhome

The **temp_envhome** parameter is the home directory of the temporary database environment that is used internally during the verification. It can be NULL, meaning the environment and all databases are in-memory.

cachesize

The **cachesize** parameter specifies the size of the cache of the temporary internal environment in bytes.

dbfile

The **dbfile** parameter specifies that for log records involving a database file, only those related to this database file are verified. Log records not involving database files are verified regardless of this parameter.

dbname

The **dbname** parameter specifies that for log records involving a database file, only those related to this database file are verified. Log records not involving database files are verified regardless of this parameter.

start_lsn and end_lsn

The **start_lsn** and **end_lsn** parameters specify the range of log records from the entire log set, that must be verified. Either of them can be [0][0], to specify an open ended range. If both of them are [0][0] (by default) the entire log is verified.

start_time and end_time

The **start_time** and **end_time** parameters specify range of log records from the entire log set that must be verified for a time range. Either of them can be 0, to specify an open ended range. If both of them are 0 (by default), the entire log is verified.

Note that the time range specified is not precise, because such a time range is converted to an lsn range based on the time points we know from transaction commits and checkpoints.

You can specify either an lsn range or a time range. You can neither specify both nor specify an lsn and a time as a range.

continue_after_fail

The **continue_after_fail** parameter specifies whether or not continue the verification process when an error in the log is detected.

verbose

The **verbose** parameter specifies whether or not to display verbose output during the verification process.

Environment Variables

If the database is opened within a database environment, the environment variable `DB_HOME` can be used as the path of the database environment home.

Errors

The `DB_ENV->log_verify()` method may fail and return one of the following non-zero errors:

EINVAL or DB_LOG_VERIFY_BAD.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->lsn_reset()

```
#include <db.h>

int
DB_ENV->lsn_reset(DB_ENV *dbenv, const char *file, u_int32_t flags);
```

The `DB_ENV->lsn_reset()` method allows database files to be moved from one transactional database environment to another.

Database pages in transactional database environments contain references to the environment's log files (that is, log sequence numbers, or LSNs). Copying or moving a database file from one database environment to another, and then modifying it, can result in data corruption if the LSNs are not first cleared.

Note that LSNs should be reset before moving or copying the database file into a new database environment, rather than moving or copying the database file and then resetting the LSNs. Berkeley DB has consistency checks that may be triggered if an application calls `DB_ENV->lsn_reset()` on a database in a new environment when the database LSNs still reflect the old environment.

The `DB_ENV->lsn_reset()` method modifies the physical file, in-place. Applications should not reset LSNs in files that are currently in use.

The `DB_ENV->lsn_reset()` method may be called at any time during the life of the application.

The `DB_ENV->lsn_reset()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The name of the physical file in which the LSNs are to be cleared.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_ENCRYPT`

  The file contains encrypted databases.

Errors

The `DB_ENV->lsn_reset()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->open()

```
#include <db.h>

int
DB_ENV->open(DB_ENV *dbenv, char *db_home, u_int32_t flags, int mode);
```

The `DB_ENV->open()` method opens a Berkeley DB environment. It provides a structure for creating a consistent environment for processes using one or more of the features of Berkeley DB.

The `DB_ENV->open()` method method returns a non-zero error value on failure and 0 on success. If `DB_ENV->open()` fails, the DB_ENV->close() (page 197) method must be called to discard the DB_ENV handle.

Parameters

db_home

The **db_home** parameter is the database environment's home directory. For more information on **db_home**, and filename resolution in general, see Berkeley DB File Naming. The environment variable **DB_HOME** may be used as the path of the database home, as described in Berkeley DB File Naming.

When using a Unicode build on Windows (the default), the **db_home** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

flags

The **flags** parameter specifies the subsystems that are initialized and how the application's environment affects Berkeley DB file naming, among other things. The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the values described in this section.

Because there are a large number of flags that can be specified, they have been grouped together by functionality. The first group of flags indicates which of the Berkeley DB subsystems should be initialized.

The choice of subsystems initialized for a Berkeley DB database environment is specified by the thread of control initially creating the environment. Any subsequent thread of control joining the environment will automatically be configured to use the same subsystems as were created in the environment (unless the thread of control requests a subsystem not available in the environment, which will fail). Applications joining an environment, able to adapt to whatever subsystems have been configured in the environment, should open the environment without specifying any subsystem flags. Applications joining an environment, requiring specific subsystems from their environments, should open the environment specifying those specific subsystem flags.

- `DB_INIT_CDB`

Initialize locking for the Berkeley DB Concurrent Data Store product. In this mode, Berkeley DB provides multiple reader/single writer access. The only other subsystem that should be specified with the DB_INIT_CDB flag is DB_INIT_MPOOL.

- DB_INIT_LOCK

  Initialize the locking subsystem. This subsystem should be used when multiple processes or threads are going to be reading and writing a Berkeley DB database, so that they do not interfere with each other. If all threads are accessing the database(s) read-only, locking is unnecessary. When the DB_INIT_LOCK flag is specified, it is usually necessary to run a deadlock detector, as well. See db_deadlock and DB_ENV->lock_detect() (page 311) for more information.

- DB_INIT_LOG

  Initialize the logging subsystem. This subsystem should be used when recovery from application or system failure is necessary. If the log region is being created and log files are already present, the log files are reviewed; subsequent log writes are appended to the end of the log, rather than overwriting current log entries.

- DB_INIT_MPOOL

  Initialize the shared memory buffer pool subsystem. This subsystem should be used whenever an application is using any Berkeley DB access method.

- DB_INIT_REP

  Initialize the replication subsystem. This subsystem should be used whenever an application plans on using replication. The DB_INIT_REP flag requires the DB_INIT_TXN and DB_INIT_LOCK flags also be configured.

- DB_INIT_TXN

  Initialize the transaction subsystem. This subsystem should be used when recovery and atomicity of multiple operations are important. The DB_INIT_TXN flag implies the DB_INIT_LOG flag.

The second group of flags govern what recovery, if any, is performed when the environment is initialized:

- DB_RECOVER

  Run normal recovery on this environment before opening it for normal use. If this flag is set, the DB_CREATE and DB_INIT_TXN flags must also be set, because the regions will be removed and re-created, and transactions are required for application recovery.

- DB_RECOVER_FATAL

  Run catastrophic recovery on this environment before opening it for normal use. If this flag is set, the DB_CREATE and DB_INIT_TXN flags must also be set, because the regions will be removed and re-created, and transactions are required for application recovery.

A standard part of the recovery process is to remove the existing Berkeley DB environment and create a new one in which to perform recovery. If the thread of control performing recovery does not specify the correct region initialization information (for example, the correct memory pool cache size), the result can be an application running in an environment with incorrect cache and other subsystem sizes. For this reason, the thread of control performing recovery should specify correct configuration information before calling the `DB_ENV->open()` method; or it should remove the environment after recovery is completed, leaving creation of the correctly sized environment to a subsequent call to the `DB_ENV->open()` method.

All Berkeley DB recovery processing must be single-threaded; that is, only a single thread of control may perform recovery or access a Berkeley DB environment while recovery is being performed. Because it is not an error to specify `DB_RECOVER` for an environment for which no recovery is required, it is reasonable programming practice for the thread of control responsible for performing recovery and creating the environment to always specify the `DB_CREATE` and `DB_RECOVER` flags during startup.

The third group of flags govern file-naming extensions in the environment:

- `DB_USE_ENVIRON`

  The Berkeley DB process' environment may be permitted to specify information to be used when naming files; see Berkeley DB File Naming. Because permitting users to specify which files are used can create security problems, environment information will be used in file naming for all users only if the `DB_USE_ENVIRON` flag is set.

- `DB_USE_ENVIRON_ROOT`

  The Berkeley DB process' environment may be permitted to specify information to be used when naming files; see Berkeley DB File Naming. Because permitting users to specify which files are used can create security problems, if the `DB_USE_ENVIRON_ROOT` flag is set, environment information will be used in file naming only for users with appropriate permissions (for example, users with a user-ID of 0 on `UNIX` systems).

Finally, there are a few additional unrelated flags:

- `DB_CREATE`

  Cause Berkeley DB subsystems to create any underlying files, as necessary.

- `DB_LOCKDOWN`

  Lock shared Berkeley DB environment files and memory-mapped databases into memory.

- `DB_FAILCHK`

  Internally call the DB_ENV->failchk() (page 206) method as part of opening the environment. When `DB_FAILCHK` is specified, a check is made to ensure all `DB_ENV->failchk()` prerequisites are meet.

  If the `DB_FAILCHK` flag is used in conjunction with the `DB_REGISTER` flag, then a check will be made to see if the environment needs recovery. If recovery is needed, a call will be made to the DB_ENV->failchk() method to release any database reads locks held by the thread of control that exited and, if needed, to abort the unresolved transaction. If DB_ENV->failchk() determines environment recovery is still required, the recovery actions for DB_REGISTER will be followed.

If the DB_FAILCHK flag is not used in conjunction with the DB_REGISTER flag, then make an internal call to DB_ENV->failchk() as the last step of opening the environment. If DB_ENV->failchk() determines database environment recovery is required, DB_RUNRECOVERY will be returned.

- DB_PRIVATE

  Allocate region memory from the heap instead of from memory backed by the filesystem or system shared memory.

  This flag implies the environment will only be accessed by a single process (although that process may be multithreaded). This flag has two effects on the Berkeley DB environment. First, all underlying data structures are allocated from per-process memory instead of from shared memory that is accessible to more than a single process. Second, mutexes are only configured to work between threads.

  This flag should not be specified if more than a single process is accessing the environment because it is likely to cause database corruption and unpredictable behavior. For example, if both a server application and Berkeley DB utilities (for example, db_archive, db_checkpoint or db_stat) are expected to access the environment, the DB_PRIVATE flag should not be specified.

  See Shared Memory Regions for more information.

- DB_REGISTER

  Check to see if recovery needs to be performed before opening the database environment. (For this check to be accurate, all processes using the environment must specify DB_REGISTER when opening the environment.) If recovery needs to be performed for any reason (including the initial use of the DB_REGISTER flag), and DB_RECOVER is also specified, recovery will be performed and the open will proceed normally. If recovery needs to be performed and DB_RECOVER is not specified, DB_RUNRECOVERY will be returned. If recovery does not need to be performed, the DB_RECOVER flag will be ignored. See Architecting Transactional Data Store applications for more information.

- DB_SYSTEM_MEM

  Allocate region memory from system shared memory instead of from heap memory or memory backed by the filesystem.

  See Shared Memory Regions for more information.

- DB_THREAD

  Cause the DB_ENV handle returned by DB_ENV->open() to be *free-threaded;* that is, concurrently usable by multiple threads in the address space. The DB_THREAD flag should be specified if the DB_ENV handle will be concurrently used by more than one thread in the process, or if any DB handles opened in the scope of the DB_ENV handle will be concurrently used by more than one thread in the process.

This flag is required when using the Replication Manager.

mode

On Windows systems, the mode parameter is ignored.

On UNIX systems or in IEEE/ANSI Std 1003.1 (POSIX) environments, files created by Berkeley DB are created with mode **mode** (as described in **chmod**(2)) and modified by the process' umask value at the time of creation (see **umask**(2)). Created files are owned by the process owner; the group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB. System shared memory segments created by Berkeley DB are created with mode **mode**, unmodified by the process' umask value. If **mode** is 0, Berkeley DB will use a default mode of readable and writable by both owner and group.

Errors

The `DB_ENV->open()` method may fail and return one of the following non-zero errors:

DB_RUNRECOVERY

Either the `DB_REGISTER` flag was specified, a failure occurred, and no recovery flag was specified, or the `DB_FAILCHK` flag was specified and recovery was deemed necessary.

DB_VERSION_MISMATCH

The version of the Berkeley DB library doesn't match the version that created the database environment.

EAGAIN

The shared memory region was locked and (repeatedly) unavailable.

EINVAL

If the `DB_THREAD` flag was specified and fast mutexes are not available for this architecture; The `DB_HOME` or `TMPDIR` environment variables were set, but empty; An incorrectly formatted **NAME VALUE** entry or line was found; or if an invalid flag value or parameter was specified.

ENOSPC

HP-UX only: Due to the constraints of the PA-RISC memory architecture, HP-UX does not allow a process to map a file into its address space multiple times. For this reason, each Berkeley DB environment may be opened only once by a process on HP-UX; that is, calls to `DB_ENV->open()` will fail if the specified Berkeley DB environment has been opened and not subsequently closed.

ENOENT

The file or directory does not exist.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->remove()

```
#include <db.h>

int
DB_ENV->remove(DB_ENV *dbenv, char *db_home, u_int32_t flags);
```

The `DB_ENV->remove()` method destroys a Berkeley DB environment if it is not currently in use. The environment regions, including any backing files, are removed. Any log or database files and the environment directory are not removed.

If there are processes that have called DB_ENV->open() (page 231) without calling DB_ENV->close() (page 197) (that is, there are processes currently using the environment), `DB_ENV->remove()` will fail without further action unless the DB_FORCE flag is set, in which case `DB_ENV->remove()` will attempt to remove the environment, regardless of any processes still using it.

The result of attempting to forcibly destroy the environment when it is in use is unspecified. Processes using an environment often maintain open file descriptors for shared regions within it. On UNIX systems, the environment removal will usually succeed, and processes that have already joined the region will continue to run in that region without change. However, processes attempting to join the environment will either fail or create new regions. On other systems in which the **unlink**(2) system call will fail if any process has an open file descriptor for the file (for example Windows/NT), the region removal will fail.

Calling `DB_ENV->remove()` should not be necessary for most applications because the Berkeley DB environment is cleaned up as part of normal database recovery procedures. However, applications may want to call `DB_ENV->remove()` as part of application shut down to free up system resources. For example, if the DB_SYSTEM_MEM flag was specified to DB_ENV->open() (page 231), it may be useful to call `DB_ENV->remove()` in order to release system shared memory segments that have been allocated. Or, on architectures in which mutexes require allocation of underlying system resources, it may be useful to call `DB_ENV->remove()` in order to release those resources. Alternatively, if recovery is not required because no database state is maintained across failures, and no system resources need to be released, it is possible to clean up an environment by simply removing all the Berkeley DB files in the database environment's directories.

In multithreaded applications, only a single thread may call the `DB_ENV->remove()` method.

A DB_ENV handle that has already been used to open an environment should not be used to call the `DB_ENV->remove()` method; a new DB_ENV handle should be created for that purpose.

After `DB_ENV->remove()` has been called, regardless of its return, the Berkeley DB environment handle may not be accessed again.

The `DB_ENV->remove()` method returns a non-zero error value on failure and 0 on success.

Parameters

db_home

The **db_home** parameter names the database environment to be removed.

When using a Unicode build on Windows (the default), the **db_home** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_FORCE`

  If set, the environment is removed, regardless of any processes that may still using it, and no locks are acquired during this process. (Generally, this flag is specified only when applications were unable to shut down cleanly, and there is a risk that an application may have died holding a Berkeley DB lock.)

- `DB_USE_ENVIRON`

  The Berkeley DB process' environment may be permitted to specify information to be used when naming files; see Berkeley DB File Naming. Because permitting users to specify which files are used can create security problems, environment information will be used in file naming for all users only if the `DB_USE_ENVIRON` flag is set.

- `DB_USE_ENVIRON_ROOT`

  The Berkeley DB process' environment may be permitted to specify information to be used when naming files; see Berkeley DB File Naming. Because permitting users to specify which files are used can create security problems, if the `DB_USE_ENVIRON_ROOT` flag is set, environment information will be used in file naming only for users with appropriate permissions (for example, users with a user-ID of 0 on `UNIX` systems).

Errors

The `DB_ENV->remove()` method may fail and return one of the following non-zero errors:

EBUSY

The shared memory region was in use and the force flag was not set.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_alloc()

```
#include <db.h>

int
DB_ENV->set_alloc(DB_ENV *dbenv,
    void *(*app_malloc)(size_t),
    void *(*app_realloc)(void *, size_t),
    void (*app_free)(void *));
```

Set the allocation functions used by the DB_ENV and DB methods to allocate or free memory owned by the application.

There are a number of interfaces in Berkeley DB where memory is allocated by the library and then given to the application. For example, the DB_DBT_MALLOC flag, when specified in the DBT object, will cause the DB methods to allocate and reallocate memory which then becomes the responsibility of the calling application. Other examples are the Berkeley DB interfaces which return statistical information to the application: DB->stat() (page 130), DB_ENV->lock_stat() (page 319), DB_ENV->log_archive() (page 337), DB_ENV->log_stat() (page 351), DB_ENV->memp_stat() (page 383), and DB_ENV->txn_stat() (page 549). There is one method in Berkeley DB where memory is allocated by the application and then given to the library: the callback specified to DB->associate() (page 6).

On systems in which there may be multiple library versions of the standard allocation routines (notably Windows NT), transferring memory between the library and the application will fail because the Berkeley DB library allocates memory from a different heap than the application uses to free it. To avoid this problem, the `DB_ENV->set_alloc()` and DB->set_alloc() (page 79) methods can be used to pass Berkeley DB references to the application's allocation routines.

It is not an error to specify only one or two of the possible allocation function parameters to these interfaces; however, in that case the specified interfaces must be compatible with the standard library interfaces, as they will be used together. The functions specified must match the calling conventions of the ANSI C X3.159-1989 (ANSI C) library routines of the same name.

The `DB_ENV->set_alloc()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_alloc()` method may not be called after the DB_ENV->open() (page 231) method is called.

The `DB_ENV->set_alloc()` method returns a non-zero error value on failure and 0 on success.

Parameters

app_malloc

The **app_malloc** parameter is the application-specified malloc function.

app_realloc

The **app_realloc** parameter is the application-specified realloc function.

app_free

The **app_free** parameter is the application-specified free function.

Errors

The `DB_ENV->set_alloc()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_app_dispatch()

```
#include <db.h>

int
DB_ENV->set_app_dispatch(DB_ENV *dbenv,
    int (*tx_recover)(DB_ENV *dbenv,
    DBT *log_rec, DB_LSN *lsn, db_recops op));
```

Declare a function to be called during transaction abort and recovery to process application-specific log records.

The `DB_ENV->set_app_dispatch()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_app_dispatch()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_app_dispatch()` must be consistent with the existing environment or corruption can occur.

The `DB_ENV->set_app_dispatch()` method returns a non-zero error value on failure and 0 on success.

Parameters

tx_recover

The **tx_recover** parameter is the application's abort and recovery function. The function takes four parameters:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment handle.

- `log_rec`

  The **log_rec** parameter is a log record.

- `lsn`

  The **lsn** parameter is a log sequence number.

- `op`

  The **op** parameter is one of the following values:

  - `DB_TXN_BACKWARD_ROLL`

    The log is being read backward to determine which transactions have been committed and to abort those operations that were not; undo the operation described by the log record.

- DB_TXN_FORWARD_ROLL

  The log is being played forward; redo the operation described by the log record.

- DB_TXN_ABORT

  The log is being read backward during a transaction abort; undo the operation described by the log record.

- DB_TXN_APPLY

  The log is being applied on a replica site; redo the operation described by the log record.

- DB_TXN_PRINT

  The log is being printed for debugging purposes; print the contents of this log record in the desired format.

The DB_TXN_FORWARD_ROLL and DB_TXN_APPLY operations frequently imply the same actions, redoing changes that appear in the log record, although if a recovery function is to be used on a replication client where reads may be taking place concurrently with the processing of incoming messages, DB_TXN_APPLY operations should also perform appropriate locking. The macro DB_REDO(op) checks that the operation is one of DB_TXN_FORWARD_ROLL or DB_TXN_APPLY, and should be used in the recovery code to refer to the conditions under which operations should be redone. Similarly, the macro DB_UNDO(op) checks if the operation is one of DB_TXN_BACKWARD_ROLL or DB_TXN_ABORT.

The function must return 0 on success and either **errno** or a value outside of the Berkeley DB error name space on failure.

Errors

The `DB_ENV->set_app_dispatch()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->set_data_dir()

```
#include <db.h>

int
DB_ENV->set_data_dir(DB_ENV *dbenv, const char *dir);
```

Note

This interface has been deprecated. You should use DB_ENV->add_data_dir() (page 195) and DB_ENV->set_create_dir() (page 245) instead.

Set the path of a directory to be used as the location of the access method database files. Paths specified to the DB->open() (page 66) function will be searched relative to this path. Paths set using this method are additive, and specifying more than one will result in each specified directory being searched for database files. If any directories are specified, database files will always be created in the first path specified.

If no database directories are specified, database files must be named either by absolute paths or relative to the environment home directory. See Berkeley DB File Naming for more information.

The database environment's data directories may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_data_dir", one or more whitespace characters, and the directory name.

The `DB_ENV->set_data_dir()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_data_dir()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_data_dir()` must be consistent with the existing environment or corruption can occur.

The `DB_ENV->set_data_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** parameter is a directory to be used as a location for database files.

When using a Unicode build on Windows (the default), this argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB_ENV->set_data_dir()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_create_dir()

```
#include <db.h>

int
DB_ENV->set_create_dir(DB_ENV *dbenv, const char *dir);
```

Sets the path of a directory to be used as the location to create the access method database files. When the DB->open() (page 66) function is used to create a file it will be created relative to this path.

If no database directories are specified, database files will be created either by absolute paths or relative to the environment home directory. See Berkeley DB File Naming for more information.

The database environment's create directory may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_create_dir", one or more whitespace characters, and the directory name.

The `DB_ENV->set_create_dir()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_create_dir()` method may be called at any time.

The `DB_ENV->set_create_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** parameter is a directory to be used to create database files. This directory must be one of the directories specified via a call to DB_ENV->add_data_dir() (page 195)

When using a Unicode build on Windows (the default), this argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB_ENV->set_create_dir()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_encrypt()

```
#include <db.h>

int
DB_ENV->set_encrypt(DB_ENV *dbenv, const char *passwd, u_int32_t flags);
```

Set the password used by the Berkeley DB library to perform encryption and decryption.

The `DB_ENV->set_encrypt()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_encrypt()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_encrypt()` must be consistent with the existing environment or an error will be returned.

The `DB_ENV->set_encrypt()` method returns a non-zero error value on failure and 0 on success.

Parameters

passwd

The **passwd** parameter is the password used to perform encryption and decryption.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_ENCRYPT_AES`

  Use the Rijndael/AES (also known as the Advanced Encryption Standard and Federal Information Processing Standard (FIPS) 197) algorithm for encryption or decryption.

Errors

The `DB_ENV->set_encrypt()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

EOPNOTSUPP

Cryptography is not available in this Berkeley DB release.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_event_notify()

```
#include <db.h>

int
DB_ENV->set_event_notify(DB_ENV *dbenv,
    void (*db_event_fcn)(DB_ENV *dbenv, u_int32_t event,
    void *event_info));
```

The `DB_ENV->set_event_notify()` method configures a callback function which is called to notify the process of specific Berkeley DB events.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

The `DB_ENV->set_event_notify()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_event_notify()` method may be called at any time during the life of the application.

The `DB_ENV->set_event_notify()` method returns a non-zero error value on failure and 0 on success.

Parameters

db_event_fcn

The **db_event_fcn** parameter is the application's event notification function. The function takes three parameters:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment handle.

- `event`

  The **event** parameter is one of the following values:

  - `DB_EVENT_PANIC`

    Errors can occur in the Berkeley DB library where the only solution is to shut down the application and run recovery (for example, if Berkeley DB is unable to allocate heap memory). In such cases, the Berkeley DB methods will return DB_RUNRECOVERY. It is often easier to simply exit the application when such errors occur rather than gracefully return up the stack.

When **event** is set to DB_EVENT_PANIC, the database environment has failed. All threads of control in the database environment should exit the environment, and recovery should be run.

- DB_EVENT_REG_ALIVE

  Recovery is needed in an environment where the DB_REGISTER flag was specified on the DB_ENV->open() (page 231) method and there is a process attached to the environment. The callback function is triggered once for each process attached.

  The **event_info** parameter points to a pid_t value containing the process identifier (pid) of the process the Berkeley DB library detects is attached to the environment.

- DB_EVENT_REG_PANIC

  Recovery is needed in an environment where the DB_REGISTER flag was specified on the DB_ENV->open() (page 231) method. All threads of control in the database environment should exit the environment.

  This event is different than the DB_EVENT_PANIC event because it can only be triggered when DB_REGISTER was specified. It can be used to distinguish between the case when a process dies in the environment and recovery is initiated versus the case when an error happened (for example, if Berkeley DB is unable to allocate heap memory)

- DB_EVENT_REP_CLIENT

  The local site is now a replication client.

- DB_EVENT_REP_ELECTED

  The local replication site has just won an election. An application using the Base replication API should arrange for a call to the DB_ENV->rep_start() (page 484) method after receiving this event, to reconfigure the local environment as a replication master.

  Replication Manager applications may safely ignore this event. The Replication Manager calls DB_ENV->rep_start() (page 484) automatically on behalf of the application when appropriate (resulting in firing of the DB_EVENT_REP_MASTER event).

- DB_EVENT_REP_MASTER

  The local site is now the master site of its replication group. It is the application's responsibility to begin acting as the master environment.

- DB_EVENT_REP_NEWMASTER

  The replication group of which this site is a member has just established a new master; the local site is not the new master. The **event_info** parameter points to an integer containing the environment ID of the new master.

- DB_EVENT_REP_DUPMASTER

Replication Manager has detected a duplicate master situation, and has changed the local site to the client role as a result. If the DB_REPMGR_CONF_ELECTIONS (page 469) configuration parameter has been turned off, the application should now choose and assign the correct master site. If DB_REPMGR_CONF_ELECTIONS is turned on, the application may ignore this event.

The DB_EVENT_REP_DUPMASTER event is provided only to applications configured for the replication manager.

- DB_EVENT_REP_ELECTION_FAILED

  Replication Manager tried to run an election to choose a master site, but the election failed due to lack of timely participation by a sufficient number of other sites. Replication Manager will automatically retry the election later. This event is for information only.

  The DB_EVENT_REP_ELECTION_FAILED event is provided only to applications configured for the replication manager.

- DB_EVENT_REP_MASTER_FAILURE

  A Replication Manager client site has detected the loss of connection to the master site. If the DB_REPMGR_CONF_ELECTIONS (page 469) configuration parameter is turned on, Replication Manager will automatically start an election in order to choose a new master. In this case, this event may be ignored.

  When DB_REPMGR_CONF_ELECTIONS is turned off, the application should choose and assign a new master. Failure to do so means that your replication group has no master, and so it cannot service write requests.

  The DB_EVENT_REP_MASTER_FAILURE event is provided only to applications configured for the replication manager.

- DB_EVENT_REP_PERM_FAILED

  The replication manager did not receive enough acknowledgements (based on the acknowledgement policy configured with DB_ENV->repmgr_set_ack_policy() (page 497)) to ensure a transaction's durability within the replication group. The transaction will be flushed to the master's local disk storage for durability.

  The DB_EVENT_REP_PERM_FAILED event is provided only to applications configured for the replication manager.

- DB_EVENT_REP_STARTUPDONE

  The client has completed startup synchronization and is now processing live log records received from the master.

- DB_EVENT_WRITE_FAILED

  A Berkeley DB write to stable storage failed.

- `event_info`

  The **event_info** parameter may reference memory which contains additional information describing an event. By default, **event_info** is NULL; specific events may pass non-NULL values, in which case the event will also describe the memory's structure.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_errcall()

```
#include <db.h>

void
DB_ENV->set_errcall(DB_ENV *dbenv, void (*db_errcall_fcn)
    (const DB_ENV *dbenv, const char *errpfx, const char *msg));
```

When an error occurs in the Berkeley DB library, a Berkeley DB error or an error return value is returned by the interface. In some cases, however, the **errno** value may be insufficient to completely describe the cause of the error, especially during initial application debugging.

The `DB_ENV->set_errcall()` and DB_ENV->set_errcall() (page 253) methods are used to enhance the mechanism for reporting error messages to the application. In some cases, when an error occurs, Berkeley DB will call **db_errcall_fcn** with additional error information. It is up to the **db_errcall_fcn** function to display the error message in an appropriate manner.

Setting **db_errcall_fcn** to NULL unconfigures the callback interface.

Alternatively, you can use the DB->set_errfile() (page 99) or DB->set_errfile() (page 255) methods to display the additional information via a C library `FILE *`.

This error-logging enhancement does not slow performance or significantly increase application size, and may be run during normal operation as well as during application debugging.

The `DB_ENV->set_errcall()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_errcall()` method may be called at any time during the life of the application.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_errcall_fcn

The **db_errcall_fcn** parameter is the application-specified error reporting function. The function takes three parameters:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment.

- `errpfx`

The **errpfx** parameter is the prefix string (as previously set by DB->set_errpfx() (page 101) or DB_ENV->set_errpfx() (page 257)).

- `msg`

  The **msg** parameter is the error message string.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_errfile()

```
#include <db.h>

void
DB_ENV->set_errfile(DB_ENV *dbenv, FILE *errfile);
```

When an error occurs in the Berkeley DB library, a Berkeley DB error or an error return value is returned by the interface. In some cases, however, the return value may be insufficient to completely describe the cause of the error especially during initial application debugging.

The `DB_ENV->set_errfile()` and DB->set_errfile() (page 99) methods are used to enhance the mechanism for reporting error messages to the application by setting a C library FILE * to be used for displaying additional Berkeley DB error messages. In some cases, when an error occurs, Berkeley DB will output an additional error message to the specified file reference.

Alternatively, you can use the DB_ENV->set_errcall() (page 253) or DB->set_errcall() (page 97) methods to capture the additional error information in a way that does not use C library FILE *'s.

The error message will consist of the prefix string and a colon ("**:**") (if a prefix string was previously specified using DB->set_errpfx() (page 101) or DB_ENV->set_errpfx() (page 257)), an error string, and a trailing <newline> character.

The default configuration when applications first create DB or DB_ENV handles is as if the DB->set_errfile() (page 99) or `DB_ENV->set_errfile()` methods were called with the standard error output (stderr) specified as the FILE * argument. Applications wanting no output at all can turn off this default configuration by calling the DB->set_errfile() (page 99) or `DB_ENV->set_errfile()` methods with NULL as the FILE * argument. Additionally, explicitly configuring the error output channel using any of the following methods will also turn off this default output for the application:

- `DB_ENV->set_errfile()`
- DB->set_errfile() (page 99)
- DB_ENV->set_errcall() (page 253)
- DB->set_errcall() (page 97)

This error logging enhancement does not slow performance or significantly increase application size, and may be run during normal operation as well as during application debugging.

The `DB_ENV->set_errfile()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_errfile()` method may be called at any time during the life of the application.

Parameters

errfile

The **errfile** parameter is a C library FILE * to be used for displaying additional Berkeley DB error information.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_errpfx()

```
#include <db.h>

void
DB_ENV->set_errpfx(DB_ENV *dbenv, const char *errpfx);
```

Set the prefix string that appears before error messages issued by Berkeley DB.

The DB->set_errpfx() (page 101) and `DB_ENV->set_errpfx()` methods do not copy the memory to which the **errpfx** parameter refers; rather, they maintain a reference to it. Although this allows applications to modify the error message prefix at any time (without repeatedly calling the interfaces), it means the memory must be maintained until the handle is closed.

The `DB_ENV->set_errpfx()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_errpfx()` method may be called at any time during the life of the application.

Parameters

errpfx

The **errpfx** parameter is the application-specified error prefix for additional error messages.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_feedback()

```
#include <db.h>

int
DB_ENV->set_feedback(DB_ENV *dbenv,
    void (*db_feedback_fcn)(DB_ENV *dbenv, int opcode, int percent));
```

Some operations performed by the Berkeley DB library can take non-trivial amounts of time. The `DB_ENV->set_feedback()` method can be used by applications to monitor progress within these operations. When an operation is likely to take a long time, Berkeley DB will call the specified callback function with progress information.

> ### Note
>
> Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

It is up to the callback function to display this information in an appropriate manner.

The `DB_ENV->set_feedback()` method configures operations performed using the specified DB_ENV handle.

The `DB_ENV->set_feedback()` method may be called at any time during the life of the application.

The `DB_ENV->set_feedback()` method returns a non-zero error value on failure and 0 on success.

Parameters

db_feedback_fcn

The **db_feedback_fcn** parameter is the application-specified feedback function called to report Berkeley DB operation progress. The callback function must take three parameters:

- `dbenv`

  The **dbenv** parameter is a reference to the enclosing database environment.

- `opcode`

  The **opcode** parameter is an operation code. The **opcode** parameter may take on any of the following values:

  - `DB_RECOVER`

    The environment is being recovered.

- `percent`

The **percent** parameter is the percent of the operation that has been completed, specified as an integer value between 0 and 100.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_flags()

```
#include <db.h>

int
DB_ENV->set_flags(DB_ENV *dbenv, u_int32_t flags, int onoff);
```

Configure a database environment.

The database environment's flag values may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_flags", one or more whitespace characters, and the method flag parameter as a string, and optionally one or more whitespace characters, and the string "on" or "off". If the optional string is omitted, the default is "on"; for example, "set_flags DB_TXN_NOSYNC" or "set_flags DB_TXN_NOSYNC on". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_AUTO_COMMIT`

  If set, DB handle operations for which no explicit transaction handle was specified, and which modify databases in the database environment, will be automatically enclosed within a transaction.

  Calling `DB_ENV->set_flags()` with this flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set this flag or the flag should be specified in the DB_CONFIG configuration file.

  This flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_CDB_ALLDB`

  If set, Berkeley DB Concurrent Data Store applications will perform locking on an environment-wide basis rather than on a per-database basis.

  Calling `DB_ENV->set_flags()` with the DB_CDB_ALLDB flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_CDB_ALLDB flag or the flag should be specified in the DB_CONFIG configuration file.

The DB_CDB_ALLDB flag may be used to configure Berkeley DB only before the DB_ENV->open() (page 231) method is called.

- `DB_DIRECT_DB`

  Turn off system buffering of Berkeley DB database files to avoid double caching.

  Calling `DB_ENV->set_flags()` with the DB_DIRECT_DB flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_DIRECT_DB flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_DIRECT_DB flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_DSYNC_DB`

  Configure Berkeley DB to flush database writes to the backing disk before returning from the write system call, rather than flushing database writes explicitly in a separate system call, as necessary. This is only available on some systems (for example, systems supporting the IEEE/ANSI Std 1003.1 (POSIX) standard O_DSYNC flag, or systems supporting the Windows FILE_FLAG_WRITE_THROUGH flag). This flag may result in inaccurate file modification times and other file-level information for Berkeley DB database files. This flag will almost certainly result in a performance decrease on most systems. This flag is only applicable to certain filesysystems (for example, the Veritas VxFS filesystem), where the filesystem's support for trickling writes back to stable storage behaves badly (or more likely, has been misconfigured).

  Calling `DB_ENV->set_flags()` with the DB_DSYNC_DB flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_DSYNC_DB flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_DSYNC_DB flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_MULTIVERSION`

  If set, all databases in the environment will be opened as if DB_MULTIVERSION is passed to the DB->open() (page 66) method. This flag will be ignored for queue databases for which DB_MULTIVERSION is not supported.

  Calling `DB_ENV->set_flags()` with the DB_MULTIVERSION flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_MULTIVERSION flag or the flag should be specified in the DB_CONFIG configuration file.

The DB_MULTIVERSION flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_NOLOCKING`

  If set, Berkeley DB will grant all requested mutual exclusion mutexes and database locks without regard for their actual availability. This functionality should never be used for purposes other than debugging.

  Calling `DB_ENV->set_flags()` with the DB_NOLOCKING flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle).

  The DB_NOLOCKING flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_NOMMAP`

  If set, Berkeley DB will copy read-only database files into the local cache instead of potentially mapping them into process memory (see the description of the DB_ENV->set_mp_mmapsize() (page 399) method for further information).

  Calling `DB_ENV->set_flags()` with the DB_NOMMAP flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_NOMMAP flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_NOMMAP flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_NOPANIC`

  If set, Berkeley DB will ignore any panic state in the database environment. (Database environments in a panic state normally refuse all attempts to call Berkeley DB functions, returning DB_RUNRECOVERY.) This functionality should never be used for purposes other than debugging.

  Calling `DB_ENV->set_flags()` with the DB_NOPANIC flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle).

  The DB_NOPANIC flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_OVERWRITE`

  Overwrite files stored in encrypted formats before deleting them. Berkeley DB overwrites files using alternating 0xff, 0x00 and 0xff byte patterns. For file overwriting to be effective, the underlying file must be stored on a fixed-block filesystem. Systems with journaling or logging filesystems will require operating system support and probably modification of the Berkeley DB sources.

Calling `DB_ENV->set_flags()` with the DB_OVERWRITE flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle).

The DB_OVERWRITE flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_PANIC_ENVIRONMENT`

  If set, Berkeley DB will set the panic state for the database environment. (Database environments in a panic state normally refuse all attempts to call Berkeley DB functions, returning DB_RUNRECOVERY.) This flag may not be specified using the environment's DB_CONFIG file.

  Calling `DB_ENV->set_flags()` with the DB_PANIC_ENVIRONMENT flag affects the database environment, including all threads of control accessing the database environment.

  The DB_PANIC_ENVIRONMENT flag may be used to configure Berkeley DB only after the DB_ENV->open() (page 231) method is called.

- `DB_REGION_INIT`

  In some applications, the expense of page-faulting the underlying shared memory regions can affect performance. (For example, if the page-fault occurs while holding a lock, other lock requests can convoy, and overall throughput may decrease.) If set, Berkeley DB will page-fault shared regions into memory when initially creating or joining a Berkeley DB environment. In addition, Berkeley DB will write the shared regions when creating an environment, forcing the underlying virtual memory and filesystems to instantiate both the necessary memory and the necessary disk space. This can also avoid out-of-disk space failures later on.

  Calling `DB_ENV->set_flags()` with the DB_REGION_INIT flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_REGION_INIT flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_REGION_INIT flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_TIME_NOTGRANTED`

  If set, database calls timing out based on lock or transaction timeout values will return DB_LOCK_NOTGRANTED instead of DB_LOCK_DEADLOCK. This allows applications to distinguish between operations which have deadlocked and operations which have exceeded their time limits.

  Calling `DB_ENV->set_flags()` with the DB_TIME_NOTGRANTED flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_TIME_NOTGRANTED flag or the flag should be specified in the DB_CONFIG configuration file.

The `DB_TIME_NOTGRANTED` flag may be used to configure Berkeley DB at any time during the life of the application.

Note that the DB_ENV->lock_get() (page 313) and DB_ENV->lock_vec() (page 326) methods are unaffected by this flag.

- `DB_TXN_NOSYNC`

  If set, Berkeley DB will not write or synchronously flush the log on transaction commit. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the application or system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how many log updates can fit into the log buffer, how often the operating system flushes dirty buffers to disk, and how often the log is checkpointed.

  Calling `DB_ENV->set_flags()` with the DB_TXN_NOSYNC flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_TXN_NOSYNC flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_TXN_NOSYNC flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_TXN_NOWAIT`

  If set and a lock is unavailable for any Berkeley DB operation performed in the context of a transaction, cause the operation to return DB_LOCK_DEADLOCK (or DB_LOCK_NOTGRANTED if configured using the DB_TIME_NOTGRANTED flag).

  Calling `DB_ENV->set_flags()` with the DB_TXN_NOWAIT flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_TXN_NOWAIT flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_TXN_NOWAIT flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_TXN_SNAPSHOT`

  If set, all transactions in the environment will be started as if DB_TXN_SNAPSHOT were passed to the DB_ENV->txn_begin() (page 544) method, and all non-transactional cursors will be opened as if DB_TXN_SNAPSHOT were passed to the DB->cursor() (page 149) method.

  Calling `DB_ENV->set_flags()` with the DB_TXN_SNAPSHOT flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_TXN_SNAPSHOT flag or the flag should be specified in the DB_CONFIG configuration file.

The DB_TXN_SNAPSHOT flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_TXN_WRITE_NOSYNC`

  If set, Berkeley DB will write, but will not synchronously flush, the log on transaction commit. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how often the system flushes dirty buffers to disk and how often the log is checkpointed.

  Calling `DB_ENV->set_flags()` with the DB_TXN_WRITE_NOSYNC flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_TXN_WRITE_NOSYNC flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_TXN_WRITE_NOSYNC flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_YIELDCPU`

  If set, Berkeley DB will yield the processor immediately after each page or mutex acquisition. This functionality should never be used for purposes other than stress testing.

  Calling `DB_ENV->set_flags()` with the DB_YIELDCPU flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_YIELDCPU flag or the flag should be specified in the DB_CONFIG configuration file.

  The DB_YIELDCPU flag may be used to configure Berkeley DB at any time during the life of the application.

onoff

If the **onoff** parameter is zero, the specified flags are cleared; otherwise they are set.

Errors

The `DB_ENV->set_flags()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_intermediate_dir_mode()

```
#include <db.h>

int
DB_ENV->set_intermediate_dir_mode(DB_ENV *dbenv, const char *mode);
```

By default, Berkeley DB does not create intermediate directories needed for recovery, that is, if the file **/a/b/c/mydatabase** is being recovered, and the directory path **b/c** does not exist, recovery will fail. This default behavior is because Berkeley DB does not know what permissions are appropriate for intermediate directory creation, and creating the directory might result in a security problem.

The `DB_ENV->set_intermediate_dir_mode()` method causes Berkeley DB to create any intermediate directories needed during recovery, using the specified permissions.

On UNIX systems or in IEEE/ANSI Std 1003.1 (POSIX) environments, created directories are owned by the process owner; the group ownership of created directories is based on the system and directory defaults, and is not further specified by Berkeley DB.

The database environment's intermediate directory permissions may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_intermediate_dir_mode", one or more whitespace characters, and the directory permissions. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_intermediate_dir_mode()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_intermediate_dir_mode()` method may not be called after the DB_ENV->open() (page 231) method is called.

The `DB_ENV->set_intermediate_dir_mode()` method returns a non-zero error value on failure and 0 on success.

Parameters

mode

The **mode** parameter specifies the directory permissions.

Directory permissions are interpreted as a string of nine characters, using the character set **r** (read), **w** (write), **x** (execute or search), and **-** (none). The first character is the read permissions for the directory owner (set to either **r** or **-**). The second character is the write permissions for the directory owner (set to either **w** or **-**). The third character is the execute permissions for the directory owner (set to either **x** or **-**).

Similarly, the second set of three characters are the read, write and execute/search permissions for the directory group, and the third set of three characters are the read, write and execute/search permissions for all others. For example, the string **rwx------** would configure read, write and execute/search access for the owner only. The string **rwxrwx---** would configure read, write and execute/search access for both the owner and the group. The string **rwxr-----** would configure read, write and execute/search access for the directory owner and read-only access for the directory group.

Errors

The `DB_ENV->set_intermediate_dir_mode()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_isalive()

```
#include <db.h>

int
DB_ENV->set_isalive(DB_ENV *dbenv, int (*is_alive)(DB_ENV *dbenv,
    pid_t pid, db_threadid_t tid, u_int32_t flags));
```

Declare a function that returns if a thread of control (either a true thread or a process) is still running. The `DB_ENV->set_isalive()` method supports the DB_ENV->failchk() (page 206) method. For more information, see Architecting Data Store and Concurrent Data Store applications, and Architecting Transactional Data Store applications, both in the *Berkeley DB Programmer's Reference Guide*.

The `DB_ENV->set_isalive()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_isalive()` method may be called at any time during the life of the application.

The `DB_ENV->set_isalive()` method returns a non-zero error value on failure and 0 on success.

Parameters

is_alive

The **is_alive** parameter is a function which returns non-zero if the thread of control, identified by the **pid** and **tid** arguments, is still running. The function takes four arguments:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle, allowing application access to the application-private fields of that object.

- **pid**

  The **pid** parameter is a process ID returned by the function specified to the DB_ENV->set_thread_id() (page 278) method.

- **tid**

  The **tid** parameter is a thread ID returned by the function specified to the DB_ENV->set_thread_id() (page 278) method.

- **flags**

  The **flags** parameter must be set to 0 or the following value:

  - `DB_MUTEX_PROCESS_ONLY`

    Return only if the process is alive, the thread ID should be ignored.

Errors

The `DB_ENV->set_isalive()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_msgcall()

```
#include <db.h>

void
DB_ENV->set_msgcall(DB_ENV *dbenv,
    void (*db_msgcall_fcn)(const DB_ENV *dbenv, const char *msg));
```

There are interfaces in the Berkeley DB library which either directly output informational messages or statistical information, or configure the library to output such messages when performing other operations, for example, DB_ENV->set_verbose() (page 286) and DB_ENV->stat_print() (page 289).

The `DB_ENV->set_msgcall()` and DB->set_msgcall() (page 116) methods are used to pass these messages to the application, and Berkeley DB will call **db_msgcall_fcn** with each message. It is up to the **db_msgcall_fcn** function to display the message in an appropriate manner.

Setting **db_msgcall_fcn** to NULL unconfigures the callback interface.

Alternatively, you can use the DB->set_msgfile() (page 118) or DB->set_msgfile() (page 273) methods to display the messages via a C library FILE *.

The `DB_ENV->set_msgcall()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_msgcall()` method may be called at any time during the life of the application.

Note

Berkeley DB is not re-entrant. Callback functions should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

db_msgcall_fcn

The **db_msgcall_fcn** parameter is the application-specified message reporting function. The function takes two parameters:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment.

- `msg`

  The **msg** parameter is the message string.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_msgfile()

```
#include <db.h>

void
DB_ENV->set_msgfile(DB_ENV *dbenv, FILE *msgfile);
```

There are interfaces in the Berkeley DB library which either directly output informational messages or statistical information, or configure the library to output such messages when performing other operations, for example, DB_ENV->set_verbose() (page 286) and DB_ENV->stat_print() (page 289).

The DB_ENV->set_msgfile() and DB->set_msgfile() (page 118) methods are used to display these messages for the application. In this case the message will include a trailing <newline> character.

Setting **msgfile** to NULL unconfigures the interface.

Alternatively, you can use the DB_ENV->set_msgcall() (page 271) or DB->set_msgcall() (page 116) methods to capture the additional error information in a way that does not use C library FILE *'s.

The DB_ENV->set_msgfile() method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The DB_ENV->set_msgfile() method may be called at any time during the life of the application.

Parameters

msgfile

The **msgfile** parameter is a C library FILE * to be used for displaying messages.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_shm_key()

```
#include <db.h>

int
DB_ENV->set_shm_key(DB_ENV *dbenv, long shm_key);
```

Specify a base segment ID for Berkeley DB environment shared memory regions created in system memory on VxWorks or systems supporting X/Open-style shared memory interfaces; for example, UNIX systems supporting **shmget**(2) and related System V IPC interfaces.

This base segment ID will be used when Berkeley DB shared memory regions are first created. It will be incremented a small integer value each time a new shared memory region is created; that is, if the base ID is 35, the first shared memory region created will have a segment ID of 35, and the next one will have a segment ID between 36 and 40 or so. A Berkeley DB environment always creates a master shared memory region; an additional shared memory region for each of the subsystems supported by the environment (Locking, Logging, Memory Pool and Transaction); plus an additional shared memory region for each additional memory pool cache that is supported. Already existing regions with the same segment IDs will be removed. See Shared Memory Regions for more information.

The intent behind this method is two-fold: without it, applications have no way to ensure that two Berkeley DB applications don't attempt to use the same segment IDs when creating different Berkeley DB environments. In addition, by using the same segment IDs each time the environment is created, previously created segments will be removed, and the set of segments on the system will not grow without bound.

The database environment's base segment ID may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_shm_key", one or more whitespace characters, and the ID. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_shm_key()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_shm_key()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_shm_key()` must be consistent with the existing environment or corruption can occur.

The `DB_ENV->set_shm_key()` method returns a non-zero error value on failure and 0 on success.

Parameters

shm_key

The **shm_key** parameter is the base segment ID for the database environment.

Errors

The `DB_ENV->set_shm_key()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_thread_count()

```
#include <db.h>

int
DB_ENV->set_thread_count(DB_ENV *dbenv, u_int32_t count);
```

Declare an approximate number of threads in the database environment. The DB_ENV->set_thread_count() method must be called prior to opening the database environment if the DB_ENV->failchk() (page 206) method will be used. The DB_ENV->set_thread_count() method does not set the maximum number of threads but is used to determine memory sizing and the thread control block reclamation policy.

If a process has not configured an **is_alive** function from the DB_ENV->set_isalive() (page 269) method, and then attempts to join a database environment configured for failure checking with the DB_ENV->failchk() (page 206), DB_ENV->set_thread_id() (page 278), DB_ENV->set_isalive() (page 269) and DB_ENV->set_thread_count() methods, the program may be unable to allocate a thread control block and fail to join the environment. **This is true of the standalone Berkeley DB utility programs.** To avoid problems when using the standalone Berkeley DB utility programs with environments configured for failure checking, incorporate the utility's functionality directly in the application, or call the DB_ENV->failchk() (page 206) method before running the utility.

The database environment's thread count may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_thread_count", one or more whitespace characters, and the thread count. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DB_ENV->set_thread_count() method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The DB_ENV->set_thread_count() method may not be called after the DB_ENV->open() (page 231) method is called.

The DB_ENV->set_thread_count() method returns a non-zero error value on failure and 0 on success.

Parameters

count

The **count** parameter is an approximate thread count for the database environment.

Errors

The DB_ENV->set_thread_count() method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_thread_id()

```
#include <db.h>

int
DB_ENV->set_thread_id(DB_ENV *dbenv,
    void (*thread_id)(DB_ENV *dbenv, pid_t *pid, db_threadid_t *tid));
```

Declare a function that returns a unique identifier pair for the current thread of control. The `DB_ENV->set_thread_id()` method supports the DB_ENV->failchk() (page 206) method. For more information, see Architecting Data Store and Concurrent Data Store applications , and Architecting Transactional Data Store applications , both in the *Berkeley DB Programmer's Reference Guide*.

The `DB_ENV->set_thread_id()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_thread_id()` method may be called at any time during the life of the application.

The `DB_ENV->set_thread_id()` method returns a non-zero error value on failure and 0 on success.

Parameters

thread_id

The **thread_id** parameter is a function which returns a unique identifier pair for a thread of control in a Berkeley DB application. The function takes three arguments:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle, allowing application access to the application-private fields of that object.

- **pid**

  The **pid** points to a memory location of type `pid_t`, or NULL. The process ID of the current thread of control may be returned in this memory location, if it is not NULL.

- **tid**

  The **tid** points to a memory location of type `db_threadid_t`, or NULL. The thread ID of the current thread of control may be returned in this memory location, if it is not NULL.

Errors

The `DB_ENV->set_thread_id()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Assigning Thread IDs

The standard system library calls to return process and thread IDs are often sufficient for this purpose (for example, `getpid()` and `pthread_self()` on POSIX systems or GetCurrentThreadID on Windows systems). However, if the Berkeley DB application dynamically creates processes or threads, some care may be necessary in assigning unique IDs. In most threading systems, process and thread IDs are available for re-use as soon as the process or thread exits. If a new process or thread is created between the time of process or thread exit, and the DB_ENV->failchk() (page 206) method is run, it may be possible for DB_ENV->failchk() (page 206) to not detect that a thread of control exited without properly releasing all Berkeley DB resources.

It may be possible to handle this problem by inhibiting process or thread creation between thread of control exit and calling the DB_ENV->failchk() (page 206) method. Alternatively, the **thread_id** function must be constructed to not re-use **pid/tid** pairs. For example, in a single process application, the returned process ID might be used as an incremental counter, with the returned thread ID set to the actual thread ID. Obviously, the **is_alive** function specified to the DB_ENV->set_isalive() (page 269) method must be compatible with any **thread_id** function specified to `DB_ENV->set_thread_id()`.

The db_threadid_t type is configured to be the same type as a standard thread identifier, in Berkeley DB configurations where this type is known (for example, systems supporting pthread_t or thread_t, or DWORD on Windows). If the Berkeley DB configuration process is unable to determine the type of a standard thread identifier, the db_thread_t type is set to uintmax_t (or the largest available unsigned integral type, on systems lacking the uintmax_t type). Applications running on systems lacking a detectable standard thread type, and which are also using thread APIs where a thread identifier is not an integral value and so will not fit into the configured db_threadid_t type, must either translate between the db_threadid_t type and the thread identifier (mapping the thread identifier to a unique identifier of the appropriate size), or modify the Berkeley DB sources to use an appropriate db_threadid_t type. Note: we do not currently know of any systems where this is necessary. If your application has to solve this problem, please contact our support group and let us know.

If no **thread_id** function is specified by the application, the Berkeley DB library will identify threads of control by using the `taskIdSelf()` call on VxWorks, the `getpid()` and `GetCurrentThreadID()` calls on Windows, the `getpid()` and `pthread_self()` calls when the Berkeley DB library has been configured for POSIX pthreads or Solaris LWP threads, the `getpid()` and `thr_self()` calls when the Berkeley DB library has been configured for UI threads, and otherwise `getpid()`.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_thread_id_string()

```
#include <db.h>

int
DB_ENV->set_thread_id_string(DB_ENV *dbenv,
    char *(*thread_id_string)(DB_ENV *dbenv,
    pid_t pid, db_threadid_t tid, char *buf));
```

Declare a function that formats a process ID and thread ID identifier pair for display into a caller-supplied buffer. The function must return a reference to the caller-specified buffer. The `DB_ENV->set_thread_id_string()` method supports the DB_ENV->set_thread_id() (page 278) method.

The `DB_ENV->set_thread_id_string()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_thread_id_string()` method may be called at any time during the life of the application.

The `DB_ENV->set_thread_id_string()` method returns a non-zero error value on failure and 0 on success.

Parameters

thread_id_string

The **thread_id_string** parameter is a function which returns a buffer in which is an identifier pair formatted for display. The function takes four arguments:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment handle, allowing application access to the application-private fields of that object.

- `pid`

  The **pid** argument is a process ID.

- `tid`

  The **tid** argument is a thread ID.

- `buf`

  The **buf** argument is character array of at least DB_THREADID_STRLEN bytes in length, into which the identifier pair should be formatted.

If no **thread_id_string** function is specified, the default routine displays the identifier pair as "pid/tid", that is, the process ID represented as an unsigned integer value, a slash ('/') character, then the thread ID represented as an unsigned integer value.

Errors

The DB_ENV->set_thread_id_string() method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_timeout()

```
#include <db.h>

int
DB_ENV->set_timeout(DB_ENV *dbenv, db_timeout_t timeout,
    u_int32_t flags);
```

The `DB_ENV->set_timeout()` method sets timeout values for locks or transactions in the database environment, and the wait time for a process to exit the environment when DB_REGISTER recovery is needed.

DB_SET_LOCK_TIMEOUT and DB_SET_TXN_TIMEOUT timeouts are checked whenever a thread of control blocks on a lock or when deadlock detection is performed. In the case of DB_SET_LOCK_TIMEOUT, the lock is one requested explicitly through the Lock subsystem interfaces. In the case of DB_SET_TXN_TIMEOUT, the lock is one requested on behalf of a transaction. In either case, it may be a lock requested by the database access methods underlying the application. These timeouts are only checked when the lock request first blocks or when deadlock detection is performed, the accuracy of the timeout depends on how often deadlock detection is performed.

Lock and transaction timeout values specified for the database environment may be overridden on a per-lock or per-transaction basis. See DB_ENV->lock_vec() (page 326) and DB_TXN->set_timeout() (page 566) for more information.

The `DB_ENV->set_timeout()` method may be called at any time during the life of the application.

The `DB_ENV->set_timeout()` method returns a non-zero error value on failure and 0 on success.

Parameters

timeout

The **timeout** parameter is the timeout value. It must be specified as an unsigned 32-bit number of microseconds, limiting the maximum timeout to roughly 71 minutes.

flags

The **flags** parameter must be set to one of the following values:

- `DB_SET_LOCK_TIMEOUT`

  Set the timeout value for locks in this database environment.

  The database environment's lock timeout value may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lock_timeout", one or more whitespace characters, and the lock timeout value. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

This flag configures a database environment, not only operations performed using the specified DB_ENV handle.

- `DB_SET_REG_TIMEOUT`

  Set the timeout value on how long to wait for processes to exit the environment before recovery is started when the DB_ENV->open() (page 231) method was called with the DB_REGISTER flag and recovery must be performed.

  This wait timeout value may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_reg_timeout", one or more whitespace characters, and the wait timeout value. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

  This flag configures operations performed using the specified DB_ENV handle.

- `DB_SET_TXN_TIMEOUT`

  Set the timeout value for transactions in this database environment.

  The database environment's transaction timeout value may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_txn_timeout", one or more whitespace characters, and the transaction timeout value. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

  This flag configures a database environment, not only operations performed using the specified DB_ENV handle.

Errors

The `DB_ENV->set_timeout()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_tmp_dir()

```
#include <db.h>

int
DB_ENV->set_tmp_dir(DB_ENV *dbenv, const char *dir);
```

Specify the path of a directory to be used as the location of temporary files. The files created to back in-memory access method databases will be created relative to this path. These temporary files can be quite large, depending on the size of the database.

If no directories are specified, the following alternatives are checked in the specified order. The first existing directory path is used for all temporary files.

1. The value of the environment variable **TMPDIR**.
2. The value of the environment variable **TEMP**.
3. The value of the environment variable **TMP**.
4. The value of the environment variable **TempFolder**.
5. The value returned by the **GetTempPath** interface.
6. The directory **/var/tmp**.
7. The directory **/usr/tmp**.
8. The directory **/temp**.
9. The directory **/tmp**.
10. The directory **C:/temp**.
11. The directory **C:/tmp**.

Note

Environment variables are only checked if one of the DB_USE_ENVIRON or DB_USE_ENVIRON_ROOT flags were specified.

Note

The GetTempPath interface is only checked on Win/32 platforms.

The database environment's temporary file directory may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_tmp_dir", one or more whitespace characters, and the directory name. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_tmp_dir()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_tmp_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** parameter is the directory to be used to store temporary files.

When using a Unicode build on Windows (the default), the this argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB_ENV->set_tmp_dir()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_verbose()

```
#include <db.h>

int
DB_ENV->set_verbose(DB_ENV *dbenv, u_int32_t which, int onoff);
```

The `DB_ENV->set_verbose()` method turns specific additional informational and debugging messages in the Berkeley DB message output on and off. To see the additional messages, verbose messages must also be configured for the application. For more information on verbose messages, see the DB_ENV->set_msgfile() (page 273) method.

The database environment's messages may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_verbose", one or more whitespace characters, and the method **which** parameter as a string and optionally one or more whitespace characters, and the string "on" or "off". If the optional string is omitted, the default is "on"; for example, "set_verbose DB_VERB_RECOVERY" or "set_verbose DB_VERB_RECOVERY on". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_verbose()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_verbose()` method may be called at any time during the life of the application.

The `DB_ENV->set_verbose()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter must be set to one of the following values:

- `DB_VERB_DEADLOCK`

  Display additional information when doing deadlock detection.

- `DB_VERB_FILEOPS`

  Display additional information when performing filesystem operations such as open, close or rename. May not be available on all platforms.

- `DB_VERB_FILEOPS_ALL`

  Display additional information when performing all filesystem operations, including read and write. May not be available on all platforms.

- `DB_VERB_RECOVERY`

Display additional information when performing recovery.

- `DB_VERB_REGISTER`

  Display additional information concerning support for the DB_REGISTER flag to the DB_ENV->open() (page 231) method.

- `DB_VERB_REPLICATION`

  Display all detailed information about replication. This includes the information displayed by all of the other DB_VERB_REP_* and DB_VERB_REPMGR_* values.

- `DB_VERB_REP_ELECT`

  Display detailed information about replication elections.

- `DB_VERB_REP_LEASE`

  Display detailed information about replication master leases.

- `DB_VERB_REP_MISC`

  Display detailed information about general replication processing not covered by the other DB_VERB_REP_* values.

- `DB_VERB_REP_MSGS`

  Display detailed information about replication message processing.

- `DB_VERB_REP_SYNC`

  Display detailed information about replication client synchronization.

- `DB_VERB_REP_SYSTEM`

  Saves replication system information to a system-owned file. This value is on by default.

- `DB_VERB_REPMGR_CONNFAIL`

  Display detailed information about Replication Manager connection failures.

- `DB_VERB_REPMGR_MISC`

  Display detailed information about general Replication Manager processing.

- `DB_VERB_WAITSFOR`

  Display the waits-for table when doing deadlock detection.

onoff

If the **onoff** parameter is set to non-zero, the additional messages are output.

Errors

The DB_ENV->set_verbose() method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->stat_print()

```
#include <db.h>

int
DB_ENV->stat_print(DB_ENV *dbenv, u_int32_t flags);
```

The `DB_ENV->stat_print()` method displays the default statistical information. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

- `DB_STAT_SUBSYSTEM`

  Display information for all configured subsystems.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

db_strerror

```
#include <db.h>

char *
db_strerror(int error);
```

The `db_strerror()` method returns an error message string corresponding to the error number **error** parameter.

This function is a superset of the ANSI C X3.159-1989 (ANSI C) **strerror**(3) function. If the error number **error** is greater than or equal to 0, then the string returned by the system function **strerror**(3) is returned. If the error number is less than 0, an error string appropriate to the corresponding Berkeley DB library error is returned. See Error returns to applications for more information.

Parameters

error

The **error** parameter is the error number for which an error message string is wanted.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

db_version

```
#include <db.h>

char *
db_version(int *major, int *minor, int *patch);
```

The `db_version()` method returns a pointer to a string, suitable for display, containing Berkeley DB version information. For a method that returns this information as well as Oracle release numbers, see db_full_version (page 210).

Parameters

major

If **major** is non-NULL, the major version of the Berkeley DB release is copied to the memory to which it refers.

minor

If **minor** is non-NULL, the minor version of the Berkeley DB release is copied to the memory to which it refers.

patch

If **patch** is non-NULL, the patch version of the Berkeley DB release is copied to the memory to which it refers.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

Chapter 6. The DB_LOCK Handle

```
#include <db.h>

typedef struct __db_lock_u DB_LOCK;
```

The locking interfaces for the Berkeley DB database environment are methods of the DB_ENV handle. The DB_LOCK object is the handle for a single lock, and has no methods of its own.

Locking Subsystem and Related Methods

Locking Subsystem and Related Methods	Description
DB_ENV->lock_detect()	Perform deadlock detection
DB_ENV->lock_get()	Acquire a lock
DB_ENV->lock_id()	Acquire a locker ID
DB_ENV->lock_id_free()	Release a locker ID
DB_ENV->lock_put()	Release a lock
DB_ENV->lock_stat()	Return lock subsystem statistics
DB_ENV->lock_stat_print()	Print lock subsystem statistics
DB_ENV->lock_vec()	Acquire/release locks
DB_ENV->cdsgroup_begin()	Get a locker ID in Berkeley DB Concurrent Data Store
Locking Subsystem Configuration	
DB_ENV->set_timeout(), DB_ENV->get_timeout()	Set/get lock and transaction timeout
DB_ENV->set_lk_conflicts(), DB_ENV->get_lk_conflicts()	Set/get lock conflicts matrix
DB_ENV->set_lk_detect(), DB_ENV->get_lk_detect()	Set/get automatic deadlock detection
DB_ENV->set_lk_max_lockers(), DB_ENV->get_lk_max_lockers()	Set/get maximum number of lockers
DB_ENV->set_lk_max_locks(), DB_ENV->get_lk_max_locks()	Set/get maximum number of locks
DB_ENV->set_lk_max_objects(), DB_ENV->get_lk_max_objects()	Set/get maximum number of lock objects
DB_ENV->set_lk_partitions(), DB_ENV->get_lk_partitions()	Set/get number of lock partitions
DB_ENV->set_lk_priority(), DB_ENV->get_lk_priority()	Set/get a locker's deadlock priority

DB_ENV->get_lk_conflicts()

```
#include <db.h>

int
DB_ENV->get_lk_conflicts(DB_ENV *dbenv,
    const u_int8_t **lk_conflictsp, int *lk_modesp);
```

The DB_ENV->get_lk_conflicts() method returns the current conflicts array. You can specify a conflicts array using DB_ENV->set_lk_conflicts() (page 301)

The DB_ENV->get_lk_conflicts() method may be called at any time during the life of the application.

The DB_ENV->get_lk_conflicts() method returns a non-zero error value on failure and 0 on success.

Parameters

lk_conflictsp

The **lk_conflictsp** parameter references memory into which a pointer to the current conflicts array is copied.

lk_modesp

The **lk_modesp** parameter references memory into which the size of the current conflicts array is copied.

Errors

The DB_ENV->get_lk_conflicts() method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_conflicts() (page 301)

DB_ENV->get_lk_detect()

```
#include <db.h>

int
DB_ENV->get_lk_detect(DB_ENV *dbenv, u_int32_t *lk_detectp);
```

The `DB_ENV->get_lk_detect()` method returns the deadlock detector configuration. You can manage this using the DB_ENV->set_lk_detect() (page 303) method.

The `DB_ENV->get_lk_detect()` method may be called at any time during the life of the application.

The `DB_ENV->get_lk_detect()` method returns a non-zero error value on failure and 0 on success.

Parameters

lk_detectp

The `DB_ENV->get_lk_detect()` method returns the deadlock detector configuration in **lk_detectp**.

Errors

The `DB_ENV->get_lk_detect()` method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_detect() (page 303)

DB_ENV->get_lk_max_lockers()

```
#include <db.h>

int
DB_ENV->get_lk_max_lockers(DB_ENV *dbenv, u_int32_t *lk_maxp);
```

The `DB_ENV->get_lk_max_lockers()` method returns the maximum number of potential lockers. You can configure this using the DB_ENV->set_lk_max_lockers() (page 305) method.

The `DB_ENV->get_lk_max_lockers()` method may be called at any time during the life of the application.

The `DB_ENV->get_lk_max_lockers()` method returns a non-zero error value on failure and 0 on success.

Parameters

lk_maxp

The `DB_ENV->get_lk_max_lockers()` method returns the maximum number of lockers in **lk_maxp**.

Errors

The `DB_ENV->get_lk_max_lockers()` method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_max_lockers() (page 305)

DB_ENV->get_lk_max_locks()

```
#include <db.h>

int
DB_ENV->get_lk_max_locks(DB_ENV *dbenv, u_int32_t *lk_maxp);
```

The `DB_ENV->get_lk_max_locks()` method returns the maximum number of potential locks. You can configure this using the DB_ENV->set_lk_max_locks() (page 306) method.

The `DB_ENV->get_lk_max_locks()` method may be called at any time during the life of the application.

The `DB_ENV->get_lk_max_locks()` method returns a non-zero error value on failure and 0 on success.

Parameters

lk_maxp

The `DB_ENV->get_lk_max_locks()` method returns the maximum number of locks in **lk_maxp**.

Errors

The `DB_ENV->get_lk_max_locks()` method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_max_locks() (page 306)

DB_ENV->get_lk_max_objects()

```
#include <db.h>

int
DB_ENV->get_lk_max_objects(DB_ENV *dbenv, u_int32_t *lk_maxp);
```

The `DB_ENV->get_lk_max_objects()` method returns the maximum number of locked objects. You can configure this using the DB_ENV->set_lk_max_objects() (page 307) method.

The `DB_ENV->get_lk_max_objects()` method may be called at any time during the life of the application.

The `DB_ENV->get_lk_max_objects()` method returns a non-zero error value on failure and 0 on success.

Parameters

lk_maxp

The `DB_ENV->get_lk_max_objects()` method returns the maximum number of potentially locked objects in **lk_maxp**.

Errors

The `DB_ENV->get_lk_max_objects()` method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_max_objects() (page 307)

DB_ENV->get_lk_partitions()

```
#include <db.h>

int
DB_ENV->get_lk_partitions(DB_ENV *dbenv, u_int32_t *lk_partitions);
```

The `DB_ENV->get_lk_partitions()` method returns the number of lock table partitions used in the Berkeley DB environment. You can configure this using the DB_ENV->set_lk_partitions() (page 308) method.

The `DB_ENV->get_lk_partitions()` method may be called at any time during the life of the application.

The `DB_ENV->get_lk_partitions()` method returns a non-zero error value on failure and 0 on success.

Parameters

lk_partitions

The `DB_ENV->get_lk_partitions()` method returns the number of partitions in **lk_partitions**.

Errors

The `DB_ENV->get_lk_partitions()` method may fail and return one of the following non-zero errors:

EINVAL

The method was called on an environment which had been opened without being configured for locking.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_partitions() (page 308)

DB_ENV->get_lk_priority()

```
#include <db.h>

int
DB_ENV->get_lk_priority(DB_ENV *dbenv,
    u_int32_t lockerid, u_int32_t *priority);
```

Get the deadlock priority for the given locker.

Parameters

lockerid

The **lockerid** parameter represents a locker returned by `envM;lock_id()`.

priority

Upon return, the **priority** parameter will point to a value between 0 and 2^32-1.

Errors

The `DB_ENV->get_lk_priority()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293), DB_ENV->set_lk_priority() (page 310)

DB_ENV->set_lk_conflicts()

```
#include <db.h>

int
DB_ENV->set_lk_conflicts(DB_ENV *dbenv,
    u_int8_t *conflicts, int nmodes);
```

Set the locking conflicts matrix.

If `DB_ENV->set_lk_conflicts()` is never called, a standard conflicts array is used; see Standard Lock Modes for more information.

The `DB_ENV->set_lk_conflicts()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_conflicts()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lk_conflicts()` will be ignored.

The `DB_ENV->set_lk_conflicts()` method returns a non-zero error value on failure and 0 on success.

Parameters

conflicts

The **conflicts** parameter is the new locking conflicts matrix. The **conflicts** parameter is an **nmodes** by **nmodes** array. A non-0 value for the array element indicates that requested_mode and held_mode conflict:

```
        conflicts[requested_mode][held_mode]
```

The *not-granted* mode must be represented by 0.

nmodes

The **nmodes** parameter is the size of the lock conflicts matrix.

Errors

The `DB_ENV->set_lk_conflicts()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

ENOMEM

The conflicts array could not be copied.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_detect()

```
#include <db.h>

int
DB_ENV->set_lk_detect(DB_ENV *dbenv, u_int32_t detect);
```

Set if the deadlock detector is to be run whenever a lock conflict occurs, and specify what lock request(s) should be rejected. As transactions acquire locks on behalf of a single locker ID, rejecting a lock request associated with a transaction normally requires the transaction be aborted.

The database environment's deadlock detector configuration may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lk_detect", one or more whitespace characters, and the method **detect** parameter as a string; for example, "set_lk_detect DB_LOCK_OLDEST". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lk_detect()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_detect()` method may be called at any time during the life of the application.

The `DB_ENV->set_lk_detect()` method returns a non-zero error value on failure and 0 on success.

Parameters

detect

The **detect** parameter configures the deadlock detector. The deadlock detector will reject the lock request with the lowest priority. If multiple lock requests have the lowest priority, then the **detect** parameter is used to select which of those lock requests to reject. The specified value must be one of the following list:

- `DB_LOCK_DEFAULT`

  Use whatever lock policy was specified when the database environment was created. If no lock policy has yet been specified, set the lock policy to DB_LOCK_RANDOM.

- `DB_LOCK_EXPIRE`

  Reject lock requests which have timed out. No other deadlock detection is performed.

- `DB_LOCK_MAXLOCKS`

  Reject the lock request for the locker ID with the most locks.

- `DB_LOCK_MAXWRITE`

Reject the lock request for the locker ID with the most write locks.

- `DB_LOCK_MINLOCKS`

  Reject the lock request for the locker ID with the fewest locks.

- `DB_LOCK_MINWRITE`

  Reject the lock request for the locker ID with the fewest write locks.

- `DB_LOCK_OLDEST`

  Reject the lock request for the locker ID with the oldest lock.

- `DB_LOCK_RANDOM`

  Reject the lock request for a random locker ID.

- `DB_LOCK_YOUNGEST`

  Reject the lock request for the locker ID with the youngest lock.

Errors

The `DB_ENV->set_lk_detect()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_max_lockers()

```
#include <db.h>

int
DB_ENV->set_lk_max_lockers(DB_ENV *dbenv, u_int32_t max);
```

Sets the maximum number of locking entities supported by the Berkeley DB environment. This value is used by DB_ENV->open() (page 231) to estimate how much space to allocate for various lock-table data structures. The default value is 1000 lockers. For specific information on configuring the size of the lock subsystem, see Configuring locking: sizing the system.

The database environment's maximum number of lockers may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lk_max_lockers", one or more whitespace characters, and the number of lockers. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lk_max_lockers()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_max_lockers()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lk_max_lockers()` will be ignored.

The `DB_ENV->set_lk_max_lockers()` method returns a non-zero error value on failure and 0 on success.

Parameters

max

The **max** parameter is the maximum number simultaneous locking entities supported by the Berkeley DB environment.

Errors

The `DB_ENV->set_lk_max_lockers()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_max_locks()

```
#include <db.h>

int
DB_ENV->set_lk_max_locks(DB_ENV *dbenv, u_int32_t max);
```

Set the maximum number of locks supported by the Berkeley DB environment. This value is used by DB_ENV->open() (page 231) to estimate how much space to allocate for various lock-table data structures. The default value is 1000 locks. For specific information on configuring the size of the lock subsystem, see Configuring locking: sizing the system.

The database environment's maximum number of locks may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lk_max_locks", one or more whitespace characters, and the number of locks. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lk_max_locks()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_max_locks()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lk_max_locks()` will be ignored.

The `DB_ENV->set_lk_max_locks()` method returns a non-zero error value on failure and 0 on success.

Parameters

max

The **max** parameter is the maximum number of locks supported by the Berkeley DB environment.

Errors

The `DB_ENV->set_lk_max_locks()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_max_objects()

```
#include <db.h>

int
DB_ENV->set_lk_max_objects(DB_ENV *dbenv, u_int32_t max);
```

Set the maximum number of locked objects supported by the Berkeley DB environment. This value is used by DB_ENV->open() (page 231) to estimate how much space to allocate for various lock-table data structures. The default value is 1000 objects. For specific information on configuring the size of the lock subsystem, see Configuring locking: sizing the system.

The database environment's maximum number of objects may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lk_max_objects", one or more whitespace characters, and the number of objects. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lk_max_objects()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_max_objects()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lk_max_objects()` will be ignored.

The `DB_ENV->set_lk_max_objects()` method returns a non-zero error value on failure and 0 on success.

Parameters

max

The **max** parameter is the maximum number of locked objects supported by the Berkeley DB environment.

Errors

The `DB_ENV->set_lk_max_objects()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_partitions()

```
#include <db.h>

int
DB_ENV->set_lk_partitions(DB_ENV *dbenv, u_int32_t partitions);
```

Set the number of lock table partitions in the Berkeley DB environment. The default value is 10 times the number of CPUs on the system if there is more than one CPU. Increasing the number of partitions can provide for greater throughput on a system with multiple CPUs and more than one thread contending for the lock manager. On single processor systems more than one partition may increase the overhead of the lock manager. Systems often report threading contexts as CPUs. If your system does this, set the number of partitions to 1 to get optimal performance.

The database environment's number of partitions may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lk_partitions", one or more whitespace characters, and the number of partitions. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lk_partitions()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lk_partitions()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lk_partitions()` will be ignored.

The `DB_ENV->set_lk_partitions()` method returns a non-zero error value on failure and 0 on success.

Parameters

partitions

The **partitions** parameter is the number of partitions to be configured in the Berkeley DB environment.

Errors

The `DB_ENV->set_lk_partitions()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->set_lk_priority()

```
#include <db.h>

int
DB_ENV->set_lk_priority(DB_ENV *dbenv,
    u_int32_t lockerid, u_int32_t priority);
```

Set the priority of the given locker. This value is used when resolving deadlocks, the deadlock resolution algorithm will reject a lock request from a locker with a lower priority before a request from a locker with a higher priority.

By default, all lockers are created with a priority of 100.

The `DB_ENV->set_lk_priority()` method may be called at any time during the life of the application.

The `DB_ENV->set_lk_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

lockerid

The **lockerid** parameter represents a locker returned by `DB_ENV->lock_id()`.

priority

The **priority** parameter must be a value between 0 and 2^32-1.

Errors

The `DB_ENV->set_lk_priority()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_detect()

```
#include <db.h>

int
DB_ENV->lock_detect(DB_ENV *env,
    u_int32_t flags, u_int32_t atype, int *rejected);
```

The `DB_ENV->lock_detect()` method runs one iteration of the deadlock detector. The deadlock detector traverses the lock table and marks one of the participating lock requesters for rejection in each deadlock it finds.

The `DB_ENV->lock_detect()` method is the underlying method used by the db_deadlock utility. See the db_deadlock utility source code for an example of using `DB_ENV->lock_detect()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The `DB_ENV->lock_detect()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

atype

The **atype** parameter specifies which lock request(s) to reject. The deadlock detector will reject the lock request with the lowest priority. If multiple lock requests have the lowest priority, then the **atype** parameter is used to select which of those lock requests to reject. It must be set to one of the following list:

- `DB_LOCK_DEFAULT`

  Use the default lock policy, which is DB_LOCK_RANDOM.

- `DB_LOCK_EXPIRE`

  Reject lock requests which have timed out. No other deadlock detection is performed.

- `DB_LOCK_MAXLOCKS`

  Reject the lock request for the locker ID with the most locks.

- `DB_LOCK_MAXWRITE`

  Reject the lock request for the locker ID with the most write locks.

- `DB_LOCK_MINLOCKS`

  Reject the lock request for the locker ID with the fewest locks.

- `DB_LOCK_MINWRITE`

  Reject the lock request for the locker ID with the fewest write locks.

- `DB_LOCK_OLDEST`

  Reject the lock request for the locker ID with the oldest lock.

- `DB_LOCK_RANDOM`

  Reject the lock request for a random locker ID.

- `DB_LOCK_YOUNGEST`

  Reject the lock request for the locker ID with the youngest lock.

rejected

If the **rejected** parameter is non-NULL, the memory location to which it refers will be set to the number of lock requests that were rejected.

Errors

The `DB_ENV->lock_detect()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_get()

```
#include <db.h>

int
DB_ENV->lock_get(DB_ENV *env, u_int32_t locker,
    u_int32_t flags, const DBT *object,
    const db_lockmode_t lock_mode, DB_LOCK *lock);
```

The `DB_ENV->lock_get()` method acquires a lock from the lock table, returning information about it in the **lock** parameter.

The `DB_ENV->lock_get()` method returns a non-zero error value on failure and 0 on success.

Parameters

locker

The **locker** parameter is an unsigned 32-bit integer quantity. It represents the entity requesting the lock.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_LOCK_NOWAIT`

  If a lock cannot be granted because the requested lock conflicts with an existing lock, return DB_LOCK_NOTGRANTED immediately instead of waiting for the lock to become available.

object

The **object** parameter is an untyped byte string that specifies the object to be locked. Applications using the locking subsystem directly while also doing locking via the Berkeley DB access methods must take care not to inadvertently lock objects that happen to be equal to the unique file IDs used to lock files. See Access method locking conventions in the *Berkeley DB Programmer's Reference Guide* for more information.

lock_mode

The **lock_mode** parameter is used as an index into the environment's lock conflict matrix. When using the default lock conflict matrix, **lock_mode** must be set to one of the following values:

- `DB_LOCK_READ`

  read (shared)

- `DB_LOCK_WRITE`

  write (exclusive)

- `DB_LOCK_IWRITE`

  intention to write (shared)

- `DB_LOCK_IREAD`

  intention to read (shared)

- `DB_LOCK_IWR`

  intention to read and write (shared)

See DB_ENV->set_lk_conflicts() (page 301) and Standard Lock Modes for more information on the lock conflict matrix.

lock

The `DB_ENV->lock_get()` method returns the lock information in **lock**.

Errors

The `DB_ENV->lock_get()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_LOCK_NOTGRANTED

The DB_LOCK_NOWAIT flag or lock timers were configured and the lock could not be granted before the wait-time expired.

EINVAL

An invalid flag value or parameter was specified.

EINVAL

The method was called on an environment which had been opened without being configured for locking.

ENOMEM

The maximum number of locks has been reached.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_id()

```
#include <db.h>

int
DB_ENV->lock_id(DB_ENV *env, u_int32_t *idp);
```

The `DB_ENV->lock_id()` method copies a locker ID, which is guaranteed to be unique in the environment's lock table, into the memory location to which **idp** refers.

The DB_ENV->lock_id_free() (page 317) method should be called to return the locker ID to the Berkeley DB library when it is no longer needed.

The `DB_ENV->lock_id()` method returns a non-zero error value on failure and 0 on success.

Parameters

idp

The **idp** parameter references memory into which the allocated locker ID is copied.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_id_free()

```
#include <db.h>

int
DB_ENV->lock_id_free(DB_ENV *env, u_int32_t id);
```

The `DB_ENV->lock_id_free()` method frees a locker ID allocated by the DB_ENV->lock_id() (page 316) method.

The `DB_ENV->lock_id_free()` method returns a non-zero error value on failure and 0 on success.

Parameters

id

The **id** parameter is the locker id to be freed.

Errors

The `DB_ENV->lock_id_free()` method may fail and return one of the following non-zero errors:

EINVAL

If the locker ID is invalid or locks are still held by this locker ID; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_put()

```
#include <db.h>

int
DB_ENV->lock_put(DB_ENV *env, DB_LOCK *lock);
```

The `DB_ENV->lock_put()` method releases **lock**.

The `DB_ENV->lock_put()` method returns a non-zero error value on failure and 0 on success.

Parameters

lock

The **lock** parameter is the lock to be released.

Errors

The `DB_ENV->lock_put()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_stat()

```
#include <db.h>

int
DB_ENV->lock_stat(DB_ENV *env, DB_LOCK_STAT **statp, u_int32_t flags);
```

The `DB_ENV->lock_stat()` method returns the locking subsystem statistics.

The `DB_ENV->lock_stat()` method creates a statistical structure of type DB_LOCK_STAT and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following DB_LOCK_STAT fields will be filled in:

- **u_int32_t st_id;**

  The last allocated locker ID.

- **u_int32_t st_cur_maxid;**

  The current maximum unused locker ID.

- **int st_nmodes;**

  The number of lock modes.

- **u_int32_t st_maxlocks;**

  The maximum number of locks possible.

- **u_int32_t st_maxlockers;**

  The maximum number of lockers possible.

- **u_int32_t st_maxobjects;**

  The maximum number of lock objects possible.

- **u_int32_t st_partitions;**

  The number of lock table partitions.

- **u_int32_t st_nlocks;**

  The number of current locks.

- **u_int32_t st_maxnlocks;**

The maximum number of locks at any one time. Note that if there is more than one partition, this is the sum of the maximum across all partitions.

- **u_int32_t st_maxhlocks;**

  The maximum number of locks in any hash bucket at any one time.

- **uintmax_t st_locksteals;**

  The maximum number of locks stolen by an empty partition.

- **uintmax_t st_maxlsteals;**

  The maximum number of lock steals for any one partition.

- **u_int32_t st_nlockers;**

  The number of current lockers.

- **u_int32_t st_maxnlockers;**

  The maximum number of lockers at any one time.

- **u_int32_t st_nobjects;**

  The number of current lock objects.

- **u_int32_t st_maxnobjects;**

  The maximum number of lock objects at any one time. Note that if there is more than one partition this is the sum of the maximum across all partitions.

- **u_int32_t st_maxhobjects;**

  The maximum number of objects in any hash bucket at any one time.

- **uintmax_t st_objectsteals;**

  The maximum number of objects stolen by an empty partition.

- **uintmax_t st_maxosteals;**

  The maximum number of object steals for any one partition.

- **uintmax_t st_nrequests;**

  The total number of locks requested.

- **uintmax_t st_nreleases;**

  The total number of locks released.

- **uintmax_t st_nupgrade;**

The total number of locks upgraded.

- **uintmax_t st_ndowngrade;**

  The total number of locks downgraded.

- **uintmax_t st_lock_wait;**

  The number of lock requests not immediately available due to conflicts, for which the thread of control waited.

- **uintmax_t st_lock_nowait;**

  The number of lock requests not immediately available due to conflicts, for which the thread of control did not wait.

- **uintmax_t st_ndeadlocks;**

  The number of deadlocks.

- **db_timeout_t st_locktimeout;**

  Lock timeout value.

- **uintmax_t st_nlocktimeouts;**

  The number of lock requests that have timed out.

- **u_int32_t st_txntimeout;**

  Transaction timeout value.

- **uintmax_t st_ntxntimeouts;**

  The number of transactions that have timed out. This value is also a component of **st_ndeadlocks**, the total number of deadlocks detected.

- **uintmax_t st_objs_wait;**

  The number of requests to allocate or deallocate an object for which the thread of control waited.

- **uintmax_t st_objs_nowait;**

  The number of requests to allocate or deallocate an object for which the thread of control did not wait.

- **uintmax_t st_lockers_wait;**

  The number of requests to allocate or deallocate a locker for which the thread of control waited.

- **uintmax_t st_lockers_nowait;**

The number of requests to allocate or deallocate a locker for which the thread of control did not wait.

- **u_int32_t st_hash_len;**

  Maximum length of a lock hash bucket.

- **roff_t st_regsize;**

  The size of the lock region, in bytes.

- **uintmax_t st_part_wait;**

  The number of times that a thread of control was forced to wait before obtaining the lock partition mutex.

- **uintmax_t st_part_nowait;**

  The number of times that a thread of control was able to obtain the lock partition mutex without waiting.

- **uintmax_t st_part_max_wait;**

  The maximum number of times that a thread of control was forced to wait before obtaining any one lock partition mutex.

- **uintmax_t st_part_max_nowait;**

  The number of times that a thread of control was able to obtain any one lock partition mutex without waiting.

- **uintmax_t st_region_wait;**

  The number of times that a thread of control was forced to wait before obtaining the lock region mutex.

- **uintmax_t st_region_nowait;**

  The number of times that a thread of control was able to obtain the lock region mutex without waiting.

The `DB_ENV->lock_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->lock_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->lock_stat()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_stat_print()

```
#include <db.h>

int
DB_ENV->lock_stat_print(DB_ENV *env, u_int32_t flags);
```

The `DB_ENV->lock_stat_print()` method displays the locking subsystem statistical information, as described for the `DB_ENV->lock_stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->lock_stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->lock_stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

- `DB_STAT_LOCK_CONF`

  Display the lock conflict matrix.

- `DB_STAT_LOCK_LOCKERS`

  Display the lockers within hash chains.

- `DB_STAT_LOCK_OBJECTS`

  Display the lock objects within hash chains.

- `DB_STAT_LOCK_PARAMS`

  Display the locking subsystem parameters.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

DB_ENV->lock_vec()

```
#include <db.h>

int
DB_ENV->lock_vec(DB_ENV *env, u_int32_t locker, u_int32_t flags,
    DB_LOCKREQ list[], int nlist, DB_LOCKREQ **elistp);
```

The `DB_ENV->lock_vec()` method atomically obtains and releases one or more locks from the lock table. The `DB_ENV->lock_vec()` method is intended to support acquisition or trading of multiple locks under one lock table semaphore, as is needed for lock coupling or in multigranularity locking for lock escalation.

If any of the requested locks cannot be acquired, or any of the locks to be released cannot be released, the operations before the failing operation are guaranteed to have completed successfully, and `DB_ENV->lock_vec()` returns a non-zero value. In addition, if **elistp** is not NULL, it is set to point to the DB_LOCKREQ entry that was being processed when the error occurred.

Unless otherwise specified, the `DB_ENV->lock_vec()` method returns a non-zero error value on failure and 0 on success.

Parameters

locker

The **locker** parameter is an unsigned 32-bit integer quantity. It represents the entity requesting or releasing the lock.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_LOCK_NOWAIT`

  If a lock cannot be granted because the requested lock conflicts with an existing lock, return DB_LOCK_NOTGRANTED immediately instead of waiting for the lock to become available. In this case, if non-NULL, **elistp** identifies the request that was not granted.

list

The **list** array provided to `DB_ENV->lock_vec()` is typedef'd as DB_LOCKREQ.

To ensure compatibility with future releases of Berkeley DB, all fields of the DB_LOCKREQ structure that are not explicitly set should be initialized to 0 before the first time the structure is used. Do this by declaring the structure external or static, or by calling **memset**(3).

A DB_LOCKREQ structure has at least the following fields:

- `lockop_t op;`

The operation to be performed, which must be set to one of the following values:

- `DB_LOCK_GET`

  Get the lock defined by the values of the **mode** and **obj** structure fields, for the specified **locker**. Upon return from `DB_ENV->lock_vec()`, if the **lock** field is non-NULL, a reference to the acquired lock is stored there. (This reference is invalidated by any call to `DB_ENV->lock_vec()` or DB_ENV->lock_put() (page 318) that releases the lock.)

- `DB_LOCK_GET_TIMEOUT`

  Identical to DB_LOCK_GET except that the value in the **timeout** structure field overrides any previously specified timeout value for this lock. A value of 0 turns off any previously specified timeout.

- `DB_LOCK_PUT`

  The lock to which the **lock** structure field refers is released. The **locker** parameter, and **mode** and **obj** fields are ignored.

- `DB_LOCK_PUT_ALL`

  All locks held by the specified **locker** are released. The **lock**, **mode**, and **obj** structure fields are ignored. Locks acquired in operations performed by the current call to `DB_ENV->lock_vec()` which appear before the DB_LOCK_PUT_ALL operation are released; those acquired in operations appearing after the DB_LOCK_PUT_ALL operation are not released.

- `DB_LOCK_PUT_OBJ`

  All locks held on **obj** are released. The **locker** parameter and the **lock** and **mode** structure fields are ignored. Locks acquired in operations performed by the current call to `DB_ENV->lock_vec()` that appear before the DB_LOCK_PUT_OBJ operation are released; those acquired in operations appearing after the DB_LOCK_PUT_OBJ operation are not released.

- `DB_LOCK_TIMEOUT`

  Cause the specified **locker** to timeout immediately. If the database environment has not configured automatic deadlock detection, the transaction will timeout the next time deadlock detection is performed. As transactions acquire locks on behalf of a single locker ID, timing out the locker ID associated with a transaction will time out the transaction itself.

- `DB_LOCK lock;`

  A lock reference.

- `const lockmode_t mode;`

  The lock mode, used as an index into the environment's lock conflict matrix. When using the default lock conflict matrix, **mode** must be set to one of the following values:

- `DB_LOCK_READ`

  read (shared)

- `DB_LOCK_WRITE`

  write (exclusive)

- `DB_LOCK_IWRITE`

  intention to write (shared)

- `DB_LOCK_IREAD`

  intention to read (shared)

- `DB_LOCK_IWR`

  intention to read and write (shared)

See DB_ENV->set_lk_conflicts() (page 301) and Standard Lock Modes for more information on the lock conflict matrix.

- **const DBT obj;**

  An untyped byte string that specifies the object to be locked or released. Applications using the locking subsystem directly while also doing locking via the Berkeley DB access methods must take care not to inadvertently lock objects that happen to be equal to the unique file IDs used to lock files. See Access method locking conventions in the *Berkeley DB Programmer's Reference Guide* for more information.

- **u_int32_t timeout;**

  The lock timeout value.

nlist

The **nlist** parameter specifies the number of elements in the **list** array.

elistp

If an error occurs, and the **elistp** parameter is non-NULL, it is set to point to the DB_LOCKREQ entry that was being processed when the error occurred.

Errors

The `DB_ENV->lock_vec()` method may fail and return one of the following non-zero errors:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

DB_LOCK_NOTGRANTED

The DB_LOCK_NOWAIT flag or lock timers were configured and the lock could not be granted before the wait-time expired.

EINVAL

An invalid flag value or parameter was specified.

ENOMEM

The maximum number of locks has been reached.

Class

DB_ENV, DB_LOCK

See Also

Locking Subsystem and Related Methods (page 293)

Chapter 7. The DB_LSN Handle

```
#include <db.h>

typedef struct __typedef struct __db_lsn DB_LSN; ;
```

The DB_LSN object is a *log sequence number* which specifies a unique location in a log file. A DB_LSN consists of two unsigned 32-bit integers -- one specifies the log file number, and the other specifies an offset in the log file.

Logging Subsystem and Related Methods

Logging Subsystem and Related Methods	Description
DB_ENV->log_archive()	List log and database files
DB_ENV->log_file()	Map Log Sequence Numbers to log files
DB_ENV->log_flush()	Flush log records
DB_ENV->log_printf()	Append informational message to the log
DB_ENV->log_put()	Write a log record
DB_ENV->log_stat()	Return log subsystem statistics
DB_ENV->log_stat_print()	Print log subsystem statistics
log_compare	Compare two Log Sequence Numbers
Logging Subsystem Cursors	
DB_ENV->log_cursor()	Create a log cursor handle
The DB_LOGC Handle	A log cursor handle
DB_LOGC->close()	Close a log cursor
DB_LOGC->get()	Retrieve a log record
Logging Subsystem Configuration	
DB_ENV->log_set_config(), DB_ENV->log_get_config()	Configure the logging subsystem
DB_ENV->set_lg_bsize(), DB_ENV->get_lg_bsize()	Set/get log buffer size
DB_ENV->set_lg_dir(), DB_ENV->get_lg_dir()	Set/get the environment logging directory
DB_ENV->set_lg_filemode(), DB_ENV->get_lg_filemode()	Set/get log file mode
DB_ENV->set_lg_max(), DB_ENV->get_lg_max()	Set/get log file size
DB_ENV->set_lg_regionmax(), DB_ENV->get_lg_regionmax()	Set/get logging region size

DB_ENV->get_lg_bsize()

```
#include <db.h>

int
DB_ENV->get_lg_bsize(DB_ENV *dbenv, u_int32_t *lg_bsizep);
```

The `DB_ENV->get_lg_bsize()` method returns the size of the log buffer, in bytes. You can manage this value using the DB_ENV->set_lg_bsize() (page 355) method.

The `DB_ENV->get_lg_bsize()` method may be called at any time during the life of the application.

The `DB_ENV->get_lg_bsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_bsizep

The `DB_ENV->get_lg_bsize()` method returns the size of the log buffer, in bytes in **lg_bsizep**.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->set_lg_bsize() (page 355)

DB_ENV->get_lg_dir()

```
#include <db.h>

int
DB_ENV->get_lg_dir(DB_ENV *dbenv, const char **dirp);
```

The `DB_ENV->get_lg_dir()` method returns the log directory, which is the location for logging files. You can manage this value using the DB_ENV->set_lg_dir() (page 357) method.

The `DB_ENV->get_lg_dir()` method may be called at any time during the life of the application.

The `DB_ENV->get_lg_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dirp

The `DB_ENV->get_lg_dir()` method returns a reference to the log directory in **dirp**.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->set_lg_dir() (page 357)

DB_ENV->get_lg_filemode()

```
#include <db.h>

int
DB_ENV->get_lg_filemode(DB_ENV *dbenv, int *lg_modep);
```

The `DB_ENV->set_lg_filemode()` method returns the log file mode. You can manage this value using the DB_ENV->set_lg_filemode() (page 359) method.

The `DB_ENV->set_lg_filemode()` method may be called at any time during the life of the application.

The `DB_ENV->set_lg_filemode()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_modep

The `DB_ENV->set_lg_filemode()` method returns the log file mode in **lg_modep**.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->set_lg_filemode() (page 359)

DB_ENV->get_lg_max()

```
#include <db.h>

int
DB_ENV->get_lg_max(DB_ENV *dbenv, u_int32_t *lg_maxp);
```

The `DB_ENV->get_lg_max()` method returns the maximum log file size. You can manage this value using the DB_ENV->set_lg_max() (page 360) method.

The `DB_ENV->get_lg_max()` method may be called at any time during the life of the application.

The `DB_ENV->get_lg_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_maxp

The `DB_ENV->get_lg_max()` method returns the maximum log file size in **lg_maxp**.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->set_lg_max() (page 360)

DB_ENV->get_lg_regionmax()

```
#include <db.h>

int
DB_ENV->get_lg_regionmax(DB_ENV *dbenv, u_int32_t *lg_regionmaxp);
```

The DB_ENV->get_lg_regionmax() method returns the size of the underlying logging subsystem region. You can manage this value using the DB_ENV->set_lg_regionmax() (page 362) method.

The DB_ENV->get_lg_regionmax() method may be called at any time during the life of the application.

The DB_ENV->get_lg_regionmax() method returns a non-zero error value on failure and 0 on success.

Parameters

lg_regionmaxp

The DB_ENV->get_lg_regionmax() method returns the size of the underlying logging subsystem region in **lg_regionmaxp**.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->set_lg_regionmax() (page 362)

DB_ENV->log_archive()

```
#include <db.h>

int
DB_ENV->log_archive(DB_ENV *env, char *(*listp)[], u_int32_t flags);
```

The DB_ENV->log_archive() method returns an array of log or database filenames.

By default, DB_ENV->log_archive() returns the names of all of the log files that are no longer in use (for example, that are no longer involved in active transactions), and that may safely be archived for catastrophic recovery and then removed from the system. If there are no filenames to return, the memory location to which **listp** refers will be set to NULL.

When Replication Manager is in use, log archiving is performed in a replication group-aware manner such that the log file status of other sites in the group is considered to determine if a log file is in use.

Arrays of log filenames are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

Log cursor handles (returned by the DB_ENV->log_cursor() (page 340) method) may have open file descriptors for log files in the database environment. Also, the Berkeley DB interfaces to the database environment logging subsystem (for example, DB_ENV->log_put() (page 346) and DB_TXN->abort() (page 554)) may allocate log cursors and have open file descriptors for log files as well. On operating systems where filesystem related system calls (for example, rename and unlink on Windows/NT) can fail if a process has an open file descriptor for the affected file, attempting to move or remove the log files listed by DB_ENV->log_archive() may fail. All Berkeley DB internal use of log cursors operates on active log files only and furthermore, is short-lived in nature. So, an application seeing such a failure should be restructured to close any open log cursors it may have, and otherwise to retry the operation until it succeeds. (Although the latter is not likely to be necessary; it is hard to imagine a reason to move or rename a log file in which transactions are being logged or aborted.)

See db_archive for more information on database archival procedures.

The DB_ENV->log_archive() method is the underlying method used by the db_archive utility. See the db_archive utility source code for an example of using DB_ENV->log_archive() in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The DB_ENV->log_archive() method returns a non-zero error value on failure and 0 on success.

Parameters

listp

The **listp** parameter references memory into which the allocated array of log or database filenames is copied. If there are no filenames to return, the memory location to which **listp** refers will be set to NULL.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_ARCH_ABS`

  All pathnames are returned as absolute pathnames, instead of relative to the database home directory.

- `DB_ARCH_DATA`

  Return the database files that need to be archived in order to recover the database from catastrophic failure. If any of the database files have not been accessed during the lifetime of the current log files, `DB_ENV->log_archive()` will not include them in this list. It is also possible that some of the files referred to by the log have since been deleted from the system.

  The `DB_ARCH_DATA` and `DB_ARCH_LOG flags` are mutually exclusive.

- `DB_ARCH_LOG`

  Return all the log filenames, regardless of whether or not they are in use.

  The `DB_ARCH_DATA` and `DB_ARCH_LOG flags` are mutually exclusive.

- `DB_ARCH_REMOVE`

  Remove log files that are no longer needed; no filenames are returned. Automatic log file removal is likely to make catastrophic recovery impossible.

  The `DB_ARCH_REMOVE` flag may not be specified with any other flag.

Errors

The `DB_ENV->log_archive()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_cursor()

```
#include <db.h>

int
DB_ENV->log_cursor(DB_ENV *dbenv, DB_LOGC **cursorp, u_int32_t flags);
```

The `DB_ENV->log_cursor()` method returns a created log cursor.

The `DB_ENV->log_cursor()` method returns a non-zero error value on failure and 0 on success.

Parameters

cursorp

The **cursorp** parameter references memory into which a pointer to the created log cursor is copied.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_ENV->log_cursor()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_file()

```
#include <db.h>

int
DB_ENV->log_file(DB_ENV *env,
    const DB_LSN *lsn, char *namep, size_t len);
```

The DB_ENV->log_file() method maps DB_LSN structures to filenames, returning the name of the file containing the record named by **lsn**.

This mapping of DB_LSN structures to files is needed for database administration. For example, a transaction manager typically records the earliest DB_LSN needed for restart, and the database administrator may want to archive log files to tape when they contain only DB_LSN entries before the earliest one needed for restart.

The DB_ENV->log_file() method returns a non-zero error value on failure and 0 on success.

Parameters

lsn

The **lsn** parameter is the DB_LSN structure for which a filename is wanted.

namep

The **namep** parameter references memory into which the name of the file containing the record named by **lsn** is copied.

len

The **len** parameter is the length of the **namep** buffer in bytes. If **namep** is too short to hold the filename, DB_ENV->log_file() will fail. (Log filenames are always 14 characters long.)

Errors

The DB_ENV->log_file() method may fail and return one of the following non-zero errors:

EINVAL

If supplied buffer was too small to hold the log filename; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_flush()

```
#include <db.h>

int
DB_ENV->log_flush(DB_ENV *env, const DB_LSN *lsn);
```

The `DB_ENV->log_flush()` method writes log records to disk.

The `DB_ENV->log_flush()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn

All log records with DB_LSN values less than or equal to the **lsn** parameter are written to disk. If **lsn** is NULL, all records in the log are flushed.

Errors

The `DB_ENV->log_flush()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_get_config()

```
#include <db.h>

int
DB_ENV->log_get_config(DB_ENV *dbenv, u_int32_t which, int *onoffp);
```

The `DB_ENV->log_get_config()` method returns whether the specified **which** parameter is currently set or not. You can manage this value using the DB_ENV->log_set_config() (page 348) method.

The `DB_ENV->log_get_config()` method may be called at any time during the life of the application.

The `DB_ENV->log_get_config()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter is the message value for which configuration is being checked. Must be set to one of the following values:

- `DB_LOG_DIRECT`

  System buffering is turned off for Berkeley DB log files to avoid double caching.

- `DB_LOG_DSYNC`

  Berkeley DB is configured to flush log writes to the backing disk before returning from the write system call, rather than flushing log writes explicitly in a separate system call, as necessary.

- `DB_LOG_AUTO_REMOVE`

  Berkeley DB automatically removes log files that are no longer needed.

- `DB_LOG_IN_MEMORY`

  Transaction logs are maintained in memory rather than on disk. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability).

- `DB_LOG_ZERO`

  All pages of a log file are zeroed when that log file is created.

onoffp

The **onoffp** parameter references memory into which the configuration of the specified **which** parameter is copied.

If the returned **onoff** value is zero, the parameter is off; otherwise, on.

Class

DB_ENV

See Also

Logging Subsystem and Related Methods (page 331), DB_ENV->log_set_config() (page 348)

DB_ENV->log_printf()

```
#include <db.h>

int
DB_ENV->log_printf(DB_ENV *env, DB_TXN *txnid, const char *fmt, ...);
```

The `DB_ENV->log_printf()` method appends an informational message to the Berkeley DB database environment log files.

The `DB_ENV->log_printf()` method allows applications to include information in the database environment log files, for later review using the db_printlog utility. This method is intended for debugging and performance tuning.

The `DB_ENV->log_printf()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the logged message refers to an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); otherwise NULL.

fmt

A format string that specifies how subsequent arguments (or arguments accessed via the variable-length argument facilities of stdarg(3)) are converted for output. The format string may contain any formatting directives supported by the underlying C library vsnprintf(3) function.

Errors

The `DB_ENV->log_printf()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_put()

```
#include <db.h>

int
DB_ENV->log_put(DB_ENV *env,
    DB_LSN *lsn, const DBT *data, u_int32_t flags);
```

The `DB_ENV->log_put()` method appends records to the log. The DB_LSN of the put record is returned in the **lsn** parameter.

The `DB_ENV->log_put()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn

The **lsn** parameter references memory into which the DB_LSN of the put record is copied.

data

The **data** parameter is the record to write to the log.

The caller is responsible for providing any necessary structure to **data**. (For example, in a write-ahead logging protocol, the application must understand what part of **data** is an operation code, what part is redo information, and what part is undo information. In addition, most transaction managers will store in **data** the DB_LSN of the previous log record for the same transaction, to support chaining back through the transaction's log records during undo.)

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_FLUSH`

  The log is forced to disk after this record is written, guaranteeing that all records with DB_LSN values less than or equal to the one being "put" are on disk before `DB_ENV->log_put()` returns.

Errors

The `DB_ENV->log_put()` method may fail and return one of the following non-zero errors:

EINVAL

If the record to be logged is larger than the maximum log record; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_set_config()

```
#include <db.h>

int
DB_ENV->log_set_config(DB_ENV *dbenv, u_int32_t flags, int onoff);
```

The `DB_ENV->log_set_config()` method configures the Berkeley DB logging subsystem.

The `DB_ENV->log_set_config()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->log_set_config()` method may be called at any time during the life of the application.

The `DB_ENV->log_set_config()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_LOG_DIRECT`

  Turn off system buffering of Berkeley DB log files to avoid double caching.

  Calling `DB_ENV->log_set_config()` with the DB_LOG_DIRECT flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_LOG_DIRECT flag or the flag should be specified in the DB_CONFIG configuration file.

  The `DB_LOG_DIRECT` flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_LOG_DSYNC`

  Configure Berkeley DB to flush log writes to the backing disk before returning from the write system call, rather than flushing log writes explicitly in a separate system call, as necessary. This is only available on some systems (for example, systems supporting the IEEE/ANSI Std 1003.1 (POSIX) standard O_DSYNC flag, or systems supporting the Windows FILE_FLAG_WRITE_THROUGH flag). This flag may result in inaccurate file modification times and other file-level information for Berkeley DB log files. This flag may offer a performance increase on some systems and a performance decrease on others.

  Calling `DB_ENV->log_set_config()` with the DB_LOG_DSYNC flag only affects the specified DB_ENV handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DB_ENV handles opened in the environment must either set the DB_LOG_DSYNC flag or the flag should be specified in the DB_CONFIG configuration file.

The `DB_LOG_DSYNC` flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_LOG_AUTO_REMOVE`

  If set, Berkeley DB will automatically remove log files that are no longer needed.

  Automatic log file removal is likely to make catastrophic recovery impossible.

  Replication Manager applications operate in a group-aware manner for log file removal, and automatic log file removal simplifies the application.

  Replication Base API applications will rarely want to configure automatic log file removal as it increases the likelihood a master will be unable to satisfy a client's request for a recent log record.

  Calling `DB_ENV->log_set_config()` with the `DB_LOG_AUTO_REMOVE` flag affects the database environment, including all threads of control accessing the database environment.

  The `DB_LOG_AUTO_REMOVE` flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_LOG_IN_MEMORY`

  If set, maintain transaction logs in memory rather than on disk. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the application or system fails, integrity will not persist. All database files must be verified and/or restored from a replication group master or archival backup after application or system failure.

  When in-memory logs are configured and no more log buffer space is available, Berkeley DB methods may return an additional error value, `DB_LOG_BUFFER_FULL`. When choosing log buffer and file sizes for in-memory logs, applications should ensure the in-memory log buffer size is large enough that no transaction will ever span the entire buffer, and avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started in the first log "file" is still active.

  Calling `DB_ENV->log_set_config()` with the `DB_LOG_IN_MEMORY` flag affects the database environment, including all threads of control accessing the database environment.

  The `DB_LOG_IN_MEMORY` flag may be used to configure Berkeley DB only before the DB_ENV->open() (page 231) method is called.

- `DB_LOG_ZERO`

  If set, zero all pages of a log file when that log file is created. This has shown to provide greater transaction throughput in some environments. The log file will be zeroed by the thread which needs to re-create the new log file. Other threads may not write to the log file while this is happening.

Calling `DB_ENV->log_set_config()` with the `DB_LOG_ZERO` flag affects only the current environment handle.

The `DB_LOG_ZERO` flag may be used to configure Berkeley DB at any time.

onoff

If the **onoff** parameter is zero, the specified flags are cleared; otherwise they are set.

Errors

The `DB_ENV->log_set_config()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_stat()

```
#include <db.h>

int
DB_ENV->log_stat(DB_ENV *env, DB_LOG_STAT **statp, u_int32_t flags);
```

The `DB_ENV->log_stat()` method returns the logging subsystem statistics.

The `DB_ENV->log_stat()` method creates a statistical structure of type `DB_LOG_STAT` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following `DB_LOG_STAT` fields will be filled in:

- **u_int32_t st_magic;**

  The magic number that identifies a file as a log file.

- **u_int32_t st_version;**

  The version of the log file type.

- **int st_mode;**

  The mode of any created log files.

- **u_int32_t st_lg_bsize;**

  The in-memory log record cache size.

- **u_int32_t st_lg_size;**

  The log file size.

- **uintmax_t st_record;**

  The number of records written to this log.

- **u_int32_t st_w_mbytes;**

  The number of megabytes written to this log.

- **u_int32_t st_w_bytes;**

  The number of bytes over and above **st_w_mbytes** written to this log.

- **u_int32_t st_wc_mbytes;**

The number of megabytes written to this log since the last checkpoint.

- **u_int32_t st_wc_bytes;**

  The number of bytes over and above **st_wc_mbytes** written to this log since the last checkpoint.

- **uintmax_t st_wcount;**

  The number of times the log has been written to disk.

- **uintmax_t st_wcount_fill;**

  The number of times the log has been written to disk because the in-memory log record cache filled up.

- **uintmax_t st_rcount;**

  The number of times the log has been read from disk.

- **uintmax_t st_scount;**

  The number of times the log has been flushed to disk.

- **u_int32_t st_cur_file;**

  The current log file number.

- **u_int32_t st_cur_offset;**

  The byte offset in the current log file.

- **u_int32_t st_disk_file;**

  The log file number of the last record known to be on disk.

- **u_int32_t st_disk_offset;**

  The byte offset of the last record known to be on disk.

- **u_int32_t st_maxcommitperflush;**

  The maximum number of commits contained in a single log flush.

- **u_int32_t st_mincommitperflush;**

  The minimum number of commits contained in a single log flush that contained a commit.

- **roff_t st_regsize;**

  The size of the log region, in bytes.

- **uintmax_t st_region_wait;**

The number of times that a thread of control was forced to wait before obtaining the log region mutex.

- **uintmax_t st_region_nowait;**

  The number of times that a thread of control was able to obtain the log region mutex without waiting.

The `DB_ENV->log_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->log_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->log_stat()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->log_stat_print()

```
#include <db.h>

int
DB_ENV->log_stat_print(DB_ENV *env, u_int32_t flags);
```

The `DB_ENV->log_stat_print()` method displays the logging subsystem statistical information, as described for the `DB_ENV->log_stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->log_stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->log_stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->set_lg_bsize()

```
#include <db.h>

int
DB_ENV->set_lg_bsize(DB_ENV *dbenv, u_int32_t lg_bsize);
```

Sets the size of the in-memory log buffer, in bytes.

When the logging subsystem is configured for on-disk logging, the default size of the in-memory log buffer is approximately 32KB. Log information is stored in-memory until the storage space fills up or s transaction commit forces the information to be flushed to stable storage. In the presence of long-running transactions or transactions producing large amounts of data, larger buffer sizes can increase throughput.

When the logging subsystem is configured for in-memory logging, the default size of the in-memory log buffer is 1MB. Log information is stored in-memory until the storage space fills up or transaction abort or commit frees up the memory for new transactions. In the presence of long-running transactions or transactions producing large amounts of data, the buffer size must be sufficient to hold all log information that can accumulate during the longest running transaction. When choosing log buffer and file sizes for in-memory logs, applications should ensure the in-memory log buffer size is large enough that no transaction will ever span the entire buffer, and avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started in the first log "file" is still active.

The database environment's log buffer size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lg_bsize", one or more whitespace characters, and the size in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lg_bsize()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lg_bsize()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lg_bsize()` will be ignored.

The `DB_ENV->set_lg_bsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_bsize

The **lg_bsize** parameter is the size of the in-memory log buffer, in bytes.

Errors

The `DB_ENV->set_lg_bsize()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->set_lg_dir()

```
#include <db.h>

int
DB_ENV->set_lg_dir(DB_ENV *dbenv, const char *dir);
```

The path of a directory to be used as the location of logging files. Log files created by the Log Manager subsystem will be created in this directory.

If no logging directory is specified, log files are created in the environment home directory. See Berkeley DB File Naming for more information.

For the greatest degree of recoverability from system or application failure, database files and log files should be located on separate physical devices.

The database environment's logging directory may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lg_dir", one or more whitespace characters, and the directory name. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lg_dir()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_lg_dir()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lg_dir()` must be consistent with the existing environment or corruption can occur.

The `DB_ENV->set_lg_dir()` method returns a non-zero error value on failure and 0 on success.

Parameters

dir

The **dir** parameter is the directory used to store the logging files.

When using a Unicode build on Windows (the default), the **dir** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

Errors

The `DB_ENV->set_lg_dir()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->set_lg_filemode()

```
#include <db.h>

int
DB_ENV->set_lg_filemode(DB_ENV *dbenv, int lg_filemode);
```

Set the absolute file mode for created log files. This method is **only** useful for the rare Berkeley DB application that does not control its umask value.

Normally, if Berkeley DB applications set their umask appropriately, all processes in the application suite will have read permission on the log files created by any process in the application suite. However, if the Berkeley DB application is a library, a process using the library might set its umask to a value preventing other processes in the application suite from reading the log files it creates. In this rare case, the `DB_ENV->set_lg_filemode()` method can be used to set the mode of created log files to an absolute value.

The database environment's log file mode may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lg_filemode", one or more whitespace characters, and the absolute mode of created log files. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lg_filemode()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lg_filemode()` method may be called at any time during the life of the application.

The `DB_ENV->set_lg_filemode()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_filemode

The **lg_filemode** parameter is the absolute mode of the created log file.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->set_lg_max()

```
#include <db.h>

int
DB_ENV->set_lg_max(DB_ENV *dbenv, u_int32_t lg_max);
```

Sets the maximum size of a single file in the log, in bytes. Because DB_LSN file offsets are unsigned four-byte values, the set value may not be larger than the maximum unsigned four-byte value.

When the logging subsystem is configured for on-disk logging, the default size of a log file is 10MB.

When the logging subsystem is configured for in-memory logging, the default size of a log file is 256KB. In addition, the configured log buffer size must be larger than the log file size. (The logging subsystem divides memory configured for in-memory log records into "files", as database environments configured for in-memory log records may exchange log records with other members of a replication group, and those members may be configured to store log records on-disk.) When choosing log buffer and file sizes for in-memory logs, applications should ensure the in-memory log buffer size is large enough that no transaction will ever span the entire buffer, and avoid a state where the in-memory buffer is full and no space can be freed because a transaction that started in the first log "file" is still active.

See Log File Limits for more information.

The database environment's log file size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lg_max", one or more whitespace characters, and the size in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lg_max()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lg_max()` method may be called at any time during the life of the application.

If no size is specified by the application, the size last specified for the database region will be used, or if no database region previously existed, the default will be used.

The `DB_ENV->set_lg_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_max

The **lg_max** parameter is the size of a single log file, in bytes.

Errors

The `DB_ENV->set_lg_max()` method may fail and return one of the following non-zero errors:

EINVAL

If the size of the log file is less than four times the size of the in-memory log buffer; the specified log file size was too large; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_ENV->set_lg_regionmax()

```
#include <db.h>

int
DB_ENV->set_lg_regionmax(DB_ENV *dbenv, u_int32_t lg_regionmax);
```

Set the size of the underlying logging area of the Berkeley DB environment, in bytes. By default, or if the value is set to 0, the minimum region size is used, approximately 128KB. The log region is used to store filenames, and so may need to be increased in size if a large number of files will be opened and registered with the specified Berkeley DB environment's log manager.

The database environment's log region size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_lg_regionmax", one or more whitespace characters, and the size in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_lg_regionmax()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_lg_regionmax()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_lg_regionmax()` will be ignored.

The `DB_ENV->set_lg_regionmax()` method returns a non-zero error value on failure and 0 on success.

Parameters

lg_regionmax

The **lg_regionmax** parameter is the size of the logging area in the Berkeley DB environment, in bytes.

Errors

The `DB_ENV->set_lg_regionmax()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

The DB_LOGC Handle

```
#include <db.h>

typedef struct __typedef struct __db_log_cursor DB_LOGC;
```

The DB_LOGC object is the handle for a cursor into the log files, supporting sequential access to the records stored in log files. The handle is not free-threaded. Once the DB_LOGC->close() (page 365) method is called, the handle may not be accessed again, regardless of that method's return.

For more information, see the DB_LSN handle.

DB_LOGC->close()

```
#include <db.h>

int
DB_LOGC->close(DB_LOGC *cursor, u_int32_t flags);
```

The `DB_LOGC->close()` method discards the log cursor. After `DB_LOGC->close()` has been called, regardless of its return, the cursor handle may not be used again.

The `DB_LOGC->close()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_LOGC->close()` method may fail and return one of the following non-zero errors:

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

DB_LOGC->get()

```
#include <db.h>

int
DB_LOGC->get(DB_LOGC *logc, DB_LSN *lsn, DBT *data, u_int32_t flags);
```

The `DB_LOGC->get()` method returns records from the log.

Unless otherwise specified, the `DB_LOGC->get()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn

When the **flag** parameter is set to DB_CURRENT, DB_FIRST, DB_LAST, DB_NEXT or DB_PREV, the **lsn** parameter is overwritten with the DB_LSN value of the record retrieved. When **flag** is set to DB_SET, the **lsn** parameter is the DB_LSN value of the record to be retrieved.

data

The data field of the **data** structure is set to the record retrieved, and the size field indicates the number of bytes in the record. See DBT for a description of other fields in the **data** structure. The DB_DBT_MALLOC, DB_DBT_REALLOC and DB_DBT_USERMEM flags may be specified for any DBT used for data retrieval.

flags

The **flags** parameter must be set to one of the following values:

- `DB_CURRENT`

  Return the log record to which the log currently refers.

- `DB_FIRST`

  The first record from any of the log files found in the log directory is returned in the **data** parameter. The **lsn** parameter is overwritten with the DB_LSN of the record returned.

  The `DB_LOGC->get()` method will return DB_NOTFOUND if DB_FIRST is set and the log is empty.

- `DB_LAST`

  The last record in the log is returned in the **data** parameter. The **lsn** parameter is overwritten with the DB_LSN of the record returned.

  The `DB_LOGC->get()` method will return DB_NOTFOUND if DB_LAST is set and the log is empty.

- `DB_NEXT`

The current log position is advanced to the next record in the log, and that record is returned in the **data** parameter. The **lsn** parameter is overwritten with the DB_LSN of the record returned.

If the cursor has not been initialized via DB_FIRST, DB_LAST, DB_SET, DB_NEXT, or DB_PREV, `DB_LOGC->get()` will return the first record in the log.

The `DB_LOGC->get()` method will return DB_NOTFOUND if DB_NEXT is set and the last log record has already been returned or the log is empty.

- `DB_PREV`

  The current log position is advanced to the previous record in the log, and that record is returned in the **data** parameter. The **lsn** parameter is overwritten with the DB_LSN of the record returned.

  If the cursor has not been initialized via DB_FIRST, DB_LAST, DB_SET, DB_NEXT, or DB_PREV, `DB_LOGC->get()` will return the last record in the log.

  The `DB_LOGC->get()` method will return DB_NOTFOUND if DB_PREV is set and the first log record has already been returned or the log is empty.

- `DB_SET`

  Retrieve the record specified by the **lsn** parameter.

Errors

The `DB_LOGC->get()` method may fail and return one of the following non-zero errors:

EINVAL

If the DB_CURRENT flag was set and the log cursor has not yet been initialized; the DB_CURRENT, DB_NEXT, or DB_PREV flags were set and the log was opened with the DB_THREAD flag set; the DB_SET flag was set and the specified log sequence number does not appear in the log; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

log_compare

```
#include <db.h>

int
log_compare(const DB_LSN *lsn0, const DB_LSN *lsn1);
```

The `log_compare()` method allows the caller to compare two `DB_LSN` structures, returning 0 if they are equal, 1 if **lsn0** is greater than **lsn1**, and -1 if **lsn0** is less than **lsn1**.

Parameters

lsn0

The **lsn0** parameter is one of the `DB_LSN` structures to be compared.

lsn1

The **lsn1** parameter is one of the `DB_LSN` structures to be compared.

Class

DB_ENV, DB_LOGC, DB_LSN

See Also

Logging Subsystem and Related Methods (page 331)

Chapter 8. The DB_MPOOLFILE Handle

```
#include <db.h>

typedef struct __db_mpoolfile DB_MPOOLFILE;
```

The memory pool interfaces for the Berkeley DB database environment are methods of the DB_ENV handle. The DB_ENV memory pool methods and the `DB_MPOOLFILE` class provide general-purpose, page-oriented buffer management of files. Although designed to work with the other DBclasses, they are also useful for more general purposes. The memory pools are referred to in this document as simply *the cache*.

The cache may be shared between processes. The cache is usually filled by pages from one or more files. Pages in the cache are replaced in LRU (least-recently-used) order, with each new page replacing the page that has been unused the longest. Pages retrieved from the cache using DB_MPOOLFILE->get() (page 404) are *pinned* in the cache until they are returned to the control of the cache using the DB_MPOOLFILE->put() (page 409) method.

The `DB_MPOOLFILE` object is the handle for a file in the cache. The handle is not free-threaded. Once the DB_MPOOLFILE->close() (page 403) method is called, the handle may not be accessed again, regardless of that method's return.

Memory Pools and Related Methods

Memory Pools and Related Methods	Description
DB->get_mpf()	Return the DB_MPOOLFILE for a DB
DB_ENV->memp_stat()	Return cache statistics
DB_ENV->memp_stat_print()	Print cache statistics
DB_ENV->memp_sync()	Flush all pages from the cache
DB_ENV->memp_trickle()	Flush some pages from the cache
Memory Pool Configuration	
DB_ENV->memp_register()	Register a custom file type
DB_ENV->set_cache_max(), DB_ENV->get_cache_max()	Set/get the maximum cache size
DB_ENV->set_cachesize(), DB_ENV->get_cachesize()	Set/get the environment cache size
DB_ENV->set_mp_max_openfd(), DB_ENV->get_mp_max_openfd()	Set/get the maximum number of open file descriptors
DB_ENV->set_mp_max_write(), DB_ENV->get_mp_max_write()	Set/get the maximum number of sequential disk writes
DB_ENV->set_mp_mmapsize(), DB_ENV->get_mp_mmapsize()	Set/get maximum file size to memory map when opened read-only
DB_ENV->set_mp_pagesize(), DB_ENV->get_mp_pagesize()	Set/get page size to configure the buffer pool
DB_ENV->set_mp_tablesize(), DB_ENV->get_mp_tablesize()	Set/get the hash table size
Memory Pool Files	
DB_ENV->memp_fcreate()	Create a memory pool file handle
DB_MPOOLFILE->close()	Close a file in the cache
DB_MPOOLFILE->get()	Get page from a file in the cache
DB_MPOOLFILE->open()	Open a file in the cache
DB_MPOOLFILE->put()	Return a page to the cache
DB_MPOOLFILE->sync()	Flush pages from a file from the cache
Memory Pool File Configuration	
DB_MPOOLFILE->set_clear_len(), DB_MPOOLFILE->get_clear_len()	Set/get number of bytes to clear when creating a new page
DB_MPOOLFILE->set_fileid(), DB_MPOOLFILE->get_fileid()	Set/get file unique identifier
DB_MPOOLFILE->set_flags(), DB_MPOOLFILE->get_flags()	Set/get file options

Memory Pools and Related Methods	Description
DB_MPOOLFILE->set_ftype(), DB_MPOOLFILE->get_ftype()	Set/get file type
DB_MPOOLFILE->set_lsn_offset(), DB_MPOOLFILE->get_lsn_offset()	Set/get file log-sequence-number offset
DB_MPOOLFILE->set_maxsize(), DB_MPOOLFILE->get_maxsize()	Set/get maximum file size
DB_MPOOLFILE->set_pgcookie(), DB_MPOOLFILE->get_pgcookie()	Set/get file cookie for pgin/pgout
DB_MPOOLFILE->set_priority(), DB_MPOOLFILE->get_priority()	Set/get cache file priority

DB->get_mpf()

```
#include <db.h>

DB_MPOOLFILE *
DB->get_mpf(DB *db);
```

The `DB->get_mpf()` method returns the handle for the cache file underlying the database.

The `DB->get_mpf()` method should be used with caution on a replication client site. This method exposes an internal structure that may not be valid after a client site synchronizes with its master site.

The `DB->get_mpf()` method may be called at any time during the life of the application.

Class

DB

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->get_cache_max()

```
#include <db.h>

int
DB_ENV->get_cache_max(DB_ENV *dbenv, u_int32_t *gbytesp,
                      u_int32_t *bytesp);
```

The `DB_ENV->get_cache_max()` method returns the maximum size of the cache as set using the DB_ENV->set_cache_max() (page 392) method.

The `DB_ENV->get_cache_max()` method may be called at any time during the life of the application.

The `DB_ENV->get_cache_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytesp

The **gbytesp** parameter references memory into which the gigabytes of memory in the cache is copied.

bytesp

The **bytesp** parameter references memory into which the additional bytes of memory in the cache is copied.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192), DB_ENV->set_cache_max() (page 392)

DB_ENV->get_cachesize()

```
#include <db.h>

int
DB_ENV->get_cachesize(DB_ENV *dbenv,
    u_int32_t *gbytesp, u_int32_t *bytesp, int *ncachep);
```

The `DB_ENV->get_cachesize()` method returns the current size and composition of the cache, as set using the DB_ENV->set_cachesize() (page 394) method.

The `DB_ENV->get_cachesize()` method may be called at any time during the life of the application.

The `DB_ENV->get_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytesp

The **gbytesp** parameter references memory into which the gigabytes of memory in the cache is copied.

bytesp

The **bytesp** parameter references memory into which the additional bytes of memory in the cache is copied.

ncachep

The **ncachep** parameter references memory into which the number of caches is copied.

Class

DB_ENV

See Also

Memory Pools and Related Methods (page 370), Database Environments and Related Methods (page 192), DB_ENV->set_cachesize() (page 394)

DB_ENV->get_mp_max_openfd()

```
#include <db.h>

int
DB_ENV->get_mp_max_openfd(DB_ENV *env, int *maxopenfdp);
```

Returns the maximum number of file descriptors the library will open concurrently when flushing dirty pages from the cache. This value is set by the DB_ENV->set_mp_max_openfd() (page 396) method.

The `DB_ENV->get_mp_max_openfd()` method may be called at any time during the life of the application.

The `DB_ENV->get_mp_max_openfd()` method returns a non-zero error value on failure and 0 on success.

Parameters

maxopenfdp

The `DB_ENV->get_mp_max_openfd()` method returns the maximum number of file descriptors open in **maxopenfdp**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->set_mp_max_openfd() (page 396)

DB_ENV->get_mp_max_write()

```
#include <db.h>

int
DB_ENV->get_mp_max_write(DB_ENV *env, int *maxwritep,
                         db_timeout_t *maxwrite_sleepp);
```

The `DB_ENV->get_mp_max_write()` method returns the current maximum number of sequential write operations and microseconds to pause that the library can schedule when flushing dirty pages from the cache. These values are set by the DB_ENV->set_mp_max_write() (page 397) method.

The `DB_ENV->get_mp_max_write()` method may be called at any time during the life of the application.

The `DB_ENV->get_mp_max_write()` method returns a non-zero error value on failure and 0 on success.

Parameters

maxwritep

The **maxwritep** parameter references memory into which the maximum number of sequential write operations is copied.

maxwrite_sleepp

The **maxwrite_sleepp** parameter references memory into which the microseconds to pause before scheduling further write operations is copied.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->set_mp_max_write() (page 397)

DB_ENV->get_mp_mmapsize()

```
#include <db.h>

int
DB_ENV->get_mp_mmapsize(DB_ENV *dbenv, size_t *mp_mmapsizep);
```

The `DB_ENV->get_mp_mmapsize()` method returns the the maximum file size, in bytes, for a file to be mapped into the process address space. This value can be managed using the DB_ENV->set_mp_mmapsize() (page 399) method.

The `DB_ENV->get_mp_mmapsize()` method may be called at any time during the life of the application.

The `DB_ENV->get_mp_mmapsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

mp_mmapsizep

The `DB_ENV->get_mp_mmapsize()` method returns the maximum file map size in **mp_mmapsizep**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->set_mp_mmapsize() (page 399)

DB_ENV->get_mp_pagesize()

```
#include <db.h>

int
DB_ENV->get_mp_pagesize(DB_ENV *dbenv, u_int32_t *pagesizep);
```

The `DB_ENV->get_mp_pagesize()` method returns the assumed page size used to configure the buffer pool.

The `DB_ENV->get_mp_pagesize()` method may be called at any time during the life of the application.

Parameters

pagesizep

This parameter specifies the assumed page size used to configure the buffer pool.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->set_mp_pagesize() (page 401)

DB_ENV->get_mp_tablesize()

```
#include <db.h>

int
DB_ENV->get_mp_tablesize(DB_ENV *dbenv, u_int32_t *tablesizep);
```

The `DB_ENV->get_mp_tablesize()` method returns the hash table size in the buffer pool.

Parameters

tablesize

This parameter specifies the hash table size in the buffer pool.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->set_mp_tablesize() (page 402)

DB_ENV->memp_fcreate()

```
#include <db.h>

int
DB_ENV->memp_fcreate(DB_ENV *dbenvp, DB_MPOOLFILE **dbmfp,
                     u_int32_t flags);
```

The `DB_ENV->memp_fcreate()` method creates a DB_MPOOLFILE structure that is the handle for a Berkeley DB cache (that is, a shared memory buffer pool file). A pointer to this structure is returned in the memory to which **dbmfp** refers. Calling the DB_MPOOLFILE->close() (page 403) method will discard the returned handle.

The `DB_ENV->memp_fcreate()` method returns a non-zero error value on failure and 0 on success.

Parameters

dbmfp

The `DB_ENV->memp_fcreate()` method returns a pointer to a mpool structure in **dbmfp**.

flags

The **flags** parameter is currently unused, and must be set to 0.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->memp_register()

```
#include <db.h>

int
DB_ENV->memp_register(DB_ENV *env, int ftype,
    int (*pgin_fcn)(DB_ENV *env, db_pgno_t pgno, void *pgaddr,
    DBT *pgcookie), int (*pgout_fcn)(DB_ENV *env, db_pgno_t pgno,
    void *pgaddr, DBT *pgcookie));
```

The `DB_ENV->memp_register()` method registers page-in and page-out functions for files of type **ftype** in the cache.

If the **pgin_fcn** function is non-NULL, it is called each time a page is read into the cache from a file of type **ftype**, or a page is created for a file of type **ftype** (see the DB_MPOOL_CREATE flag for the DB_MPOOLFILE->get() (page 404) method).

If the **pgout_fcn** function is non-NULL, it is called each time a page is written to a file of type **ftype**.

The purpose of the `DB_ENV->memp_register()` function is to support processing when pages are entered into, or flushed from, the cache. For example, this functionality might be used to do byte-endian conversion as pages are read from, or written to, the underlying file.

A file type must be specified to make it possible for unrelated threads or processes that are sharing a cache, to evict each other's pages from the cache. During initialization, applications should call `DB_ENV->memp_register()` for each type of file requiring input or output processing that will be sharing the underlying cache. (No registry is necessary for the standard Berkeley DB access method types because DB->open() (page 66) registers them separately.)

If a thread or process does not call `DB_ENV->memp_register()` for a file type, it is impossible for it to evict pages for any file requiring input or output processing from the cache. For this reason, `DB_ENV->memp_register()` should always be called by each application sharing a cache for each type of file included in the cache, regardless of whether or not the application itself uses files of that type.

The `DB_ENV->memp_register()` method returns a non-zero error value on failure and 0 on success.

Parameters

ftype

The **ftype** parameter specifies the type of file for which the page-in and page-out functions will be called.

The **ftype** value for a file must be a non-zero positive number less than 128 (0 and negative numbers are reserved for internal use by the Berkeley DB library).

pgin_fcn, pgout_fcn

The page-in and page-out functions.

The **pgin_fcn** and **pgout_fcn** functions are called with a reference to the current database environment, the page number being read or written, a pointer to the page being read or written, and any parameter **pgcookie** that was specified to the DB_MPOOLFILE->set_pgcookie() (page 428) method.

The **pgin_fcn** and **pgout_fcn** functions should return 0 on success, and a non-zero value on failure, in which case the shared Berkeley DB library function calling it will also fail, returning that non-zero value. The non-zero value should be selected from values outside of the Berkeley DB library namespace.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->memp_stat()

```
#include <db.h>

int
DB_ENV->memp_stat(DB_ENV *env, DB_MPOOL_STAT **gsp,
    DB_MPOOL_FSTAT *(*fsp)[], u_int32_t flags);
```

The `DB_ENV->memp_stat()` method returns the memory pool (that is, the buffer cache) subsystem statistics.

The `DB_ENV->memp_stat()` method creates statistical structures of type `DB_MPOOL_STAT` and `DB_MPOOL_FSTAT`, and copy pointers to them into user-specified memory locations. The cache statistics are stored in the `DB_MPOOL_STAT` structure and the per-file cache statistics are stored the `DB_MPOOL_FSTAT` structure.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

If **gsp** is non-NULL, the global statistics for the cache **mp** are copied into the memory location to which it refers. The following `DB_MPOOL_STAT` fields will be filled in:

- **u_int32_t st_gbytes;**

  Gigabytes of cache (total cache size is st_gbytes + st_bytes).

- **u_int32_t st_bytes;**

  Bytes of cache (total cache size is st_gbytes + st_bytes).

- **u_int32_t st_ncache;**

  Number of caches.

- **u_int32_t st_max_ncache;**

  Maximum number of caches, as configured with the DB_ENV->set_cache_max() (page 392) method.

- **roff_t st_regsize;**

  Individual cache size, in bytes.

- **size_t st_mmapsize;**

  Maximum memory-mapped file size.

- **int st_maxopenfd;**

Maximum open file descriptors.

- **int st_maxwrite;**

  Maximum sequential buffer writes.

- **db_timeout_t st_maxwrite_sleep;**

  Microseconds to pause after writing maximum sequential buffers.

- **u_int32_t st_map;**

  Requested pages mapped into the process' address space (there is no available information about whether or not this request caused disk I/O, although examining the application page fault rate may be helpful).

- **uintmax_t st_cache_hit;**

  Requested pages found in the cache.

- **uintmax_t st_cache_miss;**

  Requested pages not found in the cache.

- **uintmax_t st_page_create;**

  Pages created in the cache.

- **uintmax_t st_page_in;**

  Pages read into the cache.

- **uintmax_t st_page_out;**

  Pages written from the cache to the backing file.

- **uintmax_t st_ro_evict;**

  Clean pages forced from the cache.

- **uintmax_t st_rw_evict;**

  Dirty pages forced from the cache.

- **uintmax_t st_page_trickle;**

  Dirty pages written using the DB_ENV->memp_trickle() (page 391) method.

- **u_int32_t st_pages;**

  Pages in the cache.

- **uintmax_t st_page_clean;**

Clean pages currently in the cache.

- **uintmax_t st_page_dirty;**

  Dirty pages currently in the cache.

- **uintmax_t st_hash_buckets;**

  Number of hash buckets in buffer hash table.

- **uintmax_t st_hash_searches;**

  Total number of buffer hash table lookups.

- **uintmax_t st_hash_longest;**

  Longest chain ever encountered in buffer hash table lookups.

- **uintmax_t st_hash_examined;**

  Total number of hash elements traversed during hash table lookups.

- **uintmax_t st_hash_nowait;**

  Number of times that a thread of control was able to obtain a hash bucket lock without waiting.

- **uintmax_t st_hash_wait;**

  Number of times that a thread of control was forced to wait before obtaining a hash bucket lock.

- **uintmax_t st_hash_max_nowait;**

  The number of times a thread of control was able to obtain the hash bucket lock without waiting on the bucket which had the maximum number of times that a thread of control needed to wait.

- **uintmax_t st_hash_max_wait;**

  Maximum number of times any hash bucket lock was waited for by a thread of control.

- **uintmax_t st_region_wait;**

  Number of times that a thread of control was forced to wait before obtaining a cache region mutex.

- **uintmax_t st_region_nowait;**

  Number of times that a thread of control was able to obtain a cache region mutex without waiting.

- **uintmax_t st_mvcc_frozen;**

Number of buffers frozen.

- **uintmax_t st_mvcc_thawed;**

  Number of buffers thawed.

- **uintmax_t st_mvcc_freed;**

  Number of frozen buffers freed.

- **uintmax_t st_alloc;**

  Number of page allocations.

- **uintmax_t st_alloc_buckets;**

  Number of hash buckets checked during allocation.

- **uintmax_t st_alloc_max_buckets;**

  Maximum number of hash buckets checked during an allocation.

- **uintmax_t st_alloc_pages;**

  Number of pages checked during allocation.

- **uintmax_t st_alloc_max_pages;**

  Maximum number of pages checked during an allocation.

- **uintmax_t st_io_wait;**

  Number of operations blocked waiting for I/O to complete.

- **uintmax_t st_sync_interrupted;**

  Number of mpool sync operations interrupted.

If **fsp** is non-NULL, a pointer to a NULL-terminated variable length array of statistics for individual files, in the cache **mp**, is copied into the memory location to which it refers. If no individual files currently exist in the cache, **fsp** will be set to NULL.

The per-file statistics are stored in structures of type `DB_MPOOL_FSTAT`. The following `DB_MPOOL_FSTAT` fields will be filled in for each file in the cache; that is, each element of the array:

- **char * file_name;**

  The name of the file.

- **size_t st_pagesize;**

  Page size in bytes.

- **uintmax_t st_cache_hit;**

  Requested pages found in the cache.

- **uintmax_t st_cache_miss;**

  Requested pages not found in the cache.

- **u_int32_t st_map;**

  Requested pages mapped into the process' address space.

- **uintmax_t st_page_create;**

  Pages created in the cache.

- **uintmax_t st_page_in;**

  Pages read into the cache.

- **uintmax_t st_page_out;**

  Pages written from the cache to the backing file.

The `DB_ENV->memp_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->memp_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

gsp

The **gsp** parameter references memory into which a pointer to the allocated global statistics structure is copied.

fsp

The **fsp** parameter references memory into which a pointer to the allocated per-file statistics structures is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->memp_stat()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->memp_stat_print()

```
#include <db.h>

int
DB_ENV->memp_stat_print(DB_ENV *env, u_int32_t flags);
```

The DB_ENV->memp_stat_print() method displays cache subsystem statistical information, as described for the DB_ENV->memp_stat() (page 383) method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The DB_ENV->memp_stat_print() method may not be called before the DB_ENV->open() (page 231) method is called.

The DB_ENV->memp_stat_print() method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- DB_STAT_ALL

  Display all available information.

- DB_STAT_CLEAR

  Reset statistics after displaying their values.

- DB_STAT_MEMP_HASH

  Display the buffers with hash chains.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->memp_sync()

```
#include <db.h>

int
DB_ENV->memp_sync(DB_ENV *env, DB_LSN *lsn);
```

The `DB_ENV->memp_sync()` method flushes modified pages in the cache to their backing files.

Pages in the cache that cannot be immediately written back to disk (for example, pages that are currently in use by another thread of control) are waited for and written to disk as soon as it is possible to do so.

The `DB_ENV->memp_sync()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn

The purpose of the **lsn** parameter is to enable a transaction manager to ensure, as part of a checkpoint, that all pages modified by a certain time have been written to disk.

All modified pages with a a log sequence number (DB_LSN) less than the **lsn** parameter are written to disk. If **lsn** is NULL, all modified pages in the cache are written to disk.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->memp_trickle()

```
#include <db.h>

int
DB_ENV->memp_trickle(DB_ENV *env, int percent, int *nwrotep);
```

The `DB_ENV->memp_trickle()` method ensures that a specified percent of the pages in the cache are clean, by writing dirty pages to their backing files.

The purpose of the `DB_ENV->memp_trickle()` function is to enable a memory pool manager to ensure that a page is always available for reading in new information without having to wait for a write.

The `DB_ENV->memp_trickle()` method returns a non-zero error value on failure and 0 on success.

Parameters

percent

The **percent** parameter is the percent of the pages in the cache that should be clean.

nwrotep

The **nwrotep** parameter references memory into which the number of pages written to reach the specified percentage is copied.

Errors

The `DB_ENV->memp_trickle()` method may fail and return one of the following non-zero errors: following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->set_cache_max()

```
#include <db.h>

int
DB_ENV->set_cache_max(DB_ENV *dbenv, u_int32_t gbytes, u_int32_t bytes);
```

Sets the maximum cache size in bytes. The specified size is rounded to the nearest multiple of the cache region size, which is the initial cache size divided by the number of regions specified to the DB_ENV->set_cachesize() (page 394) method. If no value is specified, it defaults to the initial cache size.

The database environment's maximum cache size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_cache_max", one or more whitespace characters, and the size in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_cache_max()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_cache_max()` method may be called at any time during the life of the application.

The `DB_ENV->set_cache_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytes

The **gbytes** parameter specifies the number of bytes which, when added to the **bytes** parameter, specifies the maximum size of the cache.

bytes

The **bytes** parameter specifies the number of bytes which, when added to the **gbytes** parameter, specifies the maximum size of the cache.

Errors

The `DB_ENV->set_cache_max()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_cachesize()

```
#include <db.h>

int
DB_ENV->set_cachesize(DB_ENV *dbenv,
    u_int32_t gbytes, u_int32_t bytes, int ncache);
```

Sets the size of the shared memory buffer pool — that is, the cache. The cache should be the size of the normal working data set of the application, with some small amount of additional memory for unusual situations. (Note: the working set is not the same as the number of pages accessed simultaneously, and is usually much larger.)

The default cache size is 256KB, and may not be specified as less than 20KB. Any cache size less than 500MB is automatically increased by 25% to account for cache overhead; cache sizes larger than 500MB are used as specified. The maximum size of a single cache is 4GB on 32-bit systems and 10TB on 64-bit systems. (All sizes are in powers-of-two, that is, 256KB is 2^18 not 256,000.) For information on tuning the Berkeley DB cache size, see Selecting a cache size.

It is possible to specify caches to Berkeley DB large enough they cannot be allocated contiguously on some architectures. For example, some releases of Solaris limit the amount of memory that may be allocated contiguously by a process. If **ncache** is 0 or 1, the cache will be allocated contiguously in memory. If it is greater than 1, the cache will be split across **ncache** separate regions, where the **region size** is equal to the initial cache size divided by **ncache**.

The cache may be resized by calling `DB_ENV->set_cachesize()` after the environment is open. The supplied size will be rounded to the nearest multiple of the region size and may not be larger than the maximum size configured with DB_ENV->set_cache_max() (page 392). The **ncache** parameter is ignored when resizing the cache.

The database environment's initial cache size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_cachesize", one or more whitespace characters, and the initial cache size specified in three parts: the gigabytes of cache, the additional bytes of cache, and the number of caches, also separated by whitespace characters. For example, "set_cachesize 2 524288000 3" would create a 2.5GB logical cache, split between three physical caches. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_cachesize()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_cachesize()` method may be called at any time during the life of the application.

The `DB_ENV->set_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytes

The size of the cache is set to **gbytes** gigabytes plus **bytes**.

bytes

The size of the cache is set to **gbytes** gigabytes plus **bytes**.

ncache

The **ncache** parameter is the number of caches to create.

Errors

The `DB_ENV->set_cachesize()` method may fail and return one of the following non-zero errors:

EINVAL

If the specified cache size was impossibly small; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Database Environments and Related Methods (page 192)

DB_ENV->set_mp_max_openfd()

```
#include <db.h>

int
DB_ENV->set_mp_max_openfd(DB_ENV *env, int maxopenfd);
```

The `DB_ENV->set_mp_max_openfd()` method limits the number of file descriptors the library will open concurrently when flushing dirty pages from the cache.

The database environment's limit on open file descriptors to flush dirty pages may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_max_openfd", one or more whitespace characters, and the number of open file descriptors. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DB_ENV->set_mp_max_openfd() (page 396) method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_mp_max_openfd()` method returns a non-zero error value on failure and 0 on success.

Parameters

maxopenfd

The maximum number of file descriptors that may be concurrently opened by the library when flushing dirty pages from the cache.

Errors

The `DB_ENV->set_mp_max_openfd()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->set_mp_max_write()

```
#include <db.h>

int
DB_ENV->set_mp_max_write(DB_ENV *env, int maxwrite,
                         db_timeout_t maxwrite_sleep);
```

The `DB_ENV->set_mp_max_write()` method limits the number of sequential write operations scheduled by the library when flushing dirty pages from the cache.

The database environment's maximum number of sequential write operations may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_mp_max_write", one or more whitespace characters, and the maximum number of sequential writes and the number of microseconds to sleep, also separated by whitespace characters. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_mp_max_write()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_mp_max_write()` method returns a non-zero error value on failure and 0 on success.

Parameters

maxwrite

The maximum number of sequential write operations scheduled by the library when flushing dirty pages from the cache, or 0 if there is no limitation on the number of sequential write operations.

maxwrite_sleep

The number of microseconds the thread of control should pause before scheduling further write operations. It must be specified as an unsigned 32-bit number of microseconds, limiting the maximum pause to roughly 71 minutes.

Errors

The `DB_ENV->set_mp_max_write()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->set_mp_mmapsize()

```
#include <db.h>

int
DB_ENV->set_mp_mmapsize(DB_ENV *dbenv, size_t mp_mmapsize);
```

Files that are opened read-only in the cache (and that satisfy a few other criteria) are, by default, mapped into the process address space instead of being copied into the local cache. This can result in better-than-usual performance because available virtual memory is normally much larger than the local cache, and page faults are faster than page copying on many systems. However, it can cause resource starvation in the presence of limited virtual memory, and it can result in immense process sizes in the presence of large databases.

The `DB_ENV->set_mp_mmapsize()` method sets the maximum file size, in bytes, for a file to be mapped into the process address space. If no value is specified, it defaults to 10MB.

The database environment's maximum mapped file size may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_mp_mmapsize", one or more whitespace characters, and the size in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_mp_mmapsize()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_mp_mmapsize()` method may be called at any time during the life of the application.

The `DB_ENV->set_mp_mmapsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

mp_mmapsize

The **mp_mmapsize** parameter is the maximum file size, in bytes, for a file to be mapped into the process address space.

Errors

The `DB_ENV->set_mp_mmapsize()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_ENV->set_mp_pagesize()

```
#include <db.h>

int
DB_ENV->set_mp_pagesize(DB_ENV *dbenv, u_int32_t pagesize);
```

The `DB_ENV->set_mp_pagesize()` method sets the pagesize used to allocate the hash table and the number of mutexes expected to be needed by the buffer pool.

You must call this method only before the environment is opened.

Parameters

pagesize

The pagesize parameter specifies expected page size use. Generally, it is set to the expected average page size for all the data pages that are in the buffer pool.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->get_mp_pagesize() (page 378)

DB_ENV->set_mp_tablesize()

```
#include <db.h>

int
DB_ENV->set_mp_tablesize(DB_ENV *dbenv, u_int32_t tablesize);
```

The `DB_ENV->set_mp_tablesize()` method overrides the calculated hash tablesize. Tablesize is adjusted to a near prime number to enhance the hashing algorithm.

You must call this method only before the environment is opened.

Parameters

tablesize

The tablesize parameter specifies the size of the buffer pool hash table. It is adjusted to a near prime number to enhance the hashing algorithm.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_ENV->get_mp_tablesize() (page 379)

DB_MPOOLFILE->close()

```
#include <db.h>

int
DB_MPOOLFILE->close(DB_MPOOLFILE *mpf, u_int32_t flags);
```

The `DB_MPOOLFILE->close()` method closes the source file indicated by the DB_MPOOLFILE structure. Calling `DB_MPOOLFILE->close()` does not imply a call to DB_MPOOLFILE->sync() (page 411); that is, no pages are written to the source file as as a result of calling `DB_MPOOLFILE->close.()`.

If the DB_MPOOLFILE was temporary, any underlying files created for this DB_MPOOLFILE will be removed.

After `DB_MPOOLFILE->close()` has been called, regardless of its return, the DB_MPOOLFILE handle may not be accessed again.

The `DB_MPOOLFILE->close()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->get()

```
#include <db.h>

int
DB_MPOOLFILE->get(DB_MPOOLFILE *mpf,
    db_pgno_t *pgnoaddr, DB_TXN * txnid, u_int32_t flags, void **pagep);
```

The `DB_MPOOLFILE->get()` method returns pages from the cache.

All pages returned by `DB_MPOOLFILE->get()` will be retained (that is, *latched*) in the cache until a subsequent call to DB_MPOOLFILE->put() (page 409). There is no deadlock detection among latches so care must be taken in the application if the DB_MPOOL_DIRTY or DB_MPOOL_EDIT flags are used as these get exlusive latches on the pages.

The returned page is **size_t** type aligned.

Fully or partially created pages have all their bytes set to a nul byte, unless the DB_MPOOLFILE->set_clear_len() (page 420) method was called to specify other behavior before the file was opened.

The `DB_MPOOLFILE->get()` method will return `DB_PAGE_NOTFOUND` if the requested page does not exist and DB_MPOOL_CREATE was not set. Unless otherwise specified, the `DB_MPOOLFILE->get()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_MPOOL_CREATE`

  If the specified page does not exist, create it. In this case, the pgin method, if specified, is called.

- `DB_MPOOL_DIRTY`

  The page will be modified and must be written to the source file before being evicted from the cache. For files open with the DB_MULTIVERSION flag set, a new copy of the page will be made if this is the first time the specified transaction is modifying it. A page fetched with the `DB_MPOOL_DIRTY` flag will be **exclusively latched** until a subsequent call to DB_MPOOLFILE->put() (page 409).

- `DB_MPOOL_EDIT`

  The page will be modified and must be written to the source file before being evicted from the cache. No copy of the page will be made, regardless of the DB_MULTIVERSION setting. This flag is only intended for use in situations where a transaction handle is not available, such as during aborts or recovery. A page fetched with the DB_MPOOL_EDIT flag will be **exclusively latched** until a subsequent call to DB_MPOOLFILE->put() (page 409).

- DB_MPOOL_LAST

  Return the last page of the source file, and copy its page number into the memory location to which **pgnoaddr** refers.

- DB_MPOOL_NEW

  Create a new page in the file, and copy its page number into the memory location to which **pgnoaddr** refers. In this case, the pgin_fcn callback, if specified on DB_ENV->memp_register() (page 381), is **not** called.

The DB_MPOOL_CREATE, DB_MPOOL_LAST, and DB_MPOOL_NEW flags are mutually exclusive.

pagep

The **pagep** parameter references memory into which a pointer to the returned page is copied.

pgnoaddr

If the **flags** parameter is set to DB_MPOOL_LAST or DB_MPOOL_NEW, the page number of the created page is copied into the memory location to which the **pgnoaddr** parameter refers. Otherwise, the **pgnoaddr** parameter is the page to create or retrieve.

Note

Page numbers begin at 0; that is, the first page in the file is page number 0, not page number 1.

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); otherwise NULL. A transaction is required if the file is open for multiversion concurrency control by passing DB_MULTIVERSION to DB_MPOOLFILE->open() (page 407) and the DB_MPOOL_DIRTY, DB_MPOOL_CREATE or DB_MPOOL_NEW flags were specified. Otherwise it is ignored.

Errors

The DB_MPOOLFILE->get() method may fail and return one of the following non-zero errors:

EACCES

The DB_MPOOL_DIRTY or DB_MPOOL_EDIT flag was set and the source file was not opened for writing.

EAGAIN

The page reference count has overflowed. (This should never happen unless there is a bug in the application.)

EINVAL

If the DB_MPOOL_NEW flag was set, and the source file was not opened for writing; more than one of DB_MPOOL_CREATE, DB_MPOOL_LAST, and DB_MPOOL_NEW was set; or if an invalid flag value or parameter was specified.

DB_LOCK_DEADLOCK

For transactions configured with DB_TXN_SNAPSHOT, the page has been modified since the transaction began.

ENOMEM

The cache is full, and no more pages will fit in the cache.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->open()

```
#include <db.h>

int
DB_MPOOLFILE->open(DB_MPOOLFILE *mpf,
    char *file, u_int32_t flags, int mode, size_t pagesize);
```

The `DB_MPOOLFILE->open()` method opens a file in the in-memory cache.

The `DB_MPOOLFILE->open()` method returns a non-zero error value on failure and 0 on success.

Parameters

file

The **file** parameter is the name of the file to be opened. If **file** is NULL, a private temporary file is created that cannot be shared with any other process (although it may be shared with other threads of control in the same process).

When using a Unicode build on Windows (the default), the **file** argument will be interpreted as a UTF-8 string, which is equivalent to ASCII for Latin characters.

flags

The **flags** parameter must be set to zero or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_CREATE`

  Create any underlying files, as necessary. If the database do not already exist and the `DB_CREATE` flag is not specified, the call will fail.

- `DB_DIRECT`

  If set and supported by the system, turn off system buffering of the file to avoid double caching.

- `DB_MULTIVERSION`

  Open the file with support for multiversion concurrency control. Calls to DB_MPOOLFILE->get() (page 404) with dirty pages will cause copies to be made in the cache.

- `DB_NOMMAP`

  Always copy this file into the local cache instead of potentially mapping it into process memory (see the DB_ENV->set_mp_mmapsize() (page 399) method for further information).

- `DB_ODDFILESIZE`

Attempts to open files which are not a multiple of the page size in length will fail, by default. If the DB_ODDFILESIZE flag is set, any partial page at the end of the file will be ignored and the open will proceed.

- DB_RDONLY

  Open any underlying files for reading only. Any attempt to modify the file using the memory pool (cache) functions will fail, regardless of the actual permissions of the file.

mode

On Windows systems, the mode parameter is ignored.

On UNIX systems or in IEEE/ANSI Std 1003.1 (POSIX) environments, files created by `DB_MPOOLFILE->open()` are created with mode **mode** (as described in **chmod**(2)) and modified by the process' umask value at the time of creation (see **umask**(2)). Created files are owned by the process owner; the group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB. System shared memory segments created by `DB_MPOOLFILE->open()` are created with mode **mode**, unmodified by the process' umask value. If **mode** is 0, `DB_MPOOLFILE->open()` will use a default mode of readable and writable by both owner and group.

pagesize

The **pagesize** parameter is the size, in bytes, of the unit of transfer between the application and the cache, although it is not necessarily the unit of transfer between the cache and the underlying filesystem.

Errors

The `DB_MPOOLFILE->open()` method may fail and return one of the following non-zero errors:

EINVAL

If the file has already been entered into the cache, and the **pagesize** value is not the same as when the file was entered into the cache, or the length of the file is not zero or a multiple of the **pagesize**; the DB_RDONLY flag was specified for an in-memory cache; or if an invalid flag value or parameter was specified.

ENOMEM

The maximum number of open files has been reached.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->put()

```
#include <db.h>

int
DB_MPOOLFILE->put(DB_MPOOLFILE *mpf,
    void *pgaddr, DB_CACHE_PRIORITY priority, u_int32_t flags);
```

The `DB_MPOOLFILE->put()` method returns a reference to a page in the cache, setting the priority of the page as specified by the **priority** parameter.

The `DB_MPOOLFILE->put()` method returns a non-zero error value on failure and 0 on success.

Parameters

pgaddr

The **pgaddr** parameter is the address of the page to be returned to the cache. The **pgaddr** parameter must be a value previously returned by the DB_MPOOLFILE->get() (page 404) method.

priority

Set the page's **priority** as follows:

- `DB_PRIORITY_UNCHANGED`

  The priority is unchanged.

- `DB_PRIORITY_VERY_LOW`

  The lowest priority: pages are the most likely to be discarded.

- `DB_PRIORITY_LOW`

  The next lowest priority.

- `DB_PRIORITY_DEFAULT`

  The default priority.

- `DB_PRIORITY_HIGH`

  The next highest priority.

- `DB_PRIORITY_VERY_HIGH`

  The highest priority: pages are the least likely to be discarded.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_MPOOLFILE->put()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->sync()

```
#include <db.h>

int
DB_MPOOLFILE->sync(DB_MPOOLFILE *mpf);
```

The `DB_MPOOLFILE->sync()` method writes all modified pages associated with the DB_MPOOLFILE back to the source file. If any of the modified pages are *pinned* (that is, currently in use), `DB_MPOOLFILE->sync()` will ignore them.

The `DB_MPOOLFILE->sync()` method returns a non-zero error value on failure and 0 on success.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->get_clear_len()

```
#include <db.h>

int
DB_MPOOLFILE->get_clear_len(DB_MPOOLFILE *mpf, u_int32_t *lenp);
```

The `DB_MPOOLFILE->get_clear_len()` method returns the bytes to be cleared.

The `DB_MPOOLFILE->get_clear_len()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_clear_len()` method returns a non-zero error value on failure and 0 on success.

Parameters

lenp

The `DB_MPOOLFILE->get_clear_len()` method returns the bytes to be cleared in **lenp**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->get_fileid()

```
#include <db.h>

int
DB_MPOOLFILE->get_fileid(DB_MPOOLFILE *mpf, u_int8_t *fileid);
```

The `DB_MPOOLFILE->get_fileid()` method copies the file's identifier into the memory location referenced by **fileid**. The fileid specifies a unique identifier for the file, which is used so that the cache functions (that is, the shared memory buffer pool functions) are able to uniquely identify files. This is necessary for multiple processes wanting to share a file to correctly identify the file in the cache.

The `DB_MPOOLFILE->get_fileid()` method returns a non-zero error value on failure and 0 on success.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_fileid() (page 421)

DB_MPOOLFILE->get_flags()

```
#include <db.h>

int
DB_MPOOLFILE->get_flags(DB_MPOOLFILE *mpf, u_int32_t *flagsp);
```

The `DB_MPOOLFILE->get_flags()` method returns the flags used to configure a file in the cache.

The `DB_MPOOLFILE->get_flags()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB_MPOOLFILE->get_flags()` method returns the flags in **flagsp**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_flags() (page 423)

DB_MPOOLFILE->get_ftype()

```
#include <db.h>

int
DB_MPOOLFILE->get_ftype(DB_MPOOLFILE *mpf, int *ftypep);
```

The `DB_MPOOLFILE->get_ftype()` method returns the file type. The file type is used for the purposes of file processing, and will be the same as is set using the DB_ENV->memp_register() (page 381) method.

The `DB_MPOOLFILE->get_ftype()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_ftype()` method returns a non-zero error value on failure and 0 on success.

Parameters

ftypep

The `DB_MPOOLFILE->get_ftype()` method returns the file type in **ftypep**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_ftype() (page 425)

DB_MPOOLFILE->get_lsn_offset()

```
#include <db.h>

int
DB_MPOOLFILE->get_lsn_offset(DB_MPOOLFILE *mpf, int32_t *lsn_offsetp);
```

The `DB_MPOOLFILE->get_lsn_offset()` method returns the log sequence number byte offset configured for a file's pages using the DB_MPOOLFILE->set_lsn_offset() (page 426) method.

The `DB_MPOOLFILE->get_lsn_offset()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_lsn_offset()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn_offsetp

The `DB_MPOOLFILE->get_lsn_offset()` method returns the log sequence number byte offset in **lsn_offsetp**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_lsn_offset() (page 426)

DB_MPOOLFILE->get_maxsize()

```
#include <db.h>

int
DB_MPOOLFILE->get_maxsize(DB_MPOOLFILE *mpf,
    u_int32_t *gbytesp, u_int32_t *bytesp);
```

Returns the maximum size configured for the file, as configured using the DB_MPOOLFILE->set_maxsize() (page 427) method.

The `DB_MPOOLFILE->get_maxsize()` method returns a non-zero error value on failure and 0 on success.

The `DB_MPOOLFILE->get_maxsize()` method may be called at any time during the life of the application.

Parameters

gbytesp

The **gbytesp** parameter references memory into which the gigabytes of memory in the maximum file size is copied.

bytesp

The **bytesp** parameter references memory into which the additional bytes of memory in the maximum file size is copied.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_maxsize() (page 427)

DB_MPOOLFILE->get_pgcookie()

```
#include <db.h>

int
DB_MPOOLFILE->get_pgcookie(DB_MPOOLFILE *mpf, DBT *dbt);
```

The `DB_MPOOLFILE->get_pgcookie()` method returns the byte string provided to the functions registered to do input or output processing of the file's pages as they are read from or written to, the backing filesystem store. This byte string is configured using the DB_MPOOLFILE->set_pgcookie() (page 428) method.

The `DB_MPOOLFILE->get_pgcookie()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_pgcookie()` method returns a non-zero error value on failure and 0 on success.

Parameters

dbt

The `DB_MPOOLFILE->get_pgcookie()` method returns a reference to the byte string in **dbt**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_pgcookie() (page 428)

DB_MPOOLFILE->get_priority()

```
#include <db.h>

int
DB_MPOOLFILE->get_priority(DB_MPOOLFILE *mpf,
                           DB_CACHE_PRIORITY *priorityp);
```

The `DB_MPOOLFILE->get_priority()` method returns the cache priority for the file referenced by the DB_MPOOLFILE handle. The priority of a page biases the replacement algorithm to be more or less likely to discard a page when space is needed in the cache. This value is set using the DB_MPOOLFILE->set_priority() (page 429) method.

The `DB_MPOOLFILE->get_priority()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->get_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priorityp

The `DB_MPOOLFILE->get_priority()` method returns a reference to the cache priority for the file referenced by the DB_MPOOLFILE handle in **priorityp**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370), DB_MPOOLFILE->set_priority() (page 429)

DB_MPOOLFILE->set_clear_len()

```
#include <db.h>

int
DB_MPOOLFILE->set_clear_len(DB_MPOOLFILE *mpf, u_int32_t len);
```

The `DB_MPOOLFILE->set_clear_len()` method sets the number of initial bytes in a page that should be set to nul when the page is created as a result of the DB_MPOOL_CREATE or DB_MPOOL_NEW flags specified to DB_MPOOLFILE->get() (page 404). If no clear length is specified, the entire page is cleared when it is created.

The `DB_MPOOLFILE->set_clear_len()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_clear_len()` method may not be called after the DB_MPOOLFILE->open() (page 407) method is called. If the file is already open in the cache when DB_MPOOLFILE->open() (page 407) is called, the information specified to `DB_MPOOLFILE->set_clear_len()` must be consistent with the existing file or an error will be returned.

The `DB_MPOOLFILE->set_clear_len()` method returns a non-zero error value on failure and 0 on success.

Parameters

len

The **len** parameter is the number of initial bytes in a page that should be set to nul when the page is created. A value of 0 results in the entire page being set to nul bytes.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_fileid()

```
#include <db.h>

int
DB_MPOOLFILE->set_fileid(DB_MPOOLFILE *mpf, u_int8_t *fileid);
```

The `DB_MPOOLFILE->set_fileid()` method specifies a unique identifier for the file. (The shared memory buffer pool functions must be able to uniquely identify files in order that multiple processes wanting to share a file will correctly identify it in the cache.)

On most UNIX/POSIX systems, the **fileid** field will not need to be set, and the memory pool functions will use the file's device and inode numbers for this purpose. On Windows systems, the memory pool functions use the values returned by `GetFileInformationByHandle()` by default — these values are known to be constant between processes and over reboot in the case of NTFS (in which they are the NTFS MFT indices).

On other filesystems (for example, FAT or NFS), these default values are not necessarily unique between processes or across system reboots. **Applications wanting to maintain a shared cache between processes or across system reboots, in which the cache contains pages from files stored on such filesystems, must specify a unique file identifier using the `DB_MPOOLFILE->set_fileid()` method, and each process opening the file must provide the same unique identifier.**

This call should not be necessary for most applications. Specifically, it is not necessary if the cache is not shared between processes and is reinstantiated after each system reboot, if the application is using the Berkeley DB access methods instead of calling the pool functions explicitly, or if the files in the cache are stored on filesystems in which the default values as described previously are invariant between process and across system reboots.

The `DB_MPOOLFILE->set_fileid()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_fileid()` method may not be called after the DB_MPOOLFILE->open() (page 407) method is called.

The `DB_MPOOLFILE->set_fileid()` method returns a non-zero error value on failure and 0 on success.

Parameters

fileid

The **fileid** parameter is the unique identifier for the file. Unique file identifiers must be a `DB_FILE_ID_LEN` length array of bytes.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_flags()

```
#include <db.h>

int
DB_MPOOLFILE->set_flags(DB_MPOOLFILE *mpf, u_int32_t flags, int onoff)
```

Configure a file in the cache.

To set the flags for a particular database, call the `DB_MPOOLFILE->set_flags()` method using the DB_MPOOLFILE handle stored in the **mpf** field of the DB handle.

The `DB_MPOOLFILE->set_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_MPOOL_NOFILE`

  If set, no backing temporary file will be opened for the specified in-memory database, even if it expands to fill the entire cache. Attempts to create new database pages after the cache has been filled will fail.

  The `DB_MPOOL_NOFILE` flag configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

  The `DB_MPOOL_NOFILE` flag may be used to configure Berkeley DB at any time during the life of the application.

- `DB_MPOOL_UNLINK`

  If set, remove the file when the last reference to it is closed.

  The `DB_MPOOL_ULINK` flag configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

  The `DB_MPOOL_ULINK` flag may be used to configure Berkeley DB at any time during the life of the application.

onoff

If **onoff** is zero, the specified flags are cleared; otherwise they are set.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_ftype()

```
#include <db.h>

int
DB_MPOOLFILE->set_ftype(DB_MPOOLFILE *mpf, int ftype);
```

The `DB_MPOOLFILE->set_ftype()` method specifies a file type for the purposes of input or output processing of the file's pages as they are read from or written to, the backing filesystem store.

The `DB_MPOOLFILE->set_ftype()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_ftype()` method may not be called after the DB_MPOOLFILE->open() (page 407) method is called. If the file is already open in the cache when DB_MPOOLFILE->open() (page 407) is called, the information specified to `DB_MPOOLFILE->set_ftype()` will replace the existing information.

The `DB_MPOOLFILE->set_ftype()` method returns a non-zero error value on failure and 0 on success.

Parameters

ftype

The **ftype** parameter sets the file's type for the purposes of input and output processing. The **ftype** must be the same as a **ftype** parameter previously specified to the DB_ENV->memp_register() (page 381) method.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_lsn_offset()

```
#include <db.h>

int
DB_MPOOLFILE->set_lsn_offset(DB_MPOOLFILE *mpf, int32_t lsn_offset);
```

The `DB_MPOOLFILE->set_lsn_offset()` method specifies the zero-based byte offset of a log sequence number (DB_LSN) on the file's pages, for the purposes of page-flushing as part of transaction checkpoint. (See the DB_ENV->memp_sync() (page 390) documentation for more information.)

The `DB_MPOOLFILE->set_lsn_offset()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_lsn_offset()` method may not be called after the DB_MPOOLFILE->open() (page 407) method is called. If the file is already open in the cache when DB_MPOOLFILE->open() (page 407) is called, the information specified to `DB_MPOOLFILE->set_lsn_offset()` must be consistent with the existing file or an error will be returned.

The `DB_MPOOLFILE->set_lsn_offset()` method returns a non-zero error value on failure and 0 on success.

Parameters

lsn_offset

The **lsn_offset** parameter is the zero-based byte offset of the log sequence number on the file's pages.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_maxsize()

```
#include <db.h>

int
DB_MPOOLFILE->set_maxsize(DB_MPOOLFILE *mpf,
    u_int32_t gbytes, u_int32_t bytes);
```

Set the maximum size for the file to be **gbytes** gigabytes plus **bytes**. Attempts to allocate new pages in the file after the limit has been reached will fail.

To set the maximum file size for a particular database, call the `DB_MPOOLFILE->set_maxsize()` method using the DB_MPOOLFILE handle stored in the **mpf** field of the DB handle. Attempts to insert new items into the database after the limit has been reached may fail.

The `DB_MPOOLFILE->set_maxsize()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_maxsize()` method may be called at any time during the life of the application.

The `DB_MPOOLFILE->set_maxsize()` method returns a non-zero error value on failure and 0 on success.

Parameters

bytes

The maximum size of the file is set to **gbytes** gigabytes plus **bytes**.

gbytes

The maximum size of the file is set to **gbytes** gigabytes plus **bytes**.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_pgcookie()

```
#include <db.h>

int
DB_MPOOLFILE->set_pgcookie(DB_MPOOLFILE *mpf, DBT *pgcookie);
```

The `DB_MPOOLFILE->set_pgcookie()` method specifies a byte string that is provided to the functions registered to do input or output processing of the file's pages as they are read from or written to, the backing filesystem store. (See the DB_ENV->memp_register() (page 381) documentation for more information.)

The `DB_MPOOLFILE->set_pgcookie()` method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The `DB_MPOOLFILE->set_pgcookie()` method may not be called after the DB_MPOOLFILE->open() (page 407) method is called. If the file is already open in the cache when DB_MPOOLFILE->open() (page 407) is called, the information specified to `DB_MPOOLFILE->set_pgcookie()` will replace the existing information.

The `DB_MPOOLFILE->set_pgcookie()` method returns a non-zero error value on failure and 0 on success.

Parameters

pgcookie

The **pgcookie** parameter is a byte string provided to the functions registered to do input or output processing of the file's pages.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

DB_MPOOLFILE->set_priority()

```
#include <db.h>

int
DB_MPOOLFILE->set_priority(DB_MPOOLFILE *mpf, DB_CACHE_PRIORITY priority);
```

Set the cache priority for pages referenced by the DB_MPOOLFILE handle.

The priority of a page biases the replacement algorithm to be more or less likely to discard a page when space is needed in the cache. The bias is temporary, and pages will eventually be discarded if they are not referenced again. The DB_MPOOLFILE->set_priority() method is only advisory, and does not guarantee pages will be treated in a specific way.

To set the priority for the pages belonging to a particular database, call the DB_MPOOLFILE->set_priority() method using the DB_MPOOLFILE handle returned by the DB->get_mpf() (page 372) method.

The DB_MPOOLFILE->set_priority() method configures a file in the cache, not only operations performed using the specified DB_MPOOLFILE handle.

The DB_MPOOLFILE->set_priority() method may be called at any time during the life of the application.

The DB_MPOOLFILE->set_priority() method returns a non-zero error value on failure and 0 on success.

Parameters

priority

The **priority** parameter must be set to one of the following values:

- DB_PRIORITY_VERY_LOW

  The lowest priority: pages are the most likely to be discarded.

- DB_PRIORITY_LOW

  The next lowest priority.

- DB_PRIORITY_DEFAULT

  The default priority.

- DB_PRIORITY_HIGH

  The next highest priority.

- DB_PRIORITY_VERY_HIGH

  The highest priority: pages are the least likely to be discarded.

Class

DB_ENV, DB_MPOOLFILE

See Also

Memory Pools and Related Methods (page 370)

Chapter 9. Mutex Methods

This chapter describes methods that can be used to manage mutexes within DB. Many of the methods described here are used to configure DB's internal mutex system. However, a series of APIs are available for use as a general-purpose, cross platform mutex management system. These methods can be used independently of DB's main purpose, which is as a high-end data management engine.

Mutex Methods

Mutexes and Related Methods	Description
DB_ENV->mutex_alloc()	Allocate a mutex
DB_ENV->mutex_free()	Free a mutex
DB_ENV->mutex_lock()	Lock a mutex
DB_ENV->mutex_stat()	Mutex statistics
DB_ENV->mutex_stat_print()	Print mutex statistics
DB_ENV->mutex_unlock()	Unlock a mutex
Mutex Configuration	
DB_ENV->mutex_set_align(), DB_ENV->mutex_get_align()	Configure mutex alignment
DB_ENV->mutex_set_increment(), DB_ENV->mutex_get_increment()	Configure number of additional mutexes
DB_ENV->mutex_set_max(), DB_ENV->mutex_get_max()	Configure total number of mutexes
DB_ENV->mutex_set_tas_spins(), DB_ENV->mutex_get_tas_spins()	Configure test-and-set mutex spin count

DB_ENV->mutex_alloc()

```
#include <db.h>

int
DB_ENV->mutex_alloc(DB_ENV *dbenv, u_int32_t flags, db_mutex_t *mutexp);
```

The `DB_ENV->mutex_alloc()` method allocates a mutex and returns a reference to it into the memory specified by **mutexp**.

The `DB_ENV->mutex_alloc()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->mutex_alloc()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_MUTEX_PROCESS_ONLY`

  The mutex is associated with a single process. The DB_ENV->failchk() (page 206) method will release mutexes held by any process which has exited.

- `DB_MUTEX_SELF_BLOCK`

  The mutex must be self-blocking. That is, if a thread of control locks the mutex and then attempts to lock the mutex again, the thread of control will block until another thread of control releases the original lock on the mutex, allowing the original thread of control to lock the mutex the second time. Attempting to re-acquire a mutex for which the `DB_MUTEX_SELF_BLOCK` flag was not specified will result in undefined behavior.

mutexp

The **mutexp** parameter references memory into which the mutex reference is copied.

Errors

The `DB_ENV->mutex_alloc()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_free()

```
#include <db.h>

int
DB_ENV->mutex_free(DB_ENV *dbenv, db_mutex_t mutex);
```

The `DB_ENV->mutex_free()` method discards a mutex allocated by DB_ENV->mutex_alloc() (page 433).

The `DB_ENV->mutex_free()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->mutex_free()` method returns a non-zero error value on failure and 0 on success.

Parameters

mutex

The **mutex** parameter is a mutex previously allocated by DB_ENV->mutex_alloc() (page 433).

Errors

The `DB_ENV->mutex_free()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_get_align()

```
#include <db.h>

int
DB_ENV->mutex_get_align(DB_ENV *dbenv, u_int32_t *alignp);
```

The `DB_ENV->mutex_get_align()` method returns the mutex alignment, in bytes.

The `DB_ENV->mutex_get_align()` method may be called at any time during the life of the application.

The `DB_ENV->mutex_get_align()` method returns a non-zero error value on failure and 0 on success.

Parameters

alignp

The `DB_ENV->mutex_get_align()` method returns the mutex alignment, in bytes in **alignp**.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_get_increment()

```
#include <db.h>

int
DB_ENV->mutex_get_increment(DB_ENV *dbenv, u_int32_t *incrementp);
```

The `DB_ENV->mutex_get_increment()` method returns the number of additional mutexes to allocate.

The `DB_ENV->mutex_get_increment()` method may be called at any time during the life of the application.

The `DB_ENV->mutex_get_increment()` method returns a non-zero error value on failure and 0 on success.

Parameters

incrementp

The `DB_ENV->mutex_get_increment()` method returns the number of additional mutexes to allocate in **incrementp**.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_get_max()

```
#include <db.h>

int
DB_ENV->mutex_get_max(DB_ENV *dbenv, u_int32_t *maxp);
```

The `DB_ENV->mutex_get_max()` method returns the total number of mutexes allocated.

The `DB_ENV->mutex_get_max()` method may be called at any time during the life of the application.

The `DB_ENV->mutex_get_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

maxp

The `DB_ENV->mutex_get_max()` method returns the total number of mutexes allocated in **maxp**.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_get_tas_spins()

```
#include <db.h>

int
DB_ENV->mutex_get_tas_spins(DB_ENV *dbenv, u_int32_t *tas_spinsp);
```

The `DB_ENV->mutex_get_tas_spins()` method returns the test-and-set spin count. This value may be configured using the DB_ENV->mutex_set_tas_spins() (page 446) method.

The `DB_ENV->mutex_get_tas_spins()` method may be called at any time during the life of the application.

The `DB_ENV->mutex_get_tas_spins()` method returns a non-zero error value on failure and 0 on success.

Parameters

tas_spinsp

The `DB_ENV->mutex_get_tas_spins()` method returns the test-and-set spin count in **tas_spinsp**.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_lock()

```
#include <db.h>

int
DB_ENV->mutex_lock(DB_ENV *dbenv, db_mutex_t mutex);
```

The `DB_ENV->mutex_lock()` method locks the mutex allocated by DB_ENV->mutex_alloc() (page 433). The thread of control calling `DB_ENV->mutex_lock()` will block until the lock is available.

The `DB_ENV->mutex_lock()` method returns a non-zero error value on failure and 0 on success.

Parameters

mutex

The **mutex** parameter is a mutex previously allocated by DB_ENV->mutex_alloc() (page 433).

Errors

The `DB_ENV->mutex_lock()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_set_align()

```
#include <db.h>

int
DB_ENV->mutex_set_align(DB_ENV *dbenv, u_int32_t align);
```

Set the mutex alignment, in bytes.

It is sometimes advantageous to align mutexes on specific byte boundaries in order to minimize cache line collisions. The `DB_ENV->mutex_set_align()` method specifies an alignment for mutexes allocated by Berkeley DB.

The database environment's mutex alignment may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "mutex_set_align", one or more whitespace characters, and the mutex alignment in bytes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->mutex_set_align()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->mutex_set_align()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->mutex_set_align()` will be ignored.

The `DB_ENV->mutex_set_align()` method returns a non-zero error value on failure and 0 on success.

Parameters

align

The **align** parameter is the mutex alignment, in bytes. The mutex alignment must be a power-of-two.

Errors

The `DB_ENV->mutex_set_align()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_set_increment()

```
#include <db.h>

int
DB_ENV->mutex_set_increment(DB_ENV *dbenv, u_int32_t increment);
```

Configure the number of additional mutexes to allocate.

If an application will allocate mutexes for its own use, the DB_ENV->mutex_set_increment() method is used to add a number of mutexes to the default allocation.

Calling the DB_ENV->mutex_set_increment() method discards any value previously set using the DB_ENV->mutex_set_max() method.

The database environment's number of additional mutexes may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "mutex_set_increment", one or more whitespace characters, and the number of additional mutexes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DB_ENV->mutex_set_increment() method configures a database environment, not only operations performed using the specified DB_ENV handle.

The DB_ENV->mutex_set_increment() method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to DB_ENV->mutex_set_increment() will be ignored.

The DB_ENV->mutex_set_increment() method returns a non-zero error value on failure and 0 on success.

Parameters

increment

The **increment** parameter is the number of additional mutexes to allocate.

Errors

The DB_ENV->mutex_set_increment() method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_set_max()

```
#include <db.h>

int
DB_ENV->mutex_set_max(DB_ENV *dbenv, u_int32_t max);
```

Configure the total number of mutexes to allocate.

Berkeley DB allocates a default number of mutexes based on the initial configuration of the database environment. That default calculation may be too small if the application has an unusual need for mutexes (for example, if the application opens an unexpectedly large number of databases) or too large (if the application is trying to minimize its memory footprint). The `DB_ENV->mutex_set_max()` method is used to specify an absolute number of mutexes to allocate.

Calling the `DB_ENV->mutex_set_max()` method discards any value previously set using the `DB_ENV->mutex_set_increment()` method.

The database environment's total number of mutexes may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "mutex_set_max", one or more whitespace characters, and the total number of mutexes. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->mutex_set_max()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->mutex_set_max()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->mutex_set_max()` will be ignored.

The `DB_ENV->mutex_set_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

max

The **max** parameter is the absolute number of mutexes to allocate.

Errors

The `DB_ENV->mutex_set_max()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_set_tas_spins()

```
#include <db.h>

int
DB_ENV->mutex_set_tas_spins(DB_ENV *dbenv, u_int32_t tas_spins);
```

Specify that test-and-set mutexes should spin **tas_spins** times without blocking. The value defaults to 1 on uniprocessor systems and to 50 times the number of processors on multiprocessor systems.

The database environment's test-and-set spin count may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_tas_spins", one or more whitespace characters, and the number of spins. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->mutex_set_tas_spins()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->mutex_set_tas_spins()` method may be called at any time during the life of the application.

The `DB_ENV->mutex_set_tas_spins()` method returns a non-zero error value on failure and 0 on success.

Parameters

tas_spins

The **tas_spins** parameter is the number of spins test-and-set mutexes should execute before blocking.

Errors

The `DB_ENV->mutex_set_tas_spins()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_stat()

```
#include <db.h>

int
DB_ENV->mutex_stat(DB_ENV *env, DB_MUTEX_STAT **statp, u_int32_t flags);
```

The `DB_ENV->mutex_stat()` method returns the mutex subsystem statistics.

The `DB_ENV->mutex_stat()` method creates a statistical structure of type `DB_MUTEX_STAT` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following `DB_MUTEX_STAT` fields will be filled in:

- **u_int32_t st_mutex_align;**

  The mutex alignment, in bytes.

- **u_int32_t st_mutex_tas_spins;**

  The number of times test-and-set mutexes will spin without blocking.

- **int st_mutex_cnt;**

  The total number of mutexes configured.

- **u_int32_t st_mutex_free;**

  The number of mutexes currently available.

- **u_int32_t st_mutex_inuse;**

  The number of mutexes currently in use.

- **u_int32_t st_mutex_inuse_max;**

  The maximum number of mutexes ever in use.

- **roff_t st_regsize;**

  The size of the mutex region, in bytes.

- **uintmax_t st_region_wait;**

  The number of times that a thread of control was forced to wait before obtaining the mutex region mutex.

- **uintmax_t st_region_nowait;**

  The number of times that a thread of control was able to obtain the mutex region mutex without waiting.

The `DB_ENV->mutex_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->mutex_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->mutex_stat()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_stat_print()

```
#include <db.h>

int
DB_ENV->mutex_stat_print(DB_ENV *env, u_int32_t flags);
```

The `DB_ENV->mutex_stat_print()` method displays the mutex subsystem statistical information, as described for the `DB_ENV->mutex_stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->mutex_stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->mutex_stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

Class

DB_ENV

See Also

Mutex Methods (page 432)

DB_ENV->mutex_unlock()

```
#include <db.h>

int
DB_ENV->mutex_unlock(DB_ENV *dbenv, db_mutex_t mutex);
```

The `DB_ENV->mutex_unlock()` method unlocks the mutex locked by DB_ENV->mutex_lock() (page 440).

The `DB_ENV->mutex_unlock()` method returns a non-zero error value on failure and 0 on success.

Parameters

mutex

The **mutex** parameter is a mutex previously locked by DB_ENV->mutex_lock() (page 440).

Errors

The `DB_ENV->mutex_unlock()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Mutex Methods (page 432)

Chapter 10. Replication Methods

This chapter describes the APIs available to build Berkeley DB replicated applications. There are two different ways to build replication into a Berkeley DB application, and the APIs for both are described in this chapter.

For an overview of the two different ways to build a replicated application, see the *Berkeley DB Getting Started with Replicated Applications* guide.

The first, and simplest, way to build a replication Berkeley DB application is via the *Replication Manager*. If the Replication Manager does not meet your application's architectural requirements, you can write your own replication implementation using the "Base APIs".

Note that the Replication Manager is written using the Base APIs.

Note, also, that applications which make use of the Replication Manager use many of the Base APIs as the situation warrants. That said, a few Base API methods cannot be used by applications that are making use of the Replication Manager. Where this is the case, this is noted in the following method descriptions.

Replication and Related Methods

Replication Manager Methods	Description
DB_ENV->repmgr_add_remote_site()	Specify the Replication Manager's remote sites
DB_ENV->repmgr_set_ack_policy(), DB_ENV->repmgr_get_ack_policy()	Specify the Replication Manager's client acknowledgement policy
DB_ENV->repmgr_set_local_site()	Specify the Replication Manager's local site
DB_ENV->repmgr_site_list()	List the sites and their status
DB_ENV->repmgr_start()	Start the Replication Manager
DB_ENV->repmgr_stat()	Replication Manager statistics
DB_ENV->repmgr_stat_print()	Print Replication Manager statistics
Base API Methods	
DB_ENV->rep_elect()	Hold a replication election
DB_ENV->rep_process_message()	Process a replication message
DB_ENV->rep_set_transport()	Configure replication transport callback
DB_ENV->rep_start()	Start replication
Additional Replication Methods	
DB_ENV->rep_stat()	Replication statistics
DB_ENV->rep_stat_print()	Print replication statistics
DB_ENV->rep_sync()	Replication synchronization
DB_ENV->txn_applied()	Check if a transaction has been replicated
Replication Configuration	
DB_ENV->rep_set_clockskew(), DB_ENV->rep_get_clockskew()	Configure master lease clock adjustment
DB_ENV->rep_set_config(), DB_ENV->rep_get_config()	Configure the replication subsystem
DB_ENV->rep_set_limit(), DB_ENV->rep_get_limit()	Limit data sent in response to a single message
DB_ENV->rep_set_nsites(), DB_ENV->rep_get_nsites()	Configure replication group site count
DB_ENV->rep_set_priority(), DB_ENV->rep_get_priority()	Configure replication site priority
DB_ENV->rep_set_request(), DB_ENV->rep_get_request()	Configure replication client retransmission requests
DB_ENV->rep_set_timeout(), DB_ENV->rep_get_timeout()	Configure replication timeouts
Transaction Operations	
DB_TXN->set_commit_token()	Set a commit token

DB_ENV->rep_elect()

```
#include <db.h>

int
DB_ENV->rep_elect(DB_ENV *env,
    u_int32_t nsites, u_int32_t nvotes, u_int32_t flags);
```

The `DB_ENV->rep_elect()` method holds an election for the master of a replication group.

The `DB_ENV->rep_elect()` method is not called by most replication applications. It should only be called by Base API applications implementing their own network transport layer, explicitly holding replication group elections and handling replication messages outside of the Replication Manager framework.

If the election is successful, Berkeley DB will notify the application of the results of the election by means of either the DB_EVENT_REP_ELECTED or DB_EVENT_REP_NEWMASTER events (see DB_ENV->set_event_notify() (page 249) method for more information). The application is responsible for adjusting its relationship to the other database environments in the replication group, including directing all database updates to the newly selected master, in accordance with the results of the election.

The thread of control that calls the `DB_ENV->rep_elect()` method must not be the thread of control that processes incoming messages; processing the incoming messages is necessary to successfully complete an election.

Before calling this method do the following:

- open the database environment by calling the DB_ENV->open() (page 231) method.
- configure the database environment to send replication messages by calling the DB_ENV->rep_set_transport() (page 481) method.
- configure the database environment as a client or a master by calling the DB_ENV->rep_start() (page 484) method.

How Elections are Held

Elections are done in two parts: first, replication sites collect information from the other replication sites they know about, and second, replication sites cast their votes for a new master. The second phase is triggered by one of two things: either the replication site gets election information from **nsites** sites, or the election timeout expires. Once the second phase is triggered, the replication site will cast a vote for the new master of its choice if, and only if, the site has election information from at least **nvotes** sites. If a site receives **nvotes** votes for it to become the new master, then it will become the new master.

We recommend **nvotes** be set to at least:

```
        (sites participating in the election / 2) + 1
```

to ensure there are never more than two masters active at the same time even in the case of a network partition. When a network partitions, the side of the partition with more than half the environments will elect a new master and continue, while the environments communicating with fewer than half of the environments will fail to find a new master, as no site can get **nvotes** votes.

We recommend **nsites** be set to:

```
          number of sites in the replication group - 1
```

when choosing a new master after a current master fails. This allows the group to reach a consensus without having to wait for the timeout to expire.

When choosing a master from among a group of client sites all restarting at the same time, it makes more sense to set **nsites** to the total number of sites in the group, since there is no known missing site. Furthermore, in order to ensure the best choice from among sites that may take longer to boot than the local site, setting **nvotes** also to this same total number of sites will guarantee that every site in the group is considered. Alternatively, using the special timeout for full elections allows full participation on restart but allows election of a master if one site does not reboot and rejoin the group in a reasonable amount of time. (See the Elections section in the *Berkeley DB Programmer's Reference Guide* for more information.)

Setting **nsites** to lower values can increase the speed of an election, but can also result in election failure, and is usually not recommended.

Parameters

nsites

The **nsites** parameter specifies the number of replication sites expected to participate in the election. Once the current site has election information from that many sites, it will short-circuit the election and immediately cast its vote for a new master. The **nsites** parameter must be no less than **nvotes**, or 0 if the election should use the value previously set using the DB_ENV->rep_set_nsites() (page 473) method. If an application is using master leases, then the value **must** be 0 and the value from DB_ENV->rep_set_nsites() (page 473) method must be used.

nvotes

The **nvotes** parameter specifies the minimum number of replication sites from which the current site must have election information, before the current site will cast a vote for a new master. The **nvotes** parameter must be no greater than **nsites**, or 0 if the election should use the value ((**nsites** / 2) + 1) as the **nvotes** argument.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_ENV->rep_elect()` method may fail and return one of the following non-zero errors:

DB_REP_UNAVAIL

The replication group was unable to elect a master, or was unable to complete the election in the election timeout period (see DB_ENV->rep_set_timeout() (page 478) method for more information).

EINVAL

If the database environment was not already configured to communicate with a replication group by a call to DB_ENV->rep_set_transport() (page 481); if the database environment was not already opened; if this method is called from a Replication Manager application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_get_clockskew()

```
#include <db.h>

int
DB_ENV->rep_get_clockskew(DB_ENV *env,
    u_int32_t *fast_clockp, u_int32_t *slow_clockp);
```

The `DB_ENV->rep_get_clockskew()` method returns the current clock skew ratio values, as set by the DB_ENV->rep_set_clockskew() (page 466) method.

The `DB_ENV->rep_get_clockskew()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_clockskew()` method returns a non-zero error value on failure and 0 on success.

Parameters

fast_clockp

The **fast_clockp** parameter references memory into which the value for the fastest clock in the group of sites is copied.

slow_clockp

The **slow_clockp** parameter references memory into which the value for the slowest clock in the group of sites is copied.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_clockskew() (page 466)

DB_ENV->rep_get_config()

```
#include <db.h>

int
DB_ENV->rep_get_config(DB_ENV *env, u_int32_t which, int *onoffp);
```

The `DB_ENV->rep_get_config()` method returns whether the specified **which** parameter is currently set or not. See the DB_ENV->rep_set_config() (page 468) method for the configuration flags that can be set for replication.

The `DB_ENV->rep_get_config()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_config()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter is the configuration flag which is being checked. See the DB_ENV->rep_set_config() (page 468) method for a list of configuration flags that you can provide to this parameter.

onoffp

The **onoffp** parameter references memory into which the configuration of the specified **which** parameter is copied.

If the returned **onoff** value is zero, the parameter is off; otherwise it is on.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_config() (page 468)

DB_ENV->rep_get_limit()

```
#include <db.h>

int
DB_ENV->rep_get_limit(DB_ENV *env, u_int32_t *gbytesp,
    u_int32_t *bytesp);
```

The `DB_ENV->rep_get_limit()` method returns the byte-count limit on the amount of data that will be transmitted from a site in response to a single message processed by the DB_ENV->rep_process_message() (page 463) method. This value is configurable using the DB_ENV->rep_set_limit() (page 471) method.

The `DB_ENV->rep_get_limit()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_limit()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytesp

The **gbytesp** parameter references memory into which the gigabytes component of the current transmission limit is copied.

bytesp

The **bytesp** parameter references memory into which the bytes component of the current transmission limit is copied.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_limit() (page 471)

DB_ENV->rep_get_nsites()

```
#include <db.h>

int
DB_ENV->rep_get_nsites(DB_ENV *env, u_int32_t *nsitesp);
```

The `DB_ENV->rep_get_nsites()` method returns the total number of sites in the replication group. This value is configurable using the DB_ENV->rep_set_nsites() (page 473) method.

The `DB_ENV->rep_get_nsites()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_nsites()` method returns a non-zero error value on failure and 0 on success.

Parameters

nsitesp

The `DB_ENV->rep_get_nsites()` method returns the total number of sites in the replication group in **nsitesp**.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_nsites() (page 473)

DB_ENV->rep_get_priority()

```
#include <db.h>

int
DB_ENV->rep_get_priority(DB_ENV *env, u_int32_t *priorityp);
```

The `DB_ENV->rep_get_priority()` method returns the database environment priority as configured using the DB_ENV->rep_set_priority() (page 475) method.

The `DB_ENV->rep_get_priority()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priorityp

The `DB_ENV->rep_get_priority()` method returns the database environment priority in **priorityp**.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_priority() (page 475)

DB_ENV->rep_get_request()

```
#include <db.h>

int
DB_ENV->rep_get_request(DB_ENV *env, u_int32_t *minp, u_int32_t *maxp);
```

The `DB_ENV->rep_get_request()` method returns the minimum and maximum number of microseconds a client waits before requesting retransmission. These values can be configured using the DB_ENV->rep_set_request() (page 476) method.

The `DB_ENV->rep_get_request()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_request()` method returns a non-zero error value on failure and 0 on success.

Parameters

minp

The **minp** parameter references memory into which the minimum number of microseconds a client will wait before requesting retransmission is copied.

maxp

The **maxp** parameter references memory into which the maximum number of microseconds a client will wait before requesting retransmission is copied.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_request() (page 476)

DB_ENV->rep_get_timeout()

```
#include <db.h>

int
DB_ENV->rep_get_timeout(DB_ENV *env, int which, u_int32_t *timeoutp);
```

The `DB_ENV->rep_get_timeout()` method returns the timeout value for the specified **which** parameter. Timeout values can be managed using the DB_ENV->rep_set_timeout() (page 478) method.

The `DB_ENV->rep_get_timeout()` method may be called at any time during the life of the application.

The `DB_ENV->rep_get_timeout()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter is the timeout for which the value is being returned. See the DB_ENV->rep_set_timeout() (page 478) method for a list of timeouts that you can provide to this parameter.

timeoutp

The **timeoutp** parameter references memory into which the timeout value of the specified **which** parameter is copied.

The returned timeout value is in microseconds.

Errors

The `DB_ENV->rep_get_timeout()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->rep_set_timeout() (page 478)

DB_ENV->rep_process_message()

```
#include <db.h>

int
DB_ENV->rep_process_message(DB_ENV *env,
    DBT *control, DBT *rec, int envid, DB_LSN *ret_lsnp)
```

The `DB_ENV->rep_process_message()` method processes an incoming replication message sent by a member of the replication group to the local database environment.

The `DB_ENV->rep_process_message()` method is not called by most replication applications. It should only be called by Base API applications implementing their own network transport layer, explicitly holding replication group elections and handling replication messages outside of the Replication Manager framework.

For implementation reasons, all incoming replication messages must be processed using the same DB_ENV handle. It is not required that a single thread of control process all messages, only that all threads of control processing messages use the same handle.

Before calling this method, the enclosing database environment must already have been opened by calling the DB_ENV->open() (page 231) method and must already have been configured to send replication messages by calling the DB_ENV->rep_set_transport() (page 481) method.

The `DB_ENV->rep_process_message()` method has additional return values:

- **DB_REP_DUPMASTER**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_DUPMASTER` if the replication group has more than one master. The application should reconfigure itself as a client by calling the DB_ENV->rep_start() (page 484) method, and then call for an election by calling DB_ENV->rep_elect() (page 453).

- **DB_REP_HOLDELECTION**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_HOLDELECTION` if an election is needed. The application should call for an election by calling DB_ENV->rep_elect() (page 453).

- **DB_REP_IGNORE**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_IGNORE` if this message cannot be processed. This is an indication that this message is irrelevant to the current replication state (for example, an old message from a previous generation arrives and is processed late).

- **DB_REP_ISPERM**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_ISPERM` if processing this message results in the processing of records that are permanent. The maximum LSN of the permanent records stored is returned.

- **DB_REP_JOIN_FAILURE**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_JOIN_FAILURE` if a new master has been chosen but the client is unable to synchronize with the new master (possibly because the client has been configured with the DB_REP_CONF_AUTOINIT flag set to turn off automatic internal initialization).

- **DB_REP_NEWSITE**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_NEWSITE` if the system received contact information from a new environment. The **rec** parameter contains the opaque data specified to the DB_ENV->rep_start() (page 484) **cdata** parameter. The application should take whatever action is needed to establish a communication channel with this new environment.

- **DB_REP_NOTPERM**

  The `DB_ENV->rep_process_message()` method will return `DB_REP_NOTPERM` if a message carrying a DB_REP_PERMANENT flag was processed successfully, but was not written to disk. The LSN of this record is returned. The application should take whatever action is deemed necessary to retain its recoverability characteristics.

Unless otherwise specified, the `DB_ENV->rep_process_message()` method returns a non-zero error value on failure and 0 on success.

Parameters

control

The **control** parameter should reference a copy of the **control** parameter specified by Berkeley DB on the sending environment. See the DB_ENV->rep_set_transport() (page 481) method for more information.

rec

The **rec** parameter should reference a copy of the **rec** parameter specified by Berkeley DB on the sending environment. See the DB_ENV->rep_set_transport() (page 481) method for more information.

envid

The **envid** parameter should contain the local identifier that corresponds to the environment that sent the message to be processed (see Replication environment IDs for more information).

ret_lsnp

If `DB_ENV->rep_process_message()` method returns `DB_REP_NOTPERM` then the **ret_lsnp** parameter will contain the log sequence number of this permanent log message that could not be written to disk. If `DB_ENV->rep_process_message()` method returns `DB_REP_ISPERM` then the **ret_lsnp** parameter will contain largest log sequence number of the permanent records that are now written to disk as a result of processing this message. In all other cases the value of **ret_lsnp** is undefined.

Errors

The `DB_ENV->rep_process_message()` method may fail and return one of the following non-zero errors:

EINVAL

If the database environment was not already configured to communicate with a replication group by a call to DB_ENV->rep_set_transport() (page 481); if the database environment was not already opened; if this method is called from a Replication Manager application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_clockskew()

```
#include <db.h>

int
DB_ENV->rep_set_clockskew(DB_ENV *env,
    u_int32_t fast_clock, u_int32_t slow_clock);
```

The DB_ENV->rep_set_clockskew() method sets the clock skew ratio among replication group members based on the fastest and slowest measurements among the group for use with master leases. Calling this method is optional; the default values for clock skew assume no skew. The user must also configure leases via the DB_ENV->rep_set_config() (page 468) method. Additionally, the user must also set the master lease timeout via the DB_ENV->rep_set_timeout() (page 478) method and the number of sites in the replication group via the DB_ENV->rep_set_nsites() (page 473) method. These methods may be called in any order. For a description of the clock skew values, see Clock skew in the *Berkeley DB Programmer's Reference Guide*. For a description of master leases, see Master leases in the *Berkeley DB Programmer's Reference Guide*.

These arguments can be used to express either raw measurements of a clock timing experiment or a percentage across machines. For example, if a group of sites has a 2% variance, then **fast_clock** should be set to 102, and **slow_clock** should be set to 100. Or, for a 0.03% difference, you can use 10003 and 10000 respectively.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_clockskew", one or more whitespace characters, and the clockskew specified in two parts: the fast_clock and the slow_clock. For example, "rep_set_clockskew 102 100". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DB_ENV->rep_set_clockskew() method configures a database environment, not only operations performed using the specified DB_ENV handle.

The DB_ENV->rep_set_clockskew() method may not be called after the DB_ENV->repmgr_start() (page 502) or DB_ENV->rep_start() (page 484) methods are called.

The DB_ENV->rep_set_clockskew() method returns a non-zero error value on failure and 0 on success.

Parameters

fast_clock

The value, relative to the **slow_clock**, of the fastest clock in the group of sites.

slow_clock

The value of the slowest clock in the group of sites.

Errors

The `DB_ENV->rep_set_clockskew()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after replication is started with a call to the DB_ENV->repmgr_start() (page 502) or the DB_ENV->rep_start() (page 484) method; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_config()

```
#include <db.h>

int
DB_ENV->rep_set_config(DB_ENV *env, u_int32_t which, int onoff);
```

The `DB_ENV->rep_set_config()` method configures the Berkeley DB replication subsystem.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_config", one or more whitespace characters, and the method **which** parameter as a string and optionally one or more whitespace characters, and the string "on" or "off". If the optional string is omitted, the default is "on"; for example, "rep_set_config DB_REP_CONF_NOWAIT" or "rep_set_config DB_REP_CONF_NOWAIT on". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->rep_set_config()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->rep_set_config()` method may not be called to set in-memory replication after the environment is opened using the DB_ENV->open() (page 231) method. This method may also not be called to set master leases after the DB_ENV->rep_start() (page 484) or DB_ENV->repmgr_start() (page 502) methods are called. For all other **which** parameters, this method may be called at any time during the life of the application.

The `DB_ENV->rep_set_config()` method returns a non-zero error value on failure and 0 on success.

Parameters

which

The **which** parameter must be set to one of the following values:

- `DB_REP_CONF_AUTOINIT`

  The replication master will automatically re-initialize outdated clients. This option is turned on by default.

- `DB_REP_CONF_BULK`

  The replication master sends groups of records to the clients in a single network transfer.

- `DB_REP_CONF_DELAYCLIENT`

  The client should delay synchronizing to a newly declared master. Clients configured in this way will remain unsynchronized until the application calls the DB_ENV->rep_sync() (page 493) method.

- DB_REP_CONF_INMEM

  Store internal replication information in memory only.

  By default, replication creates files in the environment home directory to preserve some internal information. If this configuration flag is turned on, replication only stores this internal information in-memory and cannot keep persistent state across a site crash or reboot. This results in the following limitations:

  - A master site should not reappoint itself master immediately after crashing or rebooting because the application would incur a slightly higher risk of client crashes. The former master site should rejoin the replication group as a client. The application should either hold an election or appoint a different site to be the next master.

  - An application has a slightly higher risk that elections will fail or be unable to complete. Calling additional elections should eventually yield a winner.

  - An application has a slight risk that the wrong site may win an election, resulting in the loss of some data. This is consistent with the general loss of data durability when running in-memory.

  This configuration flag can only be turned on before the environment is opened with the DB_ENV->open() (page 231) method. Its value cannot be changed while the environment is open.

- DB_REP_CONF_LEASE

  Master leases will be used for this site.

  Configuring this option may result in DB_REP_LEASE_EXPIRED error returns from the DB->get() (page 30) and DBcursor->get() (page 158) methods when attempting to read entries from a database after the site's master lease has expired.

  This configuration flag may not be set after the DB_ENV->repmgr_start() (page 502) method or the DB_ENV->rep_start() (page 484) method is called. All sites in the replication group should have the same value for this configuration flag.

- DB_REP_CONF_NOWAIT

  Berkeley DB method calls that would normally block while clients are in recovery will return errors immediately.

- DB_REPMGR_CONF_ELECTIONS

  Replication Manager automatically runs elections to choose a new master when the old master appears to have become disconnected. This option is turned on by default.

  If this option is turned off, the application is responsible for assigning the new master explicitly, by calling the `DB_ENV->repmgr_start()` method.

Caution

Most Replication Manager applications should accept the default automatic behavior. Allowing two sites in a replication group to act as master simultaneously can lead to loss of data.

In an application with multiple processes per database environment, only the main replication process may change this configuration setting.

- DB_REPMGR_CONF_2SITE_STRICT

  Replication Manager observes the strict "majority" rule in managing elections, even in a group with only 2 sites. This means the client in a 2-site group will be unable to take over as master if the original master fails or becomes disconnected. (See the Elections section in the *Berkeley DB Programmer's Reference Guide* for more information.) Both sites in the replication group should have the same value for this configuration flag.

onoff

If the **onoff** parameter is zero, the configuration flag is turned off. Otherwise, it is turned on. Most configuration flags are turned off by default, exceptions are noted above.

Errors

The `DB_ENV->rep_set_config()` method may fail and return one of the following non-zero errors:

EINVAL

If setting in-memory replication after the database environment is already opened; if setting master leases after replication is started; if setting the 2-site strict majority rule for a Base API application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_limit()

```
#include <db.h>

int
DB_ENV->rep_set_limit(DB_ENV *env, u_int32_t gbytes, u_int32_t bytes);
```

The `DB_ENV->rep_set_limit()` method sets record transmission throttling. This is a byte-count limit on the amount of data that will be transmitted from a site in response to a single message processed by the DB_ENV->rep_process_message() (page 463) method. The limit is not a hard limit, and the record that exceeds the limit is the last record to be sent.

Record transmission throttling is turned on by default with a limit of 10MB.

If the values passed to the `DB_ENV->rep_set_limit()` method are both zero, then the transmission limit is turned off.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_limit", one or more whitespace characters, and the limit specified in two parts: the gigabytes and the bytes values. For example, "rep_set_limit 0 1048576" sets a 1 megabyte limit. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->rep_set_limit()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->rep_set_limit()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_limit()` method returns a non-zero error value on failure and 0 on success.

Parameters

gbytes

The **gbytes** parameter specifies the number of gigabytes which, when added to the **bytes** parameter, specifies the maximum number of bytes that will be sent in a single call to the DB_ENV->rep_process_message() (page 463) method.

bytes

The **bytes** parameter specifies the number of bytes which, when added to the **gbytes** parameter, specifies the maximum number of bytes that will be sent in a single call to the DB_ENV->rep_process_message() (page 463) method.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_nsites()

```
#include <db.h>

int
DB_ENV->rep_set_nsites(DB_ENV *env, u_int32_t nsites);
```

The `DB_ENV->rep_set_nsites()` method specifies the total number of sites in a replication group.

The `DB_ENV->rep_set_nsites()` method is typically called by Replication Manager applications. (However, see also the DB_ENV->rep_elect() (page 453) method **nsites** parameter.)

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_nsites", one or more whitespace characters, and the number of sites specified. For example, "rep_set_nsites 5" sets the number of sites to 5. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->rep_set_nsites()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

If master leases are in use, the `DB_ENV->rep_set_nsites()` method may not be called after the DB_ENV->rep_start() (page 484) or DB_ENV->repmgr_start() (page 502) methods are called. If master leases are not in use, the `DB_ENV->rep_set_nsites()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_nsites()` method returns a non-zero error value on failure and 0 on success.

Parameters

nsites

An integer specifying the total number of sites in the replication group.

Errors

The `DB_ENV->rep_set_nsites()` method may fail and return one of the following non-zero errors:

EINVAL

If master leases are in use and replication has already been started; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_priority()

```
#include <db.h>

int
DB_ENV->rep_set_priority(DB_ENV *env, u_int32_t priority);
```

The `DB_ENV->rep_set_priority()` method specifies the database environment's priority in replication group elections. A special value of 0 indicates that this environment cannot be a replication group master.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_priority", one or more whitespace characters, and the priority of this site. For example, "rep_set_priority 1" sets the priority of this site to 1. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

Note that if the application never explicitly sets a priority, then a default value of 100 is used.

The `DB_ENV->rep_set_priority()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->rep_set_priority()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priority

The priority of this database environment in the replication group. The priority must be a non-zero integer, or 0 if this environment cannot be a replication group master. (See Replication environment priorities for more information).

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_request()

```
#include <db.h>

int
DB_ENV->rep_set_request(DB_ENV *env, u_int32_t min, u_int32_t max);
```

The `DB_ENV->rep_set_request()` method sets a threshold for the minimum and maximum time that a client waits before requesting retransmission of a missing message. Specifically, if the client detects a gap in the sequence of incoming log records or database pages, Berkeley DB will wait for at least **min** microseconds before requesting retransmission of the missing record. Berkeley DB will double that amount before requesting the same missing record again, and so on, up to a maximum threshold of **max** microseconds.

These values are thresholds only. Replication Manager applications use these values to determine when to automatically request retransmission of missing messages. For Base API applications, Berkeley DB has no thread available in the library as a timer, so the threshold is only checked when a thread enters the Berkeley DB library to process an incoming replication message. Any amount of time may have passed since the last message arrived and Berkeley DB only checks whether the amount of time since a request was made is beyond the threshold value or not.

By default the minimum is 40000 and the maximum is 1280000 (1.28 seconds). These defaults are fairly arbitrary and the application likely needs to adjust these. The values should be based on expected load and performance characteristics of the master and client host platforms and transport infrastructure as well as round-trip message time.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_request", one or more whitespace characters, and the request times specified in two parts: the min and the max. For example, "rep_set_request 40000 1280000". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->rep_set_request()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->rep_set_request()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_request()` method returns a non-zero error value on failure and 0 on success.

Parameters

min

The minimum number of microseconds a client waits before requesting retransmission.

max

The maximum number of microseconds a client waits before requesting retransmission.

Errors

The `DB_ENV->rep_set_request()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_timeout()

```
#include <db.h>

int
DB_ENV->rep_set_timeout(DB_ENV *env, int which, u_int32_t timeout);
```

The `DB_ENV->rep_set_timeout()` method specifies a variety of replication timeout values.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "rep_set_timeout", one or more whitespace characters, and the **which** parameter specified as a string and the timeout specified as two parts. For example, "rep_set_timeout DB_REP_CONNECTION_RETRY 15000000" specifies the connection retry timeout for 15 seconds. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->rep_set_timeout()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->rep_set_timeout()` method may not be called to set the master lease timeout after the DB_ENV->repmgr_start() (page 502) method or the DB_ENV->rep_start() (page 484) method is called. For all other timeouts, the `DB_ENV->rep_set_timeout()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_timeout()` method returns a non-zero error value on failure and 0 on success.

Parameters

timeout

The **timeout** parameter is the timeout value. It must be specified as an unsigned 32-bit number of microseconds, limiting the maximum timeout to roughly 71 minutes.

which

The **which** parameter must be set to one of the following values:

- `DB_REP_ACK_TIMEOUT`

  Configure the amount of time the Replication Manager's transport function waits to collect enough acknowledgments from replication group clients, before giving up and returning a failure indication. The default wait time is 1 second.

- `DB_REP_CHECKPOINT_DELAY`

  Configure the amount of time a master site will delay between completing a checkpoint and writing a checkpoint record into the log. This delay allows clients to complete their own checkpoints before the master requires completion of them. The default is 30 seconds. If all databases in the environment, and the environment's transaction log, are configured to reside in memory (never preserved to disk), then, although checkpoints are still necessary, the delay is not useful and should be set to 0.

- DB_REP_CONNECTION_RETRY

  Configure the amount of time the Replication Manager will wait before trying to re-establish a connection to another site after a communication failure. The default wait time is 30 seconds.

- DB_REP_ELECTION_TIMEOUT

  The timeout period for an election. The default timeout is 2 seconds.

- DB_REP_ELECTION_RETRY

  Configure the amount of time the Replication Manager will wait before retrying a failed election. The default wait time is 10 seconds.

- DB_REP_FULL_ELECTION_TIMEOUT

  An optional configuration timeout period to wait for full election participation the first time the replication group finds a master. By default this option is turned off and normal election timeouts are used. (See the Elections section in the *Berkeley DB Programmer's Reference Guide* for more information.)

- DB_REP_HEARTBEAT_MONITOR

  The amount of time the Replication Manager, running at a client site, waits for some message activity on the connection from the master (heartbeats or other messages) before concluding that the connection has been lost. This timeout should be of longer duration than the DB_REP_HEARTBEAT_SEND timeout to ensure that heartbeats are not missed. When 0 (the default), no monitoring is performed.

- DB_REP_HEARTBEAT_SEND

  The frequency at which the Replication Manager, running at a master site, broadcasts a heartbeat message in an otherwise idle system. When 0 (the default), no heartbeat messages will be sent.

- DB_REP_LEASE_TIMEOUT

  Configure the amount of time a client grants its master lease to a master. When using master leases all sites in a replication group must use the same lease timeout value. There is no default value. If leases are desired, this method must be called prior to calling DB_ENV->rep_start() (page 484) method. See also DB_ENV->rep_set_clockskew() (page 466) method, DB_ENV->rep_set_config() (page 468) method or Master leases.

Errors

The DB_ENV->rep_set_timeout() method may fail and return one of the following non-zero errors:

EINVAL

If setting the lease timeout and replication has already been started; if setting a Replication Manager timeout for a Base API application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_set_transport()

```
#include <db.h>

int
DB_ENV->rep_set_transport(DB_ENV *env, int envid,
    int (*send)(DB_ENV *dbenv,
    const DBT *control, const DBT *rec, const DB_LSN *lsnp,
    int envid, u_int32_t flags));
```

The `DB_ENV->rep_set_transport()` method initializes the communication infrastructure for a database environment participating in a replicated application.

The `DB_ENV->rep_set_transport()` method is not called by most replication applications. It should only be called by Base API applications implementing their own network transport layer, explicitly holding replication group elections and handling replication messages outside of the Replication Manager framework.

The `DB_ENV->rep_set_transport()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->rep_set_transport()` method may be called at any time during the life of the application.

The `DB_ENV->rep_set_transport()` method returns a non-zero error value on failure and 0 on success.

Note

Berkeley DB is not re-entrant. The callback function for this method should not attempt to make library calls (for example, to release locks or close open handles). Re-entering Berkeley DB is not guaranteed to work correctly, and the results are undefined.

Parameters

envid

The **envid** parameter is the local environment's ID. It must be a non-negative integer and uniquely identify this Berkeley DB database environment (see Replication environment IDs for more information).

send

The **send** callback function is used to transmit data using the replication application's communication infrastructure. The parameters to **send** are as follows:

- `dbenv`

  The **dbenv** parameter is the enclosing database environment handle.

- `control`

  The **control** parameter is the first of the two data elements to be transmitted by the **send** function.

- `rec`

  The **rec** parameter is the second of the two data elements to be transmitted by the **send** function.

- `lsnp`

  If the type of message to be sent has an LSN associated with it, then the **lsnp** parameter contains the LSN of the record being sent. This LSN can be used to determine that certain records have been processed successfully by clients.

- `envid`

  The **envid** parameter is a positive integer identifier that specifies the replication environment to which the message should be sent (see Replication environment IDs for more information).

  The special identifier `DB_EID_BROADCAST` indicates that a message should be broadcast to every environment in the replication group. The application may use a true broadcast protocol or may send the message in sequence to each machine with which it is in communication. In both cases, the sending site should not be asked to process the message.

  The special identifier DB_EID_INVALID indicates an invalid environment ID. This may be used to initialize values that are subsequently checked for validity.

- `flags`

  The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

  - `DB_REP_ANYWHERE`

    The message is a client request that can be satisfied by another client as well as by the master.

  - `DB_REP_NOBUFFER`

    The record being sent should be transmitted immediately and not buffered or delayed.

  - `DB_REP_PERMANENT`

    The record being sent is critical for maintaining database integrity (for example, the message includes a transaction commit). The application should take appropriate action to enforce the reliability guarantees it has chosen, such as waiting for acknowledgement from one or more clients.

  - `DB_REP_REREQUEST`

The message is a client request that has already been made and to which no response was received.

It may sometimes be useful to pass application-specific data to the send function; see Environment FAQ for a discussion on how to do this.

The **send** function must return 0 on success and non-zero on failure. If the send function fails, the message being sent is necessary to maintain database integrity, and the local log is not configured for synchronous flushing, the local log will be flushed; otherwise, any error from the **send** function will be ignored.

Errors

The `DB_ENV->rep_set_transport()` method may fail and return one of the following non-zero errors:

EINVAL

The method is called from a Replication Manager application; or an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_start()

```
#include <db.h>

int
DB_ENV->rep_start(DB_ENV *env, DBT *cdata, u_int32_t flags);
```

The DB_ENV->rep_start() method configures the database environment as a client or master in a group of replicated database environments.

The DB_ENV->rep_start() method is not called by most replication applications. It should only be called by Base API applications implementing their own network transport layer, explicitly holding replication group elections and handling replication messages outside of the Replication Manager framework.

Replication master environments are the only database environments where replicated databases may be modified. Replication client environments are read-only as long as they are clients. Replication client environments may be upgraded to be replication master environments in the case that the current master fails or there is no master present. If master leases are in use, this method cannot be used to appoint a master, and should only be used to configure a database environment as a master as the result of an election.

The enclosing database environment must already have been opened by calling the DB_ENV->open() (page 231) method and must already have been configured to send replication messages by calling the DB_ENV->rep_set_transport() (page 481) method.

The DB_ENV->rep_start() method returns a non-zero error value on failure and 0 on success.

Parameters

cdata

The **cdata** parameter is an opaque data item that is sent over the communication infrastructure when the client comes online (see Connecting to a new site for more information). If no such information is useful, **cdata** should be NULL.

flags

The **flags** parameter must be set to one of the following values:

- DB_REP_CLIENT

  Configure the environment as a replication client.

- DB_REP_MASTER

  Configure the environment as a replication master.

Errors

The DB_ENV->rep_start() method may fail and return one of the following non-zero errors:

DB_REP_UNAVAIL

If the flags parameter was passed as DB_REP_MASTER but the database environment cannot currently become the replication master because it is temporarily initializing and is incomplete.

EINVAL

If the database environment was not already configured to communicate with a replication group by a call to DB_ENV->rep_set_transport() (page 481); the database environment was not already opened; this method is called from a Replication Manager application; outstanding master leases are granted; this method is used to appoint a new master when master leases are in use; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_stat()

```
#include <db.h>

int
DB_ENV->rep_stat(DB_ENV *env, DB_REP_STAT **statp, u_int32_t flags);
```

The `DB_ENV->rep_stat()` method returns the replication subsystem statistics.

The `DB_ENV->rep_stat()` method creates a statistical structure of type `DB_REP_STAT` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following `DB_REP_STAT` fields will be filled in:

- **uintmax_t st_bulk_fills;**

  The number of times the bulk buffer filled up, forcing the buffer content to be sent.

- **uintmax_t st_bulk_overflows;**

  The number of times a record was bigger than the entire bulk buffer, and therefore had to be sent as a singleton.

- **uintmax_t st_bulk_records;**

  The number of records added to a bulk buffer.

- **uintmax_t st_bulk_transfers;**

  The number of bulk buffers transferred (via a call to the application's **send** function).

- **uintmax_t st_client_rerequests;**

  The number of times this client site received a "re-request" message, indicating that a request it previously sent to another client could not be serviced by that client. (Compare to **st_client_svc_miss**.)

- **uintmax_t st_client_svc_miss;**

  The number of "request" type messages received by this client that could not be processed, forcing the originating requester to try sending the request to the master (or another client).

- **uintmax_t st_client_svc_req;**

  The number of "request" type messages received by this client. ("Request" messages are usually sent from a client to the master, but a message marked with the DB_REP_ANYWHERE flag in the invocation of the application's **send** function may be sent to another client instead.)

- **u_int32_t st_dupmasters;**

  The number of duplicate master conditions originally detected at this site.

- **u_int32_t st_egen;**

  The election generation number for the current or next election.

- **int st_election_cur_winner;**

  The environment ID of the winner of the current or last election.

- **u_int32_t st_election_gen;**

  The master generation number of the winner of the current or last election.

- **DB_LSN st_election_lsn;**

  The maximum LSN of the winner of the current or last election.

- **u_int32_t st_election_nsites;**

  The number of sites responding to this site during the current election.

- **u_int32_t st_election_nvotes;**

  The number of votes required in the current or last election.

- **u_int32_t st_election_priority;**

  The priority of the winner of the current or last election.

- **u_int32_t st_election_sec;**

  The number of seconds the last election took (the total election time is **st_election_sec** plus **st_election_usec**).

- **int st_election_status;**

  The current election phase (0 if no election is in progress).

- **u_int32_t st_election_tiebreaker;**

  The tiebreaker value of the winner of the current or last election.

- **u_int32_t st_election_usec;**

  The number of microseconds the last election took (the total election time is **st_election_sec** plus **st_election_usec**).

- **u_int32_t st_election_votes;**

  The number of votes received during the current election.

- **uintmax_t st_elections;**

  The number of elections held.

- **uintmax_t st_elections_won;**

  The number of elections won.

- **int st_env_id;**

  The current environment ID.

- **u_int32_t st_env_priority;**

  The current environment priority.

- **u_int32_t st_gen;**

  The current master generation number.

- **uintmax_t st_log_duplicated;**

  The number of duplicate log records received.

- **uintmax_t st_log_queued;**

  The number of log records currently queued.

- **uintmax_t st_log_queued_max;**

  The maximum number of log records ever queued at once.

- **uintmax_t st_log_queued_total;**

  The total number of log records queued.

- **uintmax_t st_log_records;**

  The number of log records received and appended to the log.

- **uintmax_t st_log_requested;**

  The number of times log records were missed and requested.

- **int st_master;**

  The current master environment ID.

- **uintmax_t st_master_changes;**

The number of times the master has changed.

- **u_int32_t st_max_lease_sec;**

  The number of seconds of the longest lease (the total lease time is **st_max_lease_sec** plus **st_max_lease_usec**).

- **u_int32_t st_max_lease_usec;**

  The number of microseconds of the longest lease (the total lease time is **st_max_lease_sec** plus **st_max_lease_usec**).

- **DB_LSN st_max_perm_lsn;**

  The LSN of the maximum permanent log record, or 0 if there are no permanent log records.

- **uintmax_t st_msgs_badgen;**

  The number of messages received with a bad generation number.

- **uintmax_t st_msgs_processed;**

  The number of messages received and processed.

- **uintmax_t st_msgs_recover;**

  The number of messages ignored due to pending recovery.

- **uintmax_t st_msgs_send_failures;**

  The number of failed message sends.

- **uintmax_t st_msgs_sent;**

  The number of messages sent.

- **uintmax_t st_newsites;**

  The number of new site messages received.

- **DB_LSN st_next_lsn;**

  In replication environments configured as masters, the next LSN to be used. In replication environments configured as clients, the next LSN expected.

- **u_int32_t st_next_pg;**

  The next page number we expect to receive.

- **u_int32_t st_nsites;**

  The number of sites used in the last election.

- **uintmax_t st_nthrottles;**

  Transmission limited. This indicates the number of times that data transmission was stopped to limit the amount of data sent in response to a single call to DB_ENV->rep_process_message() (page 463).

- **uintmax_t st_outdated;**

  The number of outdated conditions detected.

- **uintmax_t st_pg_duplicated;**

  The number of duplicate pages received.

- **uintmax_t st_pg_records;**

  The number of pages received and stored.

- **uintmax_t st_pg_requested;**

  The number of pages missed and requested from the master.

- **uintmax_t st_startsync_delayed;**

  The number of times the client had to delay the start of a cache flush operation (initiated by the master for an impending checkpoint) because it was missing some previous log record(s).

- **u_int32_t st_startup_complete;**

  The client site has completed its startup procedures and is now handling live records from the master.

- **u_int32_t st_status;**

  The current replication mode. Set to DB_REP_MASTER if the environment is a replication master, DB_REP_CLIENT if the environment is a replication client, or 0 if replication is not configured.

- **uintmax_t st_txns_applied;**

  The number of transactions applied.

- **DB_LSN st_waiting_lsn;**

  The LSN of the first log record we have after missing log records being waited for, or 0 if no log records are currently missing.

- **u_int32_t st_waiting_pg;**

  The page number of the first page we have after missing pages being waited for, or 0 if no pages are currently missing.

The `DB_ENV->rep_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->rep_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->rep_stat()` method may fail and return one of the following non-zero errors:

EINVAL

If the database environment was not already opened; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_stat_print()

```
#include <db.h>

int
DB_ENV->rep_stat_print(DB_ENV *env, u_int32_t flags);
```

The DB_ENV->rep_stat_print() method displays the replication subsystem statistical information, as described for the DB_ENV->rep_stat() method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The DB_ENV->rep_stat_print() method may not be called before the DB_ENV->open() (page 231) method is called.

The DB_ENV->rep_stat_print() method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- DB_STAT_ALL

  Display all available information.

- DB_STAT_CLEAR

  Reset statistics after displaying their values.

Errors

The DB_ENV->rep_stat_print() method may fail and return one of the following non-zero errors:

EINVAL

If the method was called before DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->rep_sync()

```
#include <db.h>

int
DB_ENV->rep_sync(DB_ENV *env, u_int32_t flags);
```

The DB_ENV->rep_sync() method forces master synchronization to begin for this client. This method is the other half of setting the DB_REP_CONF_DELAYCLIENT flag via the DB_ENV->rep_set_config() (page 468) method.

If an application has configured delayed master synchronization, the application must synchronize explicitly (otherwise the client will remain out-of-date and will ignore all database changes forwarded from the replication group master). The DB_ENV->rep_sync() method may be called any time after the client application learns that the new master has been established (by receiving a DB_EVENT_REP_NEWMASTER event notification).

Before calling this method, the enclosing database environment must already have been opened by calling the DB_ENV->open() (page 231) method and must already have been configured to send replication messages by calling the DB_ENV->rep_set_transport() (page 481) method.

The DB_ENV->rep_sync() method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The DB_ENV->rep_sync() method may fail and return one of the following non-zero errors:

EINVAL

If the database environment was not already configured to communicate with a replication group by a call to DB_ENV->rep_set_transport() (page 481); the database environment was not already opened; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_add_remote_site()

```
#include <db.h>

int
DB_ENV->repmgr_add_remote_site(DB_ENV *env,
    const char *host, u_int port, int *eidp, u_int32_t flags);
```

The `DB_ENV->repmgr_add_remote_site()` method adds a new replication site to the Replication Manager's list of known sites. It is not necessary for all sites in a replication group to know about all other sites in the group.

The `DB_ENV->repmgr_add_remote_site()` method may be called at any time during the life of the application.

The `DB_ENV->repmgr_add_remote_site()` method returns a non-zero error value on failure and 0 on success.

Parameters

host

The site's host identification string, generally a TCP/IP host name.

port

The port number on which the remote site is receiving.

eidp

If **eidp** is non-NULL, and the database environment has already been opened by calling DB_ENV->open() (page 231), then the environment ID assigned to the remote site is returned in the memory location referenced by **eidp**.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_REPMGR_PEER`

  Specifying the `DB_REPMGR_PEER` flag configures client-to-client synchronization with the specified remote site.

  You can configure one or more remote sites as client peers. To remove a remote site as a possible client peer, call the `DB_ENV->repmgr_add_remote_site()` method for that site with a **flags** value excluding the DB_REPMGR_PEER bit value.

Errors

The `DB_ENV->repmgr_add_remote_site()` method may fail and return one of the following non-zero errors:

EINVAL

If this method is called from a base replication API application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_get_ack_policy()

```
#include <db.h>

int
DB_ENV->repmgr_get_ack_policy(DB_ENV *env, int *ack_policyp);
```

The `DB_ENV->repmgr_get_ack_policy()` method returns the Replication Manager's client acknowledgment policy. This is configured using the DB_ENV->repmgr_set_ack_policy() (page 497) method.

The `DB_ENV->repmgr_get_ack_policy()` method may be called at any time during the life of the application.

The `DB_ENV->repmgr_get_ack_policy()` method returns a non-zero error value on failure and 0 on success.

Parameters

ack_policyp

The **ack_policyp** parameter references memory into which the Replication Manager's client acknowledgement policy is copied.

Class

DB_ENV

See Also

Replication and Related Methods (page 452), DB_ENV->repmgr_set_ack_policy() (page 497)

DB_ENV->repmgr_set_ack_policy()

```
#include <db.h>

int
DB_ENV->repmgr_set_ack_policy(DB_ENV *env, int ack_policy);
```

The `DB_ENV->repmgr_set_ack_policy()` method specifies how master and client sites will handle acknowledgment of replication messages which are necessary for "permanent" records. The current implementation requires all sites in a replication group configure the same acknowledgement policy.

The database environment's replication subsystem may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "repmgr_set_ack_policy", one or more whitespace characters, and the **ack_policy** parameter specified as a string. For example, "repmgr_set_ack_policy DB_REPMGR_ACKS_ALL". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

Waiting for client acknowledgements is always limited by the DB_REP_ACK_TIMEOUT specified by the DB_ENV->rep_set_timeout() (page 478) method.

The `DB_ENV->repmgr_set_ack_policy()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->repmgr_set_ack_policy()` method may be called at any time during the life of the application.

The `DB_ENV->repmgr_set_ack_policy()` method returns a non-zero error value on failure and 0 on success.

Parameters

ack_policy

Some acknowledgement policies use the concept of an electable peer, which is a client capable of being subsequently elected master of the replication group. The **ack_policy** parameter must be set to one of the following values:

- **DB_REPMGR_ACKS_ALL**

  The master should wait until all replication clients have acknowledged each permanent replication message.

- **DB_REPMGR_ACKS_ALL_PEERS**

  The master should wait until all electable peers have acknowledged each permanent replication message.

- **DB_REPMGR_ACKS_NONE**

  The master should not wait for any client replication message acknowledgments.

- **DB_REPMGR_ACKS_ONE**

  The master should wait until at least one client site has acknowledged each permanent replication message.

- **DB_REPMGR_ACKS_ONE_PEER**

  The master should wait until at least one electable peer has acknowledged each permanent replication message.

- **DB_REPMGR_ACKS_QUORUM**

  The master should wait until it has received acknowledgements from the minimum number of electable peers sufficient to ensure that the effect of the permanent record remains durable if an election is held. This is the default acknowledgement policy.

Errors

The `DB_ENV->repmgr_set_ack_policy()` method may fail and return one of the following non-zero errors:

EINVAL

If this method is called from a base replication API application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_set_local_site()

```
#include <db.h>

int
DB_ENV->repmgr_set_local_site(DB_ENV *env,
    const char *host, u_int port, u_int32_t flags);
```

The `DB_ENV->repmgr_set_local_site()` method specifies the host identification string and port number for the local system.

The `DB_ENV->repmgr_set_local_site()` method may not be called after the DB_ENV->repmgr_start() (page 502) method is called.

The `DB_ENV->repmgr_set_local_site()` method returns a non-zero error value on failure and 0 on success.

Parameters

host

The site's host identification string, generally a TCP/IP host name.

port

The port number on which the local site is listening.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_ENV->repmgr_set_local_site()` method may fail and return one of the following non-zero errors:

EINVAL

If this method is called from a Base API application; if the method was called after replication is started with a call to the DB_ENV->repmgr_start() (page 502) method; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_site_list()

```
#include <db.h>

int
DB_ENV->repmgr_site_list(DB_ENV *env,
    u_int *countp, DB_REPMGR_SITE **listp);
```

The `DB_ENV->repmgr_site_list()` method returns the status of the sites currently known by the Replication Manager.

The `DB_ENV->repmgr_site_list()` method creates a statistical structure of type `DB_REPMGR_SITE` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following `DB_REPMGR_SITE` fields will be filled in:

- **int eid;**

  Environment ID assigned by the Replication Manager. This is the same value that is passed to the application's event notification function for the DB_EVENT_REP_NEWMASTER event.

- **char host[];**

  Null-terminated host name.

- **u_int port;**

  TCP/IP port number.

- **u_int32_t status;**

  Zero (if unknown), or one of the following constants: `DB_REPMGR_CONNECTED`, `DB_REPMGR_DISCONNECTED`.

- **u_int32_t flags;**

  Zero or a bitwise inclusive OR of the `DB_REPMGR_ISPEER` constant. The `DB_REPMGR_ISPEER` value means that the site is a possible client-to-client peer.

The `DB_ENV->repmgr_site_list()` method may be called at any time during the life of the application.

The `DB_ENV->repmgr_site_list()` method returns a non-zero error value on failure and 0 on success.

Parameters

countp

A count of the returned structures will be stored into the memory referenced by **countp**.

listp

A reference to an array of structures will be stored into the memory referenced by **listp**.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_start()

```
#include <db.h>

int
DB_ENV->repmgr_start(DB_ENV *env, int nthreads, u_int32_t flags);
```

The `DB_ENV->repmgr_start()` method starts the Replication Manager.

There are two ways to build Berkeley DB replication applications: the most common approach is to use the Berkeley DB library Replication Manager, where the Berkeley DB library manages the replication group, including network transport, all replication message processing and acknowledgment, and group elections. Applications using the Replication Manager generally make the following calls:

1. Call DB_ENV->repmgr_set_local_site() (page 499) to configure the local site in the replication group.
2. Call DB_ENV->repmgr_add_remote_site() (page 494) to configure the remote site(s) in the replication group.
3. Call DB_ENV->repmgr_set_ack_policy() (page 497) to configure the message acknowledgment policy which best supports the replication group's transactional needs.
4. Call DB_ENV->rep_set_priority() (page 475) to configure the local site's election priority.
5. Call `DB_ENV->repmgr_start()` to start the replication application.

For more information on building Replication Manager applications, please see the *Replication Getting Started Guide* included in the Berkeley DB documentation.

Applications with special needs (for example, applications using network protocols not supported by the Berkeley DB Replication Manager), must perform additional configuration and call other Berkeley DB replication Base API methods. For more information on building Base API applications, please see the Base API Methods section in the *Berkeley DB Programmer's Reference Guide*.

Starting the Replication Manager consists of opening the TCP/IP listening socket to accept incoming connections, and starting all necessary background threads. When multiple processes share a database environment, only one process can open the listening socket; the `DB_ENV->repmgr_start()` method automatically opens the socket in the first process to call it, and skips this step in the later calls from other processes.

The `DB_ENV->repmgr_start()` method may not be called before the DB_ENV->open() (page 231) method is called to open the local environment and the DB_ENV->repmgr_set_local_site() (page 499) method is called to configure the local site. In addition, the local environment must be opened with the DB_THREAD flag set.

The `DB_ENV->repmgr_start()` method will return `DB_REP_IGNORE` as an informational, non-error return code, if another process has previously become the TCP/IP listener (though the current call has nevertheless successfully started Replication Manager's background threads).

Unless otherwise specified, the `DB_ENV->repmgr_start()` method returns a non-zero error value on failure and 0 on success.

Parameters

nthreads

Specify the number of threads of control created and dedicated to processing replication messages. In addition to these message processing threads, the Replication Manager creates and manages a few of its own threads of control.

flags

The **flags** parameter must be set to one of the following values:

- `DB_REP_MASTER`

  Start as a master site, and do not call for an election. Note there must never be more than a single master in any replication group, and only one site at a time should ever be started with the DB_REP_MASTER flag specified.

- `DB_REP_CLIENT`

  Start as a client site, and do not call for an election.

- `DB_REP_ELECTION`

  Start as a client, and call for an election if no master is found.

Errors

The `DB_ENV->repmgr_start()` method may fail and return one of the following non-zero errors:

EINVAL

If the database environment was not already opened or was opened without the `DB_THREAD` flag set; a local site has not already been configured, this method is called from a Base API application; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_stat()

```
#include <db.h>

int
DB_ENV->repmgr_stat(DB_ENV *env, DB_REPMGR_STAT **statp,
    u_int32_t flags);
```

The `DB_ENV->repmgr_stat()` method returns the Replication Manager statistics.

The `DB_ENV->repmgr_stat()` method creates a statistical structure of type `DB_REPMGR_STAT` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following `DB_REPMGR_STAT` fields will be filled in:

- **uintmax_t st_connect_fail;**

  The number of times an attempt to open a new TCP/IP connection failed.

- **uintmax_t st_connection_drop;**

  The number of times an existing TCP/IP connection failed.

- **uintmax_t st_msgs_dropped;**

  The number of outgoing messages that were completely dropped, because the outgoing message queue was full. (Berkeley DB replication is tolerant of dropped messages, and will automatically request retransmission of any missing messages as needed.)

- **uintmax_t st_msgs_queued;**

  The number of outgoing messages which could not be transmitted immediately, due to a full network buffer, and had to be queued for later delivery.

- **uintmax_t st_perm_failed;**

  The number of times a message critical for maintaining database integrity (for example, a transaction commit), originating at this site, did not receive sufficient acknowledgement from clients, according to the configured acknowledgement policy and acknowledgement timeout.

- **uintmax_t st_elect_threads;**

  The number of currently active election threads.

- **uintmax_t st_max_elect_threads;**

The number of election threads for which space is reserved.

The `DB_ENV->repmgr_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->repmgr_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->repmgr_stat()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called before DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->repmgr_stat_print()

```
#include <db.h>

int
DB_ENV->repmgr_stat_print(DB_ENV *env, u_int32_t flags);
```

The `DB_ENV->repmgr_stat_print()` method displays the Replication Manager statistical information, as described for the `DB_ENV->repmgr_stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->repmgr_stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->repmgr_stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

Errors

The `DB_ENV->repmgr_stat_print()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called before DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV

See Also

Replication and Related Methods (page 452)

DB_ENV->txn_applied()

```
#include <db.h>

int
DB_ENV->txn_applied(DB_ENV *env, DB_TXN_TOKEN *token,
                    db_timeout_t timeout, u_int32_t flags);
```

The `DB_ENV->txn_applied()` method checks to see if a specified transaction has been replicated from the master of a replication group. It may be called by applications using either the Base API or the Replication Manager.

If the transaction has not yet arrived, this method will block for the amount of time specified on the `timeout` parameter while it waits for the result to be determined. For more information, please refer to the Read your writes consistency section in the *Berkeley DB Programmer's Reference Guide*.

The `DB_ENV->txn_applied()` method may not be called before the DB_ENV->open() (page 231) method.

The `DB_ENV->txn_applied()` method returns a non-zero error on failure and 0 to indicate that the specified transaction has been applied at the local site. It may also return one of the following non-zero return codes:

- `DB_TIMEOUT`

  Returned if the specified transaction has not yet arrived at the calling site, but can be expected to arrive soon. If a non-zero timeout parameter is given, the this method always waits for the specified amount of time before returning `DB_TIMEOUT`.

- `DB_NOTFOUND`

  Returned if the transaction is expected to never arrive. This occurs if the transaction has not been applied at the local site because the transaction has been rolled back due to a master takeover.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

token

A pointer to a buffer containing a copy of a commit token previously generated at the replication group's master environment. Commit tokens are created using the DB_TXN->set_commit_token() (page 509) method.

timeout

Specifies the maximum time to wait for the transaction to arrive by replication, expressed in microseconds. To check the status of the transaction without waiting, provide a timeout value of 0.

Errors

The `DB_ENV->txn_applied()` method may fail and return one of the following non-zero errors:

DB_KEYEMPTY

The specified token was generated by a transaction that did not modify the database environment (for example, a read-only transaction).

DB_LOCK_DEADLOCK

While waiting for the result to be determined, the API became locked out due to replication role change and/or master/client synchronization. The application should abort in-flight transactions, pause briefly, and then retry.

EINVAL

If the specified token was generated from a non-replicated database environment.

Class

DB_ENV

See Also

Transaction Subsystem and Related Methods (page 534), Replication and Related Methods (page 452)

DB_TXN->set_commit_token()

```
#include <db.h>

int
DB_TXN->set_commit_token(DB_TXN *txn, DB_TXN_TOKEN *buffer);
```

The `DB_TXN->set_commit_token()` method configures the transaction for commit token generation, and accepts the address of an application-supplied buffer to receive the token. The actual generation of the token contents does not occur until commit time.

Commit tokens are used to enable some consistency guarantees for replicated applications. Please see the Read your writes consistency section in the *Berkeley DB Programmer's Reference Guide* for more information.

The `DB_TXN->set_commit_token()` method may be called at any time after the DB_ENV->txn_begin() (page 544) method has been called, and before DB_TXN->commit() (page 555) has been called.

The `DB_TXN->set_commit_token()` method returns a non-zero error value on failure and 0 on success.

Parameters

buffer

The address of an application-supplied buffer. The buffer memory must remain available, and will be filled in later by Berkeley DB, at the time of the commit() call.

Errors

The `DB_TXN->set_commit_token()` method may fail and return one of the following non-zero errors:

EINVAL

If the transaction is a nested transaction; if this method is called on a replication client; if the database environment is not configured for logging.

Class

DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534), Replication and Related Methods (page 452)

Chapter 11. The DB_SEQUENCE Handle

Sequences provide an arbitrary number of persistent objects that return an increasing or decreasing sequence of integers. Opening a sequence handle associates it with a record in a database. The handle can maintain a cache of values from the database so that a database update is not needed as the application allocates a value.

A sequence is stored as a record pair in a database. The database may be of any type, but must not have been configured to support duplicate data items. The sequence is referenced by the key used when the sequence is created, therefore the key must be compatible with the underlying access method. If the database stores fixed-length records, the record size must be at least 64 bytes long.

You create a sequence using the db_sequence_create (page 512) method.

For more information on sequences, see the *Berkeley DB Programmer's Reference Guide* guide.

Sequences and Related Methods

Sequences and Related Methods	Description
db_sequence_create	Create a sequence handle
DB_SEQUENCE->close()	Close a sequence
DB_SEQUENCE->get()	Get the next sequence element(s)
DB_SEQUENCE->get_dbp()	Return a handle for the underlying sequence database
DB_SEQUENCE->get_key()	Return the key for a sequence
DB_SEQUENCE->initial_value()	Set the initial value of a sequence
DB_SEQUENCE->open()	Open a sequence
DB_SEQUENCE->remove()	Remove a sequence
DB_SEQUENCE->stat()	Return sequence statistics
DB_SEQUENCE->stat_print()	Print sequence statistics
Sequences Configuration	
DB_SEQUENCE->set_cachesize(), DB_SEQUENCE->get_cachesize()	Set/get the cache size of a sequence
DB_SEQUENCE->set_flags(), DB_SEQUENCE->get_flags()	Set/get the flags for a sequence
DB_SEQUENCE->set_range(), DB_SEQUENCE->get_range()	Set/get the range for a sequence

db_sequence_create

```
#include <db.h>

int db_sequence_create(DB_SEQUENCE **seq, DB *db, u_int32_t flags);
```

Creates a sequence handle, which can then be opened with DB_SEQUENCE->open() (page 523).

DB_SEQUENCE handles are free-threaded if the DB_THREAD flag is specified to the DB_SEQUENCE->open() (page 523) method when the sequence is opened. Once the DB_SEQUENCE->close() (page 514) or DB_SEQUENCE->remove() (page 525) methods are called, the handle can not be accessed again, regardless of the method's return.

Each handle opened on a sequence may maintain a separate cache of values which are returned to the application using the DB_SEQUENCE->get() (page 515) method either singly or in groups depending on its **delta** parameter.

Calling the DB_SEQUENCE->close() (page 514) or DB_SEQUENCE->remove() (page 525) methods discards this handle.

db_sequence_create() method returns a non-zero error value on failure and 0 on success.

Parameters

seq

The **seq** parameter references the memory into which the returned structure pointer is stored.

db

The **db** parameter is an open database handle which holds the persistent data for the sequence. The database may be of any type, but must not have been configured to support duplicate data items.

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The db_sequence_create method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->close()

```
#include <db.h>

int
DB_SEQUENCE->close(DB_SEQUENCE *seq, u_int32_t flags);
```

The `DB_SEQUENCE->close()` method closes the sequence handle. Any unused cached values are lost.

The DB_SEQUENCE handle may not be accessed again after `DB_SEQUENCE->close()` is called, regardless of its return.

The `DB_SEQUENCE->close()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_SEQUENCE->close()` method method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get()

```
#include <db.h>

int
DB_SEQUENCE->get(DB_SEQUENCE *seq,
    DB_TXN *txnid, int32_t delta, db_seq_t *retp, u_int32_t flags);
```

The `DB_SEQUENCE->get()` method returns the next available element in the sequence and changes the sequence value by **delta**. The value of **delta** must be greater than zero. If there are enough cached values in the sequence handle then they will be returned. Otherwise the next value will be fetched from the database and incremented (decremented) by enough to cover the **delta** and the next batch of cached values.

For maximum concurrency a non-zero cache size should be specified prior to opening the sequence handle and DB_TXN_NOSYNC should be specified for each `DB_SEQUENCE->get()` method call.

By default, sequence ranges do not wrap; to cause the sequence to wrap around the beginning or end of its range, specify the DB_SEQ_WRAP flag to the DB_SEQUENCE->set_flags() (page 528) method.

The `DB_SEQUENCE->get()` method will return EINVAL if the record in the database is not a valid sequence record, or the sequence has reached the beginning or end of its range and is not configured to wrap.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected. No **txnid** handle may be specified if the sequence handle was opened with a non-zero cache size.

If the underlying database handle was opened in a transaction, calling `DB_SEQUENCE->get()` may result in changes to the sequence object; these changes will be automatically committed in a transaction internal to the Berkeley DB library. If the thread of control calling `DB_SEQUENCE->get()` has an active transaction, which holds locks on the same database as the one in which the sequence object is stored, it is possible for a thread of control calling `DB_SEQUENCE->get()` to self-deadlock because the active transaction's locks conflict with the internal transaction's locks. For this reason, it is often preferable for sequence objects to be stored in their own database.

delta

Specifies the amount to increment or decrement the sequence.

retp

retp points to the memory to hold the return value from the sequence.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- DB_TXN_NOSYNC

  If the operation is implicitly transaction protected (the **txnid** argument is NULL but the operation occurs to a transactional database), do not synchronously flush the log when the transaction commits.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get_cachesize()

```
#include <db.h>

int
DB_SEQUENCE->get_cachesize(DB_SEQUENCE *seq, u_int32_t *sizep);
```

The `DB_SEQUENCE->get_cachesize()` method returns the current cache size.

The `DB_SEQUENCE->get_cachesize()` method may be called at any time during the life of the application.

The `DB_SEQUENCE->get_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

sizep

The `DB_SEQUENCE->get_cachesize()` method returns the current cache size in **sizep**.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get_dbp()

```
#include <db.h>

int
DB_SEQUENCE->get_dbp(DB_SEQUENCE *seq, DB **dbp);
```

The DB_SEQUENCE->get_dbp() method returns the database handle used by the sequence.

The DB_SEQUENCE->get_dbp() method may be called at any time during the life of the application.

The DB_SEQUENCE->get_dbp() method returns a non-zero error value on failure and 0 on success.

Parameters

dbp

The **dbp** parameter references memory into which a pointer to the database handle is copied.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get_flags()

```
#include <db.h>

int
DB_SEQUENCE->get_flags(DB_SEQUENCE *seq, u_int32_t *flagsp);
```

The `DB_SEQUENCE->get_flags()` method returns the current flags.

The `DB_SEQUENCE->get_flags()` method may be called at any time during the life of the application.

The `DB_SEQUENCE->get_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flagsp

The `DB_SEQUENCE->get_flags()` method returns the current flags in **flagsp**.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get_key()

```
#include <db.h>

int
DB_SEQUENCE->get_key(DB_SEQUENCE *seq, DBT *key);
```

The `DB_SEQUENCE->get_key()` method returns the key for the sequence.

The `DB_SEQUENCE->get_key()` method may be called at any time during the life of the application.

The `DB_SEQUENCE->get_key()` method returns a non-zero error value on failure and 0 on success.

Parameters

key

The **key** parameter references memory into which a pointer to the key data is copied.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->get_range()

```
#include <db.h>

int
DB_SEQUENCE->get_range(DB_SEQUENCE *seq, db_seq_t *minp, db_seq_t *maxp);
```

The `DB_SEQUENCE->get_range()` method returns the range of values in the sequence.

The `DB_SEQUENCE->get_range()` method may be called at any time during the life of the application.

The `DB_SEQUENCE->get_range()` method returns a non-zero error value on failure and 0 on success.

Parameters

minp

The `DB_SEQUENCE->get_range()` method returns the minimum value in **minp**.

maxp

The `DB_SEQUENCE->get_range()` method returns the maximum value in **maxp**.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->initial_value()

```
#include <db.h>

int
DB_SEQUENCE->initial_value(DB_SEQUENCE *seq, db_seq_t value);
```

Set the initial value for a sequence. This call is only effective when the sequence is being created.

The `DB_SEQUENCE->initial_value()` method may not be called after the DB_SEQUENCE->open() (page 523) method is called.

The `DB_SEQUENCE->initial_value()` method returns a non-zero error value on failure and 0 on success.

Parameters

value

The initial value to set.

Errors

The `DB_SEQUENCE->initial_value()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->open()

```
#include <db.h>

int
DB_SEQUENCE->open(DB_SEQUENCE *seq, DB_TXN *txnid, DBT *key,
                  u_int32_t flags);
```

The `DB_SEQUENCE->open()` method opens the sequence represented by the **key**. The key must be compatible with the underlying database specified in the corresponding call to db_sequence_create (page 512).

The `DB_SEQUENCE->open()` method returns a non-zero error value on failure and 0 on success.

Parameters

key

The **key** specifies which record in the database stores the persistent sequence data.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_CREATE`

  Create the sequence. If the sequence does not already exist and the `DB_CREATE` flag is not specified, the `DB_SEQUENCE->open()` method will fail.

- `DB_EXCL`

  Return an error if the sequence already exists. This flag is only meaningful when specified with the `DB_CREATE` flag.

- `DB_THREAD`

  Cause the DB_SEQUENCE handle returned by `DB_SEQUENCE->open()` to be *free-threaded*; that is, usable by multiple threads within a single address space. Note that if multiple threads create multiple sequences using the same database handle that handle must have been opened specifying this flag.

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected. Transactionally protected operations on a DB_SEQUENCE handle require the DB_SEQUENCE handle itself be transactionally protected during its open if the open creates the sequence.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->remove()

```
#include <db.h>

int
DB_SEQUENCE->remove(DB_SEQUENCE *seq, DB_TXN *txnid, u_int32_t flags);
```

The `DB_SEQUENCE->remove()` method removes the sequence from the database. This method should not be called if there are other open handles on this sequence.

The DB_SEQUENCE handle may not be accessed again after `DB_SEQUENCE->remove()` is called, regardless of its return.

The `DB_SEQUENCE->remove()` method returns a non-zero error value on failure and 0 on success.

Parameters

txnid

If the operation is part of an application-specified transaction, the **txnid** parameter is a transaction handle returned from DB_ENV->txn_begin() (page 544); if the operation is part of a Berkeley DB Concurrent Data Store group, the **txnid** parameter is a handle returned from DB_ENV->cdsgroup_begin() (page 536); otherwise NULL. If no transaction handle is specified, but the operation occurs in a transactional database, the operation will be implicitly transaction protected.

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_TXN_NOSYNC`

  If the operation is implicitly transaction protected (the **txnid** argument is NULL but the operation occurs to a transactional database), do not synchronously flush the log when the transaction commits.

Errors

The `DB_SEQUENCE->remove()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->set_cachesize()

```
#include <db.h>

int
DB_SEQUENCE->set_cachesize(DB_SEQUENCE *seq, int32_t size);
```

Configure the number of elements cached by a sequence handle.

The `DB_SEQUENCE->set_cachesize()` method may not be called after the DB_SEQUENCE->open() (page 523) method is called.

The `DB_SEQUENCE->set_cachesize()` method returns a non-zero error value on failure and 0 on success.

Parameters

size

The number of elements in the cache.

Errors

The `DB_SEQUENCE->set_cachesize()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->set_flags()

```
#include <db.h>

int
DB_SEQUENCE->set_flags(DB_SEQUENCE *seq, u_int32_t flags);
```

Configure a sequence. The flags are only effective when creating a sequence. Calling `DB_SEQUENCE->set_flags()` is additive; there is no way to clear flags.

The `DB_SEQUENCE->set_flags()` method may not be called after the DB_SEQUENCE->open() (page 523) method is called.

The `DB_SEQUENCE->set_flags()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_SEQ_DEC`

  Specify that the sequence should be decremented.

- `DB_SEQ_INC`

  Specify that the sequence should be incremented. This is the default.

- `DB_SEQ_WRAP`

  Specify that the sequence should wrap around when it is incremented (decremented) past the specified maximum (minimum) value.

Errors

The `DB_SEQUENCE->set_flags()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->set_range()

```
#include <db.h>

int
DB_SEQUENCE->set_range(DB_SEQUENCE *seq, db_seq_t min, db_seq_t max);
```

Configure a sequence range. This call is only effective when the sequence is being created. The range is limited to a signed 64 bit integer.

The `DB_SEQUENCE->set_range()` method may not be called after the DB_SEQUENCE->open() (page 523) method is called.

The `DB_SEQUENCE->set_range()` method returns a non-zero error value on failure and 0 on success.

Parameters

min

Specifies the minimum value for the sequence.

max

Specifies the maximum value for the sequence.

Errors

The `DB_SEQUENCE->set_range()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->stat()

```
#include <db.h>

int
DB_SEQUENCE->stat(DB_SEQUENCE *db, DB_SEQUENCE_STAT **spp,
                u_int32_t flags);
```

The DB_SEQUENCE->stat() method creates a statistical structure and copies a pointer to it into user-specified memory locations. Specifically, if **spp** is non-NULL, a pointer to the statistics for the database are copied into the memory location to which it refers.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

In the presence of multiple threads or processes accessing an active sequence, the information returned by DB_SEQUENCE->stat() may be out-of-date.

The DB_SEQUENCE->stat() method cannot be transaction-protected. For this reason, it should be called in a thread of control that has no open cursors or active transactions.

The DB_SEQUENCE->stat() method returns a non-zero error value on failure and 0 on success.

The statistics are stored in a structure of type DB_SEQUENCE_STAT. The following fields will be filled in:

- **uintmax_t st_wait;**

  The number of times a thread of control was forced to wait on the handle mutex.

- **uintmax_t st_nowait;**

  The number of times that a thread of control was able to obtain handle mutex without waiting.

- **db_seq_t st_current;**

  The current value of the sequence in the database.

- **db_seq_t st_value;**

  The current cached value of the sequence.

- **db_seq_t st_last_value;**

  The last cached value of the sequence.

- **db_seq_t st_min;**

  The minimum permitted value of the sequence.

- **db_seq_t st_max;**

  The maximum permitted value of the sequence.

- **int32_t st_cache_size;**

  The number of values that will be cached in this handle.

- **u_int32_t st_flags;**

  The flags value for the sequence.

Parameters

flags

The **flags** parameter must be set by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_CLEAR`

  Reset statistics after printing their values.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

DB_SEQUENCE->stat_print()

```
#include <db.h>

int
DB_SEQUENCE->stat_print(DB_SEQUENCE *db, u_int32_t flags);
```

The `DB_SEQUENCE->stat_print()` method prints diagnostic information to the output channel described by the DB_ENV->set_msgfile() (page 273) method.

The `DB_SEQUENCE->stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_CLEAR`

  Reset statistics after printing their values.

Class

DB_SEQUENCE

See Also

Sequences and Related Methods (page 511)

Chapter 12. The DB_TXN Handle

```
#include <db.h>

typedef struct __db_txn DB_TXN;
```

The DB_TXN object is the handle for a transaction. Methods of the DB_TXN handle are used to configure, abort and commit the transaction. DB_TXN handles are provided to DB methods in order to transactionally protect those database operations.

DB_TXN handles are not free-threaded; transactions handles may be used by multiple threads, but only serially, that is, the application must serialize access to the DB_TXN handle. Once the DB_TXN->abort() (page 554) or DB_TXN->commit() (page 555) methods are called, the handle may not be accessed again, regardless of the method's return. In addition, parent transactions may not issue any Berkeley DB operations while they have active child transactions (child transactions that have not yet been committed or aborted) except for DB_ENV->txn_begin() (page 544), DB_TXN->abort() (page 554) and DB_TXN->commit() (page 555).

Transaction Subsystem and Related Methods

Transaction Subsystem and Related Methods	Description
DB_ENV->txn_recover()	Distributed transaction recovery
DB_ENV->txn_checkpoint()	Checkpoint the transaction subsystem
DB_ENV->txn_stat()	Return transaction subsystem statistics
DB_ENV->txn_stat_print()	Print transaction subsystem statistics
DB_TXN->set_timeout()	Set transaction timeout
Transaction Subsystem Configuration	
DB_ENV->set_timeout(), DB_ENV->get_timeout()	Set/get lock and transaction timeout
DB->get_transactional()	Does the DB have transaction support
DB_ENV->cdsgroup_begin()	Get a locker ID in Berkeley DB Concurrent Data Store
DB_ENV->set_tx_max(), DB_ENV->get_tx_max()	Set/get maximum number of transactions
DB_ENV->set_tx_timestamp(), DB_ENV->get_tx_timestamp()	Set/get recovery timestamp
Transaction Operations	
DB_ENV->txn_begin()	Begin a transaction
DB_TXN->abort()	Abort a transaction
DB_TXN->commit()	Commit a transaction
DB_TXN->discard()	Discard a prepared but not resolved transaction handle
DB_TXN->id()	Return a transaction's ID
DB_TXN->prepare()	Prepare a transaction for commit
DB_TXN->set_name(), DB_TXN->get_name()	Associate a string with a transaction
DB_TXN->set_priority(), DB_TXN->get_priority()	Set/get transaction's priority

DB->get_transactional()

```
#include <db.h>

int
DB->get_transactional(DB *db);
```

The `DB->get_transactional()` method returns non-zero if the DB handle has been opened in a transactional mode.

The `DB->get_transactional()` method may be called at any time during the life of the application.

The `DB->get_transactional()` method returns a non-zero error value on failure and 0 on success.

Class

DB

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->cdsgroup_begin()

```
#include <db.h>

int
DB_ENV->cdsgroup_begin(DB_ENV *dbenv, DB_TXN **tid);
```

The `DB_ENV->cdsgroup_begin()` method allocates a locker ID in an environment configured for Berkeley DB Concurrent Data Store applications. It copies a pointer to a DB_TXN that uniquely identifies the locker ID into the memory to which **tid** refers. Calling the DB_TXN->commit() (page 555) method will discard the allocated locker ID.

See Berkeley DB Concurrent Data Store applications for more information about when this is required.

The `DB_ENV->cdsgroup_begin()` method may be called at any time during the life of the application.

The `DB_ENV->cdsgroup_begin()` method returns a non-zero error value on failure and 0 on success.

Errors

The `DB_ENV->cdsgroup_begin()` method may fail and return one of the following non-zero errors:

ENOMEM

The maximum number of lockers has been reached.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->get_tx_max()

```
#include <db.h>

int
DB_ENV->get_tx_max(DB_ENV *dbenv, u_int32_t *tx_maxp);
```

The `DB_ENV->get_tx_max()` method returns the maximum number of active transactions currently configured for the environment. You can manage this value using the DB_ENV->set_tx_max() (page 539) method.

The `DB_ENV->get_tx_max()` method may be called at any time during the life of the application.

The `DB_ENV->get_tx_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

tx_maxp

The `DB_ENV->get_tx_max()` method returns the number of active transactions in **tx_maxp**.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534), DB_ENV->set_tx_max() (page 539)

DB_ENV->get_tx_timestamp()

```
#include <db.h>

int
DB_ENV->get_tx_timestamp(DB_ENV *dbenv, time_t *timestampp);
```

The `DB_ENV->get_tx_timestamp()` method returns the recovery timestamp. This value can be modified using the DB_ENV->set_tx_timestamp() (page 541) method.

The `DB_ENV->get_tx_timestamp()` method may be called at any time during the life of the application.

The `DB_ENV->get_tx_timestamp()` method returns a non-zero error value on failure and 0 on success.

Parameters

timestampp

The `DB_ENV->get_tx_timestamp()` method returns the recovery timestamp in **timestampp**.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534), DB_ENV->set_tx_timestamp() (page 541)

DB_ENV->set_tx_max()

```
#include <db.h>

int
DB_ENV->set_tx_max(DB_ENV *dbenv, u_int32_t max);
```

Configure the Berkeley DB database environment to support at least **max** active transactions. This value bounds the size of the memory allocated for transactions. Child transactions are counted as active until they either commit or abort.

Transactions that update multiversion databases are not freed until the last page version that the transaction created is flushed from cache. This means that applications using multi-version concurrency control may need a transaction for each page in cache, in the extreme case.

When all of the memory available in the database environment for transactions is in use, calls to DB_ENV->txn_begin() (page 544) will fail (until some active transactions complete). If `DB_ENV->set_tx_max()` is never called, the database environment is configured to support at least 100 active transactions.

The database environment's number of active transactions may also be configured using the environment's DB_CONFIG file. The syntax of the entry in that file is a single line with the string "set_tx_max", one or more whitespace characters, and the number of transactions. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DB_ENV->set_tx_max()` method configures a database environment, not only operations performed using the specified DB_ENV handle.

The `DB_ENV->set_tx_max()` method may not be called after the DB_ENV->open() (page 231) method is called. If the database environment already exists when DB_ENV->open() (page 231) is called, the information specified to `DB_ENV->set_tx_max()` will be ignored.

The `DB_ENV->set_tx_max()` method returns a non-zero error value on failure and 0 on success.

Parameters

max

The **max** parameter configures the minimum number of simultaneously active transactions supported by Berkeley DB database environment.

Errors

The `DB_ENV->set_tx_max()` method may fail and return one of the following non-zero errors:

EINVAL

If the method was called after DB_ENV->open() (page 231) was called; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->set_tx_timestamp()

```
#include <db.h>

int
DB_ENV->set_tx_timestamp(DB_ENV *dbenv, time_t *timestamp);
```

Recover to the time specified by **timestamp** rather than to the most current possible date.

Once a database environment has been upgraded to a new version of Berkeley DB involving a log format change (see Upgrading Berkeley DB installations), it is no longer possible to recover to a specific time before that upgrade.

The `DB_ENV->set_tx_timestamp()` method configures operations performed using the specified DB_ENV handle, not all operations performed on the underlying database environment.

The `DB_ENV->set_tx_timestamp()` method may not be called after the DB_ENV->open() (page 231) method is called.

The `DB_ENV->set_tx_timestamp()` method returns a non-zero error value on failure and 0 on success.

Parameters

timestamp

The **timestamp** parameter references the memory location where the recovery timestamp is located.

The **timestamp** parameter should be the number of seconds since 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time; that is, the Epoch.

Errors

The `DB_ENV->set_tx_timestamp()` method may fail and return one of the following non-zero errors:

EINVAL

If it is not possible to recover to the specified time using the log files currently present in the environment; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->txn_recover()

```
#include <db.h>

int
DB_ENV->txn_recover(DB_ENV *dbenv, DB_PREPLIST preplist[],
    long count, long *retp, u_int32_t flags);
```

Database environment recovery restores transactions that were prepared, but not yet resolved at the time of the system shut down or crash, to their state prior to the shut down or crash, including any locks previously held. The `DB_ENV->txn_recover()` method returns a list of those prepared transactions.

The `DB_ENV->txn_recover()` method should only be called after the environment has been recovered.

Multiple threads of control may call `DB_ENV->txn_recover()`, but only one thread of control may resolve each returned transaction, that is, only one thread of control may call DB_TXN->commit() (page 555) or DB_TXN->abort() (page 554) on each returned transaction. Callers of `DB_ENV->txn_recover()` must call DB_TXN->discard() (page 557) to discard each transaction they do not resolve.

On return from `DB_ENV->txn_recover()`, the **preplist** parameter will be filled in with a list of transactions that must be resolved by the application (committed, aborted or discarded). The **preplist** parameter is a structure of type `DB_PREPLIST`; the following `DB_PREPLIST` fields will be filled in:

- **DB_TXN * txn;**

  The transaction handle for the transaction.

- **u_int8_t gid[DB_XIDDATASIZE];**

  The global transaction ID for the transaction. The global transaction ID is the one specified when the transaction was prepared. The application is responsible for ensuring uniqueness among global transaction IDs.

The `DB_ENV->txn_recover()` method returns a non-zero error value on failure and 0 on success.

Parameters

preplist

The **preplist** parameter references memory into which the list of transactions to be resolved by the application is copied.

count

The **count** parameter specifies the number of available entries in the passed-in **preplist** array. The **retp** parameter returns the number of entries `DB_ENV->txn_recover()` has filled in, in the array.

flags

The **flags** parameter must be set to one of the following values:

- `DB_FIRST`

  Begin returning a list of prepared, but not yet resolved transactions. Specifying this flag begins a new pass over all prepared, but not yet completed transactions, regardless of whether they have already been returned in previous calls to `DB_ENV->txn_recover.()` Calls to `DB_ENV->txn_recover()` from different threads of control should not be intermixed in the same environment.

- `DB_NEXT`

  Continue returning a list of prepared, but not yet resolved transactions, starting where the last call to `DB_ENV->txn_recover()` left off.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->txn_begin()

```
#include <db.h>

int
DB_ENV->txn_begin(DB_ENV *env,
    DB_TXN *parent, DB_TXN **tid, u_int32_t flags);
```

The `DB_ENV->txn_begin()` method creates a new transaction in the environment and copies a pointer to a DB_TXN that uniquely identifies it into the memory to which **tid** refers. Calling the DB_TXN->abort() (page 554), DB_TXN->commit() (page 555) or DB_TXN->discard() (page 557) methods will discard the returned handle.

Note

Transactions may only span threads if they do so serially; that is, each transaction must be active in only a single thread of control at a time. This restriction holds for parents of nested transactions as well; no two children may be concurrently active in more than one thread of control at any one time.

Note

Cursors may not span transactions; that is, each cursor must be opened and closed within a single transaction.

Note

A parent transaction may not issue any Berkeley DB operations — except for `DB_ENV->txn_begin()`, DB_TXN->abort() (page 554) and DB_TXN->commit() (page 555) — while it has active child transactions (child transactions that have not yet been committed or aborted).

The `DB_ENV->txn_begin()` method returns a non-zero error value on failure and 0 on success.

Parameters

parent

If the **parent** parameter is non-NULL, the new transaction will be a nested transaction, with the transaction indicated by **parent** as its parent. Transactions may be nested to any level. In the presence of distributed transactions and two-phase commit, only the parental transaction, that is a transaction without a **parent** specified, should be passed as an parameter to DB_TXN->prepare() (page 562).

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_READ_COMMITTED`

This transaction will have degree 2 isolation. This provides for cursor stability but not repeatable reads. Data items which have been previously read by this transaction may be deleted or modified by other transactions before this transaction completes.

- DB_READ_UNCOMMITTED

  This transaction will have degree 1 isolation. Read operations performed by the transaction may read modified but not yet committed data. Silently ignored if the DB_READ_UNCOMMITTED flag was not specified when the underlying database was opened.

- DB_TXN_NOSYNC

  Do not synchronously flush the log when this transaction commits or prepares. This means the transaction will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained but it is possible that this transaction may be undone during recovery.

  This behavior may be set for a Berkeley DB environment using the DB_ENV->set_flags() (page 260) method. Any value specified to this method overrides that setting.

- DB_TXN_NOWAIT

  If a lock is unavailable for any Berkeley DB operation performed in the context of this transaction, cause the operation to return DB_LOCK_DEADLOCK (or DB_LOCK_NOTGRANTED if the database environment has been configured using the DB_TIME_NOTGRANTED flag).

  This behavior may be set for a Berkeley DB environment using the DB_ENV->set_flags() (page 260) method. Any value specified to this method overrides that setting.

- DB_TXN_SNAPSHOT

  This transaction will execute with snapshot isolation. For databases with the DB_MULTIVERSION flag set, data values will be read as they are when the transaction begins, without taking read locks. Silently ignored for operations on databases with DB_MULTIVERSION not set on the underlying database (read locks are acquired).

  The error DB_LOCK_DEADLOCK will be returned from update operations if a snapshot transaction attempts to update data which was modified after the snapshot transaction read it.

- DB_TXN_SYNC

  Synchronously flush the log when this transaction commits or prepares. This means the transaction will exhibit all of the ACID (atomicity, consistency, isolation, and durability) properties.

  This behavior is the default for Berkeley DB environments unless the DB_TXN_NOSYNC flag was specified to the DB_ENV->set_flags() (page 260) method. Any value specified to this method overrides that setting.

- DB_TXN_WAIT

If a lock is unavailable for any Berkeley DB operation performed in the context of this transaction, wait for the lock.

This behavior is the default for Berkeley DB environments unless the DB_TXN_NOWAIT flag was specified to the DB_ENV->set_flags() (page 260) method. Any value specified to this method overrides that setting.

- DB_TXN_WRITE_NOSYNC

  Write, but do not synchronously flush, the log when this transaction commits. This means the transaction will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how often the system flushes dirty buffers to disk and how often the log is flushed or checkpointed.

  This behavior may be set for a Berkeley DB environment using the DB_ENV->set_flags() (page 260) method. Any value specified to this method overrides that setting.

Errors

The DB_ENV->txn_begin() method may fail and return one of the following non-zero errors:

ENOMEM

The maximum number of concurrent transactions has been reached.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->txn_checkpoint()

```
#include <db.h>

int
DB_ENV->txn_checkpoint(const DB_ENV *env,
    u_int32_t kbyte, u_int32_t min, u_int32_t flags);
```

If there has been any logging activity in the database environment since the last checkpoint, the `DB_ENV->txn_checkpoint()` method flushes the underlying memory pool, writes a checkpoint record to the log, and then flushes the log.

The `DB_ENV->txn_checkpoint()` method returns a non-zero error value on failure and 0 on success.

The `DB_ENV->txn_checkpoint()` method is the underlying method used by the db_checkpoint utility. See the db_checkpoint utility source code for an example of using `DB_ENV->txn_checkpoint()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

Parameters

kbyte

If the **kbyte** parameter is non-zero, a checkpoint will be done if more than **kbyte** kilobytes of log data have been written since the last checkpoint.

min

If the **min** parameter is non-zero, a checkpoint will be done if more than **min** minutes have passed since the last checkpoint.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_FORCE`

  Force a checkpoint record, even if there has been no activity since the last checkpoint.

Errors

The `DB_ENV->txn_checkpoint()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->txn_stat()

```
#include <db.h>

int
DB_ENV->txn_stat(DB_ENV *env, DB_TXN_STAT **statp, u_int32_t flags);
```

The `DB_ENV->txn_stat()` method returns the transaction subsystem statistics.

The `DB_ENV->txn_stat()` method creates a statistical structure of type `DB_TXN_STAT` and copies a pointer to it into a user-specified memory location.

Statistical structures are stored in allocated memory. If application-specific allocation routines have been declared (see DB_ENV->set_alloc() (page 239) for more information), they are used to allocate the memory; otherwise, the standard C library **malloc**(3) is used. The caller is responsible for deallocating the memory. To deallocate the memory, free the memory reference; references inside the returned memory need not be individually freed.

The following DB_TXN_STAT fields will be filled in:

- **DB_LSN st_last_ckp;**

  The LSN of the last checkpoint.

- **time_t st_time_ckp;**

  The time the last completed checkpoint finished (as the number of seconds since the Epoch, returned by the IEEE/ANSI Std 1003.1 (POSIX) **time** function).

- **u_int32_t st_last_txnid;**

  The last transaction ID allocated.

- **u_int32_t st_maxtxns;**

  The maximum number of active transactions configured.

- **u_int32_t st_nactive;**

  The number of transactions that are currently active.

- **u_int32_t st_nsnapshot;**

  The number of transactions on the snapshot list. These are transactions which modified a database opened with DB_MULTIVERSION, and which have committed or aborted, but the copies of pages they created are still in the cache.

- **u_int32_t st_maxnactive;**

  The maximum number of active transactions at any one time.

- **u_int32_t st_maxnsnapshot;**

  The maximum number of transactions on the snapshot list at any one time.

- **uintmax_t st_nbegins;**

  The number of transactions that have begun.

- **uintmax_t st_naborts;**

  The number of transactions that have aborted.

- **uintmax_t st_ncommits;**

  The number of transactions that have committed.

- **u_int32_t st_nrestores;**

  The number of transactions that have been restored.

- **roff_t st_regsize;**

  The size of the transaction region, in bytes.

- **uintmax_t st_region_wait;**

  The number of times that a thread of control was forced to wait before obtaining the transaction region mutex.

- **uintmax_t st_region_nowait;**

  The number of times that a thread of control was able to obtain the transaction region mutex without waiting.

- **DB_TXN_ACTIVE *st_txnarray;**

  A pointer to an array of **st_nactive** DB_TXN_ACTIVE structures, describing the currently active transactions. The following fields of the DB_TXN_ACTIVE structure will be filled in:

  - **u_int32_t txnid;**

    The transaction ID of the transaction.

  - **u_int32_t parentid;**

    The transaction ID of the parent transaction (or 0, if no parent).

  - **pid_t pid;**

    The process ID of the originator of the transaction.

  - **db_threadid_t tid;**

    The thread of control ID of the originator of the transaction.

  - **DB_LSN lsn;**

    The current log sequence number when the transaction was begun.

- **DB_LSN read_lsn;**

  The log sequence number of reads for snapshot transactions.

- **u_int32_t mvcc_ref;**

  The number of buffer copies created by this transaction that remain in cache.

- **u_int32_t priority;**

  This transaction's deadlock resolution priority.

- **u_int32_t status;**

  One of the following list of constants: TXN_ABORTED, TXN_COMMITTED, TXN_PREPARED, TXN_RUNNING.

- **u_int8_t gid[DB_GID_SIZE];**

  If the transaction was prepared using DB_TXN->prepare() (page 562), then **gid** contains the transaction's Global ID. Otherwise, **gid's** contents are undefined.

- **char name[];**

  If a name was specified for the transaction, up to the first 50 bytes of that name, followed by a nul termination byte.

The `DB_ENV->txn_stat()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->txn_stat()` method returns a non-zero error value on failure and 0 on success.

Parameters

statp

The **statp** parameter references memory into which a pointer to the allocated statistics structure is copied.

flags

The **flags** parameter must be set to 0 or the following value:

- `DB_STAT_CLEAR`

  Reset statistics after returning their values.

Errors

The `DB_ENV->txn_stat()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_ENV->txn_stat_print()

```
#include <db.h>

int
DB_ENV->txn_stat_print(DB_ENV *env, u_int32_t flags);
```

The `DB_ENV->txn_stat_print()` method displays the transaction subsystem statistical information, as described for the `DB_ENV->txn_stat()` method. The information is printed to a specified output channel (see the DB_ENV->set_msgfile() (page 273) method for more information), or passed to an application callback function (see the DB_ENV->set_msgcall() (page 271) method for more information).

The `DB_ENV->txn_stat_print()` method may not be called before the DB_ENV->open() (page 231) method is called.

The `DB_ENV->txn_stat_print()` method returns a non-zero error value on failure and 0 on success.

Parameters

flags

The **flags** parameter must be set to 0 or by bitwise inclusively **OR**'ing together one or more of the following values:

- `DB_STAT_ALL`

  Display all available information.

- `DB_STAT_CLEAR`

  Reset statistics after displaying their values.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->abort()

```
#include <db.h>

int
DB_TXN->abort(DB_TXN *tid);
```

The DB_TXN->abort() method causes an abnormal termination of the transaction. The log is played backward, and any necessary undo operations are done through the **tx_recover** function specified to DB_ENV->set_app_dispatch() (page 241). Before DB_TXN->abort() returns, any locks held by the transaction will have been released.

In the case of nested transactions, aborting a parent transaction causes all children (unresolved or not) of the parent transaction to be aborted.

All cursors opened within the transaction must be closed before the transaction is aborted. If they are not closed, they will be closed by this function. If a close operation fails, the rest of the cursors are closed, and the database environment is set to the panic state.

After DB_TXN->abort() has been called, regardless of its return, the DB_TXN handle may not be accessed again.

The DB_TXN->abort() method returns a non-zero error value on failure and 0 on success.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->commit()

```
#include <db.h>

int
DB_TXN->commit(DB_TXN *tid, u_int32_t flags);
```

The `DB_TXN->commit()` method ends the transaction.

In the case of nested transactions, if the transaction is a parent transaction, committing the parent transaction causes all unresolved children of the parent to be committed. In the case of nested transactions, if the transaction is a child transaction, its locks are not released, but are acquired by its parent. Although the commit of the child transaction will succeed, the actual resolution of the child transaction is postponed until the parent transaction is committed or aborted; that is, if its parent transaction commits, it will be committed; and if its parent transaction aborts, it will be aborted.

All cursors opened within the transaction must be closed before the transaction is committed. If they are not closed, they will be closed by this function. When the close operation for a cursor fails, the method returns a non-zero error value for the first instance of such an error, closes the rest of the cursors, and then aborts the transaction.

After `DB_TXN->commit()` has been called, regardless of its return, the DB_TXN handle may not be accessed again. If `DB_TXN->commit()` encounters an error, the transaction and all child transactions of the transaction are aborted.

The `DB_TXN->commit()` method returns a non-zero error value on failure and 0 on success. The errors values that this method returns include the error values of the `DBcursor->close()` method and the following:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Parameters

flags

The **flags** parameter must be set to 0 or one of the following values:

- `DB_TXN_NOSYNC`

Do not synchronously flush the log. This means the transaction will exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but it is possible that this transaction may be undone during recovery.

This behavior may be set for a Berkeley DB environment using the DB_ENV->set_flags() (page 260) method or for a single transaction using the DB_ENV->txn_begin() (page 544) method. Any value specified to this method overrides both of those settings.

- `DB_TXN_SYNC`

  Synchronously flush the log. This means the transaction will exhibit all of the ACID (atomicity, consistency, isolation, and durability) properties.

  This behavior is the default for Berkeley DB environments unless the DB_TXN_NOSYNC flag was specified to the DB_ENV->set_flags() (page 260) method. This behavior may also be set for a single transaction using the DB_ENV->txn_begin() (page 544) method. Any value specified to this method overrides both of those settings.

- `DB_TXN_WRITE_NOSYNC`

  Write but do not synchronously flush the log on transaction commit. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but not D (durability); that is, database integrity will be maintained, but if the system fails, it is possible some number of the most recently committed transactions may be undone during recovery. The number of transactions at risk is governed by how often the system flushes dirty buffers to disk and how often the log is checkpointed.

  This form of commit protects you against application crashes, but not against OS crashes. This method offers less room for the possiblity of data loss than does `DB_TXN_NOSYNC`.

  This behavior may be set for a Berkeley DB environment using the DB_ENV->set_flags() (page 260) method or for a single transaction using the DB_ENV->txn_begin() (page 544) method. Any value specified to this method overrides both of those settings.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->discard()

```
#include <db.h>

int
DB_TXN->discard(DB_TXN *tid, u_int32_t flags);
```

The `DB_TXN->discard()` method frees up all the per-process resources associated with the specified DB_TXN handle, neither committing nor aborting the transaction. This call may be used only after calls to DB_ENV->txn_recover() (page 542) when there are multiple global transaction managers recovering transactions in a single Berkeley DB environment. Any transactions returned by DB_ENV->txn_recover() (page 542) that are not handled by the current global transaction manager should be discarded using `DB_TXN->discard()`.

All open cursors in the transaction are closed and the first cursor close error, if any, is returned.

The `DB_TXN->discard()` method returns a non-zero error value on failure and 0 on success. The errors values that this method returns include the error values of `DBcursor->close()` and the following:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

After `DB_TXN->discard()` has been called, regardless of its return, the DB_TXN handle may not be accessed again.

Parameters

flags

The **flags** parameter is currently unused, and must be set to 0.

Errors

The `DB_TXN->discard()` method may fail and return one of the following non-zero errors:

EINVAL

If the transaction handle does not refer to a transaction that was recovered into a prepared but not yet completed state; or if an invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->get_name()

```
#include <db.h>

int
DB_TXN->get_name(DB_TXN *txn, const char **namep);
```

The `DB_TXN->get_name()` method returns the string associated with the transaction.

The `DB_TXN->get_name()` method may be called at any time during the life of the application.

The `DB_TXN->get_name()` method returns a non-zero error value on failure and 0 on success.

Parameters

namep

The `DB_TXN->get_name()` method returns a reference to the string associated with the transaction in **namep**.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->get_priority()

```
#include <db.h>

int
DB_TXN->get_priority(DB_TXN *tid, u_int32_t *priority);
```

The `DB_TXN->get_priority()` method gets the priority value of the specified transaction.

The `DB_TXN->get_priority()` method may be called at any time during the life of the transaction.

The `DB_TXN->get_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priority

Upon return, the **priority** parameter will point to a value between 0 and 2^32-1.

Errors

The `DB_TXN->get_priority()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->id()

```
#include <db.h>

u_int32_t
DB_TXN->id(DB_TXN *tid);
```

The `DB_TXN->id()` method returns the unique transaction id associated with the specified transaction. Locking calls made on behalf of this transaction should use the value returned from `DB_TXN->id()` as the locker parameter to the DB_ENV->lock_get() (page 313) or DB_ENV->lock_vec() (page 326) calls.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->prepare()

```
#include <db.h>

int
DB_TXN->prepare(DB_TXN *tid, u_int8_t gid[DB_XIDDATASIZE]);
```

The `DB_TXN->prepare()` method initiates the beginning of a two-phase commit.

In a distributed transaction environment, Berkeley DB can be used as a local transaction manager. In this case, the distributed transaction manager must send *prepare* messages to each local manager. The local manager must then issue a `DB_TXN->prepare()` and await its successful return before responding to the distributed transaction manager. Only after the distributed transaction manager receives successful responses from all of its *prepare* messages should it issue any *commit* messages.

In the case of nested transactions, preparing the parent causes all unresolved children of the parent transaction to be committed. Child transactions should never be explicitly prepared. Their fate will be resolved along with their parent's during global recovery.

All open cursors in the transaction are closed and the first cursor close error will be returned.

The `DB_TXN->prepare()` method returns a non-zero error value on failure and 0 on success. The errors that this method returns include the error values of `DBcursor->close()` and the following:

DB_LOCK_DEADLOCK

A transactional database environment operation was selected to resolve a deadlock.

DB_LOCK_NOTGRANTED

A Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

EINVAL

If the cursor is already closed; or if an invalid flag value or parameter was specified.

Parameters

gid

The **gid** parameter specifies the global transaction ID by which this transaction will be known. This global transaction ID will be returned in calls to DB_ENV->txn_recover() (page 542) telling the application which global transactions must be resolved.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->set_name()

```
#include <db.h>

int
DB_TXN->set_name(DB_TXN *txn, const char *name);
```

The `DB_TXN->set_name()` method associates the specified string with the transaction. The string is returned by DB_ENV->txn_stat() (page 549) and displayed by DB_ENV->txn_stat_print() (page 553).

If the database environment has been configured for logging and the Berkeley DB library was configured with --enable-diagnostic, a debugging log record is written including the transaction ID and the name.

The `DB_TXN->set_name()` method may be called at any time during the life of the application.

The `DB_TXN->set_name()` method returns a non-zero error value on failure and 0 on success.

Parameters

name

The **name** parameter is the string to associate with the transaction.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->set_priority()

```
#include <db.h>

int
DB_TXN->set_priority(DB_TXN *tid, u_int32_t priority);
```

The `DB_TXN->set_priority()` method sets the priority for the transaction. The deadlock detector will reject lock requests from lower priority transactions before those from higher priority transactions.

By default, all transactions are created with a priority of 100.

The `DB_TXN->set_priority()` method may be called at any time during the life of the transaction.

The `DB_TXN->set_priority()` method returns a non-zero error value on failure and 0 on success.

Parameters

priority

The **priority** parameter must be a value between 0 and 2^32-1.

Errors

The `DB_TXN->set_priority()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

DB_TXN->set_timeout()

```
#include <db.h>

u_int32_t
DB_TXN->set_timeout(DB_TXN *tid, db_timeout_t timeout, u_int32_t flags);
```

The `DB_TXN->set_timeout()` method sets timeout values for locks or transactions for the specified transaction.

Timeouts are checked whenever a thread of control blocks on a lock or when deadlock detection is performed. In the case of `DB_SET_LOCK_TIMEOUT`, the timeout is for any single lock request. In the case of `DB_SET_TXN_TIMEOUT`, the timeout is for the life of the transaction. As timeouts are only checked when the lock request first blocks or when deadlock detection is performed, the accuracy of the timeout depends on how often deadlock detection is performed.

Timeout values may be specified for the database environment as a whole. See DB_ENV->set_timeout() (page 282) and for more information.

The `DB_TXN->set_timeout()` method configures operations performed on the underlying transaction, not only operations performed using the specified DB_TXN handle.

The `DB_TXN->set_timeout()` method may be called at any time during the life of the application.

The `DB_TXN->set_timeout()` method returns a non-zero error value on failure and 0 on success.

Parameters

timeout

The **timeout** parameter is specified as an unsigned 32-bit number of microseconds, limiting the maximum timeout to roughly 71 minutes. A value of 0 disables timeouts for the transaction.

flags

The **flags** parameter must be set to one of the following values:

- `DB_SET_LOCK_TIMEOUT`

  Set the timeout value for locks in this transaction.

- `DB_SET_TXN_TIMEOUT`

  Set the timeout value for this transaction.

Errors

The `DB_TXN->set_timeout()` method may fail and return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DB_ENV, DB_TXN

See Also

Transaction Subsystem and Related Methods (page 534)

Appendix A. Berkeley DB Command Line Utilities

The following describes the command line utilities that are available for Berkeley DB.

Utilities

Utility	Description
db_archive	Archival utility
db_checkpoint	Transaction checkpoint utility
db_deadlock	Deadlock detection utility
db_dump	Database dump utility
db_hotbackup	Hot backup utility
db_load	Database load utility
db_log_verify	Log verification utility
db_printlog	Transaction log display utility
db_recover	Recovery utility
db_sql_codegen	SQL schema to Berkeley DB code in C
dbsql	Command line interface to libdb_sql
db_stat	Statistics utility
db_upgrade	Database upgrade utility
db_verify	Verification utility
sqlite3	Command line tool for wrapper library libsqlite3

db_archive

```
db_archive [-adlsVv] [-h home] [-P password]
```

The **db_archive** utility writes the pathnames of log files that are no longer in use (for example, no longer involved in active transactions), to the standard output, one pathname per line. These log files should be written to backup media to provide for recovery in the case of catastrophic failure (which also requires a snapshot of the database files), but they may then be deleted from the system to reclaim disk space.

The options are as follows:

- **-a**

  Write all pathnames as absolute pathnames, instead of relative to the database home directory.

- **-d**

  Remove log files that are no longer needed; no filenames are written. This automatic log file removal is likely to make catastrophic recovery impossible.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-l**

  Write out the pathnames of all the database log files, whether or not they are involved in active transactions.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-s**

  Write the pathnames of all the database files that need to be archived in order to recover the database from catastrophic failure. If any of the database files have not been accessed during the lifetime of the current log files, **db_archive** will not include them in this output.

  It is possible that some of the files to which the log refers have since been deleted from the system. In this case, **db_archive** will ignore them. When db_recover (page 591) is run, any files to which the log refers that are not present during recovery are assumed to have been deleted and will not be recovered.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

  Run in verbose mode.

Log cursor handles (returned by the DB_ENV->log_cursor() (page 340) method) may have open file descriptors for log files in the database environment. Also, the Berkeley DB interfaces to the database environment logging subsystem (for example, DB_ENV->log_put() (page 346) and DB_TXN->abort() (page 554) may allocate log cursors and have open file descriptors for log files as well. On operating systems where filesystem related system calls (for example, rename and unlink on Windows/NT) can fail if a process has an open file descriptor for the affected file, attempting to move or remove the log files listed by **db_archive** may fail. All Berkeley DB internal use of log cursors operates on active log files only and furthermore, is short-lived in nature. So, an application seeing such a failure should be restructured to close any open log cursors it may have, and otherwise to retry the operation until it succeeds. (Although the latter is not likely to be necessary; it is hard to imagine a reason to move or rename a log file in which transactions are being logged or aborted.)

The **db_archive** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_archive** should always be given the chance to detach from the environment and exit gracefully. To cause **db_archive** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The DB_ENV->log_archive() (page 337) method is the underlying method used by the **db_archive** utility. See the **db_archive** utility source code for an example of using `DB_ENV->log_archive()` in an IEEE/ANSI Std 1003.1 (POSIX) environment.

The **db_archive** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_checkpoint

```
db_checkpoint [-1Vv] [-h home]
    [-k kbytes] [-L file] [-P password] [-p min]
```

The **db_checkpoint** utility is a daemon process that monitors the database log, and periodically calls DB_ENV->txn_checkpoint() (page 547) to checkpoint it.

The options are as follows:

- **-1**

  Force a single checkpoint of the log (regardless of whether or not there has been activity since the last checkpoint), and then exit.

  When the **-1** flag is specified, the **db_checkpoint** utility will checkpoint the log even if unable to find an existing database environment. This functionality is useful when upgrading database environments from one version of Berkeley DB to another.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-k**

  Checkpoint the database at least as often as every **kbytes** of log file are written.

- **-L**

  Log the execution of the **db_checkpoint** utility to the specified file in the following format, where ### is the process ID, and the date is the time the utility was started.

  ```
  db_checkpoint: ### Wed Jun 15 01:23:45 EDT 1995
  ```

  This file will be removed if the **db_checkpoint** utility exits gracefully.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-p**

  Checkpoint the database at least every **min** minutes if there has been any activity since the last checkpoint.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

Write the time of each checkpoint attempt to the standard output.

At least one of the **-1**, **-k**, and **-p** options must be specified.

The **db_checkpoint** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_checkpoint** should always be given the chance to detach from the environment and exit gracefully. To cause **db_checkpoint** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_checkpoint** utility does not attempt to create the Berkeley DB shared memory regions if they do not already exist. The application that creates the region should be started first, and once the region is created, the **db_checkpoint** utility should be started.

The DB_ENV->txn_checkpoint() (page 547) method is the underlying method used by the **db_checkpoint** utility. See the **db_checkpoint** utility source code for an example of using `DB_ENV->txn_checkpoint()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The **db_checkpoint** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_deadlock

```
db_deadlock [-Vv]
    [-a e | m | n | o | W | w | y] [-h home] [-L file] [-t sec.usec]
```

The **db_deadlock** utility traverses the database environment lock region, and aborts a lock request each time it detects a deadlock or a lock request that has timed out. By default, in the case of a deadlock, a random lock request is chosen to be aborted.

This utility should be run as a background daemon, or the underlying Berkeley DB deadlock detection interfaces should be called in some other way, whenever there are multiple threads or processes accessing a database and at least one of them is modifying it.

The options are as follows:

- **-a**

  When a deadlock is detected, abort the locker:

  - **m**

    with the most locks

  - **n**

    with the fewest locks

  - **o**

    with the oldest locks

  - **W**

    with the most write locks

  - **w**

    with the fewest write locks

  - **y**

    with the youngest locks

  - **e**

    When lock or transaction timeouts have been specified, abort any lock request that has timed out. Note that this option does not perform the entire deadlock detection algorithm, but instead only checks for timeouts.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-L**

  Log the execution of the **db_deadlock** utility to the specified file in the following format, where ### is the process ID, and the date is the time the utility was started.

  ```
  db_deadlock: ### Wed Jun 15 01:23:45 EDT 1995
  ```

  This file will be removed if the **db_deadlock** utility exits gracefully.

- **-t**

  Check the database environment every **sec** seconds plus **usec** microseconds to see if a process has been forced to wait for a lock; if one has, review the database environment lock structures.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

  Run in verbose mode, generating messages each time the detector runs.

If the **-t** option is not specified, **db_deadlock** will run once and exit.

The **db_deadlock** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_deadlock** should always be given the chance to detach from the environment and exit gracefully. To cause **db_deadlock** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_deadlock** utility does not attempt to create the Berkeley DB shared memory regions if they do not already exist. The application which creates the region should be started first, and then, once the region is created, the **db_deadlock** utility should be started.

The DB_ENV->lock_detect() (page 311) method is the underlying method used by the **db_deadlock** utility. See the **db_deadlock** utility source code for an example of using `DB_ENV->lock_detect()` in a IEEE/ANSI Std 1003.1 (POSIX) environment.

The **db_deadlock** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_dump

```
db_dump [-klNpRrV] [-d ahr]
 [-f output] [-h home] [-P password] [-s database] file

db_dump [-kNpV] [-d ahr] [-f output] [-h home] -m database

db_dump185 [-p] [-f output] file
```

The **db_dump** utility reads the database file **file** and writes it to the standard output using a portable flat-text format understood by the db_load (page 582) utility. The **file** argument must be a file produced using the Berkeley DB library functions.

The **db_dump185** utility is similar to the **db_dump** utility, except that it reads databases in the format used by Berkeley DB versions 1.85 and 1.86.

The options are as follows:

- **-d**

  Dump the specified database in a format helpful for debugging the Berkeley DB library routines.

  - **a**

    Display all information.

  - **h**

    Display only page headers.

  - **r**

    Do not display the free-list or pages on the free list. This mode is used by the recovery tests.

  The output format of the -d option is not standard and may change, without notice, between releases of the Berkeley DB library.

- **-f**

  Write to the specified file instead of to the standard output.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-k**

  Dump record numbers from Queue and Recno databases as keys.

- **-l**

List the databases stored in the file.

- **-m**

  Specify a named in-memory database to dump. In this case the **file** argument must be omitted.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, will be ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-p**

  If characters in either the key or data items are printing characters (as defined by **isprint**(3)), use printing characters in **file** to represent them. This option permits users to use standard text editors and tools to modify the contents of databases.

  Note: different systems may have different notions about what characters are considered *printing characters*, and databases dumped in this manner may be less portable to external systems.

- **-R**

  Aggressively salvage data from a possibly corrupt file. The **-R** flag differs from the **-r** option in that it will return all possible data from the file at the risk of also returning already deleted or otherwise nonsensical items. Data dumped in this fashion will almost certainly have to be edited by hand or other means before the data is ready for reload into another database

- **-r**

  Salvage data from a possibly corrupt file. When used on a uncorrupted database, this option should return equivalent data to a normal dump, but most likely in a different order.

- **-s**

  Specify a single database to dump. If no database is specified, all databases in the database file are dumped.

- **-V**

  Write the library version number to the standard output, and exit.

Dumping and reloading Hash databases that use user-defined hash functions will result in new databases that use the default hash function. Although using the default hash function may not be optimal for the new database, it will continue to work correctly.

Dumping and reloading Btree databases that use user-defined prefix or comparison functions will result in new databases that use the default prefix and comparison functions. **In this case, it is quite likely that the database will be damaged beyond repair permitting neither record storage or retrieval.**

The only available workaround for either case is to modify the sources for the db_load (page 582) utility to load the database using the correct hash, prefix, and comparison functions.

The **db_dump185** utility may not be available on your system because it is not always built when the Berkeley DB libraries and utilities are installed. If you are unable to find it, see your system administrator for further information.

The **db_dump** and **db_dump185** utility output formats are documented in the Dump Output Formats section of the Berkeley DB Reference Guide.

The **db_dump** utility may be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_dump** should always be given the chance to detach from the environment and exit gracefully. To cause **db_dump** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

Even when using a Berkeley DB database environment, the db_dump utility does not use any kind of database locking if it is invoked with the **-d**, **-R**, or **-r** arguments. If used with one of these arguments, the **db_dump** utility may only be safely run on databases that are not being modified by any other process; otherwise, the output may be corrupt.

The **db_dump** utility exits 0 on success, and >0 if an error occurs.

The **db_dump185** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_hotbackup

```
db_hotbackup [-cDuVv] [-d data_dir ...] [-h home]
        [-l log_dir] [-P password] -b backup_dir
```

The **db_hotbackup** utility creates "hot backup" or "hot failover" snapshots of Berkeley DB database environments.

The **db_hotbackup** utility performs the following steps:

1. If the **-c** option is specified, checkpoint the source home database environment, and remove any unnecessary log files.

2. If the target directory for the backup does not exist, it is created with mode read-write-execute for the owner.

   If the target directory for the backup does exist and the **-u** option was specified, all log files in the target directory are removed; if the **-u** option was not specified, all files in the target directory are removed.

3. If the **-u** option was not specified, copy application-specific files found in the database environment home directory, and any directories specified using the **-d** option, into the target directory for the backup.

4. Copy all log files found in the directory specified by the **-l** option (or in the database environment home directory, if no **-l** option was specified), into the target directory for the backup.

5. Perform catastrophic recovery in the target directory for the backup.

6. Remove any unnecessary log files from the target directory for the backup.

The **db_hotbackup** utility does not resolve pending transactions that are in the prepared state. Applications that use DB_TXN->prepare() (page 562) must specify DB_RECOVER_FATAL when opening the environment, and run DB_ENV->txn_recover() (page 542) to resolve any pending transactions, when failing over to the backup.

The options are as follows:

- **-b**

  Specify the target directory for the backup.

- **-c**

  Before performing the backup, checkpoint the source database environment and remove any log files that are no longer required in that environment. **To avoid making catastrophic recovery impossible, log file removal must be integrated with log file archival.**

- **-D**

Use the data and log directories listed in a DB_CONFIG configuration file in the source directory. This option has four effects:

- The specified data and log directories will be created relative to the target directory, with mode read-write-execute owner, if they do not already exist.
- In step #3 above, all files in any source data directories specified in the DB_CONFIG file will be copied to the target data directories.
- In step #4 above, log files will be copied from any log directory specified in the DB_CONFIG file, instead of from the default locations.
- The DB_CONFIG configuration file will be copied from the source directory to the target directory, and subsequently used for configuration if recovery is run in the target directory.

Care should be taken with the **-D** option where data and log directories are named relative to the source directory but are not subdirectories (that is, the name includes the element "..") Specifically, the constructed target directory names must be meaningful and distinct from the source directory names, otherwise running recovery in the target directory might corrupt the source data files.

It is an error to use absolute pathnames for data or log directories in this mode, as the DB_CONFIG configuration file copied into the target directory would then point at the source directories and running recovery would corrupt the source data files.

- **-d**

  Specify one or more directories that contain data files to be copied to the target directory.

  As all database files are copied into a single target directory, files named the same, stored in different source directories, would overwrite each other when copied to the target directory.

  Please note the database environment recovery log references database files as they are named by the application program. **If the application uses absolute or relative pathnames to name database files, (rather than filenames and the DB_ENV->set_data_dir() (page 243) method or the DB_CONFIG configuration file to specify filenames), running recovery in the target directory may not properly find the copies of the files or might even find the source files, potentially resulting in corruption.**

- **-h**

  Specify the source directory for the backup. That is, the database environment home directory.

- **-l**

  Specify a source directory that contains log files; if none is specified, the database environment home directory will be searched for log files.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-u**

  Update a pre-existing hot backup snapshot by copying in new log files. If the **-u** option is specified, no databases will be copied into the target directory.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

  Run in verbose mode, listing operations as they are done.

The **db_hotbackup** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_hotbackup** should always be given the chance to detach from the environment and exit gracefully. To cause **db_hotbackup** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_hotbackup** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_load

```
db_load [-nTV] [-c name=value] [-f file]
    [-h home] [-P password] [-t btree | hash | queue | recno] file

db_load [-r lsn | fileid] [-h home] [-P password] file
```

The **db_load** utility reads from the standard input and loads it into the database **file**. The database **file** is created if it does not already exist.

The input to **db_load** must be in the output format specified by the db_dump (page 576) utility or as specified by the **-T** option below.

The options are as follows:

- **-c**

  Specify configuration options ignoring any value they may have based on the input. The command-line format is **name=value**. See the Supported Keywords section below for a list of keywords supported by the **-c** option.

- **-f**

  Read from the specified **input** file instead of from the standard input.

- **-h**

  Specify a home directory for the database environment.

  If a home directory is specified, the database environment is opened using the DB_INIT_LOCK, DB_INIT_LOG, DB_INIT_MPOOL, DB_INIT_TXN, and DB_USE_ENVIRON flags to DB_ENV->open() (page 231) (This means that **db_load** can be used to load data into databases while they are in use by other processes.) If the DB_ENV->open() (page 231) call fails, or if no home directory is specified, the database is still updated, but the environment is ignored; for example, no locking is done.

- **-n**

  Do not overwrite existing keys in the database when loading into an already existing database. If a key/data pair cannot be loaded into the database for this reason, a warning message is displayed on the standard error output, and the key/data pair are skipped.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-r**

  Reset the database's file ID or log sequence numbers (LSNs).

All database pages in transactional environments contain references to the environment's log records. In order to copy a database into a different database environment, database page references to the old environment's log records must be reset, otherwise data corruption can occur when the database is modified in the new environment. The **-r lsn** option resets a database's log sequence numbers.

All databases contain an ID string used to identify the database in the database environment cache. If a database is copied, and used in the same environment as another file with the same ID string, corruption can occur. The **-r fileid** option resets a database's file ID to a new value.

In both cases, the physical file specified by the file argument is modified in-place.

- **-T**

  The **-T** option allows non-Berkeley DB applications to easily load text files into databases.

  If the database to be created is of type Btree or Hash, or the keyword **keys** is specified as set, the input must be paired lines of text, where the first line of the pair is the key item, and the second line of the pair is its corresponding data item. If the database to be created is of type Queue or Recno and the keyword **keys** is not set, the input must be lines of text, where each line is a new data item for the database.

  A simple escape mechanism, where newline and backslash (\) characters are special, is applied to the text input. Newline characters are interpreted as record separators. Backslash characters in the text will be interpreted in one of two ways: If the backslash character precedes another backslash character, the pair will be interpreted as a literal backslash. If the backslash character precedes any other character, the two characters following the backslash will be interpreted as a hexadecimal specification of a single character; for example, \0a is a newline character in the ASCII character set.

  For this reason, any backslash or newline characters that naturally occur in the text input must be escaped to avoid misinterpretation by **db_load**.

  If the **-T** option is specified, the underlying access method type must be specified using the **-t** option.

- **-t**

  Specify the underlying access method. If no **-t** option is specified, the database will be loaded into a database of the same type as was dumped; for example, a Hash database will be created if a Hash database was dumped.

  Btree and Hash databases may be converted from one to the other. Queue and Recno databases may be converted from one to the other. If the **-k** option was specified on the call to db_dump (page 576) then Queue and Recno databases may be converted to Btree or Hash, with the key being the integer record number.

- **-V**

  Write the library version number to the standard output, and exit.

The **db_load** utility may be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_load** should always be given the chance to detach from the environment and exit gracefully. To cause **db_load** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_load** utility exits 0 on success, 1 if one or more key/data pairs were not loaded into the database because the key already existed, and >1 if an error occurs.

Examples

The **db_load** utility can be used to load text files into databases. For example, the following command loads the standard UNIX */etc/passwd* file into a database, with the login name as the key item and the entire password entry as the data item:

```
awk -F: '{print $1; print $0}' < /etc/passwd |
        sed 's/\\/\\\\/g' | db_load -T -t hash passwd.db
```

Note that backslash characters naturally occurring in the text are escaped to avoid interpretation as escape characters by **db_load**.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

Supported Keywords

The following keywords are supported for the **-c** command-line option to the **db_load** utility. See the DB_ENV->open() (page 231) method for further discussion of these keywords and what values should be specified.

The parenthetical listing specifies how the value part of the **name=value** pair is interpreted. Items listed as (boolean) expect value to be **1** (set) or **0** (unset). Items listed as (number) convert value to a number. Items listed as (string) use the string value without modification.

- **bt_minkey (number)**

  The minimum number of keys per page.

- **chksum (boolean)**

  Enable page checksums.

- **database (string)**

  The database to load.

- **db_lorder (number)**

The byte order for integers in the stored database metadata. For big endian systems, the order should be 4,321 while for little endian systems is should be 1,234.

- **db_pagesize (number)**

  The size of database pages, in bytes.

- **duplicates (boolean)**

  The value of the DB_DUP flag.

- **dupsort (boolean)**

  The value of the DB_DUPSORT flag.

- **extentsize (number)**

  The size of database extents, in pages, for Queue databases configured to use extents.

- **h_ffactor (number)**

  The density within the Hash database.

- **h_nelem (number)**

  The size of the Hash database.

- **keys (boolean)**

  Specify whether keys are present for Queue or Recno databases.

- **re_len (number)**

  Specify the length for fixed-length records. This number represents different things, depending on the access method the database is using. See the DB->set_re_len() (page 126) method for details on what this number represents.

- **re_pad (string)**

  Specify the fixed-length record pad character.

- **recnum (boolean)**

  The value of the DB_RECNUM flag.

- **renumber (boolean)**

  The value of the DB_RENUMBER flag.

- **subdatabase (string)**

  The subdatabase to load.

db_log_verify

```
db_log_verify [-cNvV] [-h home to verify] [-H temporary home]
[-P password] [-C cache size]
[-b start lsn] [-e end lsn] [-s start time] [-z end time]
[-d database file name] [-D database name]
```

The **db_log_verify** utility verifies the log files of a specific database environment. This utility verifies a specific range of log records, or changed log records of a specific database.

The options are as follows:

- **-C**

  Specify the cache size (in megabytes) of the temporary database environment internally used during the log verification.

- **-b**

  Specify the starting log record (by lsn) to verify.

- **-c**

  Specify whether to continue the verification after an error is detected. If not specified, the verification stops when the first error is detected.

- **-D**

  Specify a database name. Only log records related to this database are verified.

- **-d**

  Specify a database file name. Only log records related this database file are verified.

- **-e**

  Specify the ending log record by lsn.

- **-h**

  Specify a home directory of the database environment whose log is to be verified.

- **-H**

  Specify a home directory for this utility to create a temporarily database environment to store runtime data during the verification.

  It is an error to specify the same directory as the -h option. If this directory is not specified, all temporary databases created during the verification will be in-memory, which is not a problem if the log files to verify are not huge.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, are ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-s**

  Specify the starting log record by time. The time range specified is not precise because the lsn of the most recent time point is used as the starting lsn.

- **-V**

  Write the library version number to the standard output and exit.

- **-v**

  Enable verbose mode to display verbose output during the verification process.

- **-z**

  Specify the ending log record by time. The time range specified is not precise because the lsn of the most recent time point is used as the ending lsn.

To specify a range of log records, you must provide either an lsn range or a time range. You can neither specify both nor specify an lsn and a time as a range.

If the log footprint is over several megabytes, specify a home directory and a big cache size for log verification internal use. Else, the process' private memory may be exhausted before the verification completes.

The **db_log_verify** utility does not perform the locking function, even in Berkeley DB environments that are configured with a locking subsystem. All errors are written to stderr, and all normal and verbose messages are written to stdout.

The **db_log_verify** utility can be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**). To avoid environment corruption when using a Berkeley DB environment, **db_log_verify** must be given the chance to detach from the environment and exit gracefully. For the **db_log_verify** utility to release all environment resources and exit, send an interrupt signal (SIGINT) to it.

The **db_log_verify** utility returns a non-zero error value on failure and 0 on success.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_printlog

```
db_printlog [-NrV] [-b start-LSN] [-e stop-LSN] [-h home] [-P password]
```

The **db_printlog** utility is a debugging utility that dumps Berkeley DB log files in a human-readable format.

The options are as follows:

- **-b**

  Display log records starting at log sequence number (LSN) **start-LSN**; **start-LSN** is specified as a file number, followed by a slash (/) character, followed by an offset number, with no intervening whitespace.

- **-e**

  Stop displaying log records at log sequence number (LSN) **stop-LSN**; **stop-LSN** is specified as a file number, followed by a slash (/) character, followed by an offset number, with no intervening whitespace.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, will be ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-r**

  Read the log files in reverse order.

- **-V**

  Write the library version number to the standard output, and exit.

For more information on the **db_printlog** output and using it to debug applications, see Reviewing Berkeley DB log files.

The **db_printlog** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a

Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_printlog** should always be given the chance to detach from the environment and exit gracefully. To cause **db_printlog** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_printlog** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_recover

```
db_recover [-cefVv] [-h home] [-P password] [-t [[CC]YY]MMDDhhmm[.SS]]]
```

The **db_recover** utility must be run after an unexpected application, Berkeley DB, or system failure to restore the database to a consistent state. All committed transactions are guaranteed to appear after **db_recover** has run, and all uncommitted transactions will be completely undone.

Note that this utility performs the same action as if the environment is opened with the DB_RECOVER flag. If `DB_RECOVER` is specified on environment open, then use of this utility is not necessary.

The options are as follows:

- **-c**

  Perform catastrophic recovery instead of normal recovery.

- **-e**

  Retain the environment after running recovery. This option will rarely be used unless a DB_CONFIG file is present in the home directory. If a DB_CONFIG file is not present, then the regions will be created with default parameter values.

- **-f**

  Display a message on the standard output showing the percent of recovery completed.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-t**

  Recover to the time specified rather than to the most current possible date. The timestamp argument should be in the form `[[CC]YY]MMDDhhmm[.SS]` where each pair of letters represents the following:

  - **CC**

    The first two digits of the year (the century).

  - **YY**

The second two digits of the year. If "YY" is specified, but "CC" is not, a value for "YY" between 69 and 99 results in a "CC" value of 19. Otherwise, a "CC" value of 20 is used.

- **MM**

  The month of the year, from 1 to 12.

- **DD**

  The day of the month, from 1 to 31.

- **hh**

  The hour of the day, from 0 to 23.

- **mm**

  The minute of the hour, from 0 to 59.

- **SS**

  The second of the minute, from 0 to 61.

If the "CC" and "YY" letter pairs are not specified, the values default to the current year. If the "SS" letter pair is not specified, the value defaults to 0.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

  Run in verbose mode.

In the case of catastrophic recovery, an archival copy — or *snapshot* — of all database files must be restored along with all of the log files written since the database file snapshot was made. (If disk space is a problem, log files may be referenced by symbolic links). For further information on creating a database snapshot, see Archival Procedures. For further information on performing recovery, see Recovery Procedures.

If the failure was not catastrophic, the files present on the system at the time of failure are sufficient to perform recovery.

If log files are missing, **db_recover** will identify the missing log file(s) and fail, in which case the missing log files need to be restored and recovery performed again.

The **db_recover** utility uses a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_recover** should always be given the chance to detach from the environment and exit gracefully. To cause **db_recover** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_recover** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_sql_codegen

```
db_sql_codegen [-i <ddl input file>] [-o <output C code file>]
    [-h <output header file>] [-t <test output file>]
```

Db_sql_codegen is a utility program that translates a schema description written in a SQL Data Definition Language dialect into C code that implements the schema using Berkeley DB. It is intended to provide a quick and easy means of getting started with Berkeley DB for users who are already conversant with SQL. It also introduces a convenient way to express a Berkeley DB schema in a format that is both external to the program that uses it and compatible with relational databases.

The **db_sql_codegen** command reads DDL from an input stream, and writes C code to an output stream. With no command line options, it will read from stdin and write to stdout. A more common usage mode would be to supply the DDL in a named input file (-i option). With only the -i option, **db_sql_codegen** will produce two files: a C-language source code (.c) file and a C-language header (.h) file, with names that are derived from the name of the input file. You can also control the names of these output files with the -o and -h options. The -x option causes the generated code to be transaction-aware. Finally, the -t option will produce a simple application that invokes the generated function API. This is a C-language source file that includes a main function, and serves the dual purposes of providing a simple test for the generated C code, and of being an example of how to use the generated API.

The options are as follows:

- **-i**<ddl input file>

  Names the input file containing SQL DDL.

- **-o** <output C code file>

  Names the output C-language source code file.

- **-h** <output header file>

  Names the output C-language header file.

- **-t** <test output file>

  Names the output C-langage test file.

- **-x**

  Sets the default transaction mode to TRANSACTIONAL.

The **db_sql_codegen** utility exits 0 on success, and >0 if an error occurs.

Note that the **db_sql_codegen** utility is built only when --enable-sql_codegen option is passed as an argument when you are configuring Berkeley DB. For more information, see "Configuring Berkeley DB"

Input Syntax

The input file can contain the following SQL DDL statements.

- **CREATE DATABASE**

  The DDL must contain a CREATE DATABASE statement. The syntax is simply

  ```
  CREATE DATABASE name;
  ```

  . The name given here is used as the name of the Berkeley DB environment in which the Berkeley DB databases are created.

- **CREATE TABLE**

  Each CREATE TABLE statement produces functions to create and delete a primary Berkeley DB database. Also produced are functions to perform record insertion, retrieval and deletion on this database.

  CREATE TABLE establishes the field set of records that can be stored in the Berkeley DB database. Every CREATE TABLE statement must identify a primary key to be used as the lookup key in the Berkeley DB database.

  Here is an example to illustrate the syntax of CREATE TABLE that is accepted by **db_sql_codegen**:

  ```
  CREATE TABLE person (person_id INTEGER PRIMARY KEY,
                       name VARCHAR(64),
                       age INTEGER);
  ```

  This results in the creation of functions to manage a database in which every record is an instance of the following C language data structure:

  ```
  typedef struct _person_data {
      int person_id;
      char name[PERSON_DATA_NAME_LENGTH];
      int age;
  } person_data;
  ```

- **CREATE INDEX** You can create secondary Berkeley DB databases to be used as indexes into a primary database. For example, to make an index on the "name" field of the "person" table mentioned above, the SQL DDL would be:

  ```
  CREATE INDEX name_index ON person(name);
  ```

  This causes **db_sql_codegen** to emit functions to manage creation and deletion of a secondary database called "name_index," which is associated with the "person" database and is set up to perform lookups on the "name" field.

Hint Comments

The SQL DDL input may contain comments. Two types of comments are recognized. C-style comments begin with "/*" and end with "*/". These comments may extend over multiple lines.

Single line comments begin with "--" and run to the end of the line.

If the first character of a comment is "+" then the comment is interpreted as a "hint comment." Hint comments can be used to configure Berkeley DB features that cannot be represented in SQL DDL.

Hint comments are comma-separated lists of property assignments of the form "property=value." Hint comments apply to the SQL DDL statement that immediately precedes their appearance in the input. For example:

```
CREATE DATABASE peopledb; /*+ CACHESIZE = 16m */
```

This causes the generated environment creation function to set the cache size to sixteen megabytes.

In addition to the CACHESIZE example above, two other hint comment keywords are recognized: DBTYPE and MODE.

After a CREATE TABLE or CREATE INDEX statement, you may set the database type by assigning the DBTYPE property in a hint comment. Possible values for DBTYPE are BTREE and HASH.

After a CREATE DATABASE or CREATE TABLE statement, you may tell **db_sql_codegen** whether to generate transaction-aware code by assigning the MODE property in a hint comment. The possible values for MODE are TRANSACTIONAL and NONTRANSACTIONAL. By default, generated code is not transaction-aware. If MODE=TRANSACTIONAL appears on a CREATE DATABASE statement, then the default for every CREATE TABLE statement becomes TRANSACTIONAL. Individual CREATE TABLE statements may have MODE=TRANSACTIONAL or MODE=NONTRANSACTIONAL, to control whether the code generated for accessing and updating the associated Berkeley DB database is transaction aware.

Transactions

By default, the code generated by **db_sql_codegen** is not transaction-aware. This means that the generated API for reading and updating BDB databases operates in nontransactional mode. When transactional mode is enabled, either through the command-line option **-x** or by the inclusion of MODE-setting hint comments in the DDL source, the generated data access functions take an extra argument which is a pointer to DB_TXN. To use transactions, application code must acquire a DB_TXN from a call to DB_ENV->txn_begin, and supply a pointer to this object when invoking the db_sql_codegen-generated functions that requre such an argument.

Transaction-aware APIs that were generated by db_sql_codegen can be used in nontransactional mode by passing NULL for the DB_TXN pointer arguments.

For more information about using BDB transactions, please consult the documentation for Transaction Subsystem and Related Methods (page 534) .

Type Mapping

db_sql_codegen must map the schema expressed as SQL types into C language types. It implements the following mappings:

```
BIN      char[]
VARBIN      char[]
```

```
CHAR        char[]
VARCHAR     char[]
VARCHAR2    char[]
BIT         char
TINYINT     char
SMALLINT    short
INTEGER     int
INT         int
BIGINT      long
REAL        float
DOUBLE      double
FLOAT       double
DECIMAL     double
NUMERIC     double
NUMBER(p,s) int, long, float, or double
```

While BIN/VARBIN and CHAR/VARCHAR are both represented as char arrays, the latter are treated as null-terminated C strings, while the former are treated as binary data.

The Oracle type NUMBER is mapped to different C types, depending on its precision and scale values. If scale is 0, then it is mapped to an integer type (long if precision is greater than 9). Otherwise it is mapped to a floating point type (float if precision is less than 7, otherwise double).

Output

Depending on the options given on the command line, **db_sql_codegen** can produce three separate files: a .c file containing function definitions that implement the generated API; a .h file containing constants, data structures and prototypes of the generated functions; and a second .c file that contains a sample program that invokes the generated API. The latter program is usually referred to as a smoke test.

Given the following sample input in a file named "people.sql":

```
CREATE DATABASE peopledb;
CREATE TABLE person (person_id INTEGER PRIMARY KEY,
                  name VARCHAR(64),
                  age INTEGER);
CREATE INDEX name_index ON person(name);
```

The command

```
db_sql_codegen -i people.sql -t test_people.c
```

Will produce files named people.h, people.c, and test_people.c.

The file people.h will contain the information needed to use the generated API. Among other things, an examination of the generated .h file will reveal:

```
#define PERSON_DATA_NAME_LENGTH 63
```

This is just a constant for the length of the string mapped from the VARCHAR field.

```
typedef struct _person_data {
   int   person_id;
   char  name[PERSON_DATA_NAME_LENGTH];
   int   age;
} person_data;
```

This is the data structure that represents the record type that is stored in the person database. There's that constant being used.

```
int create_peopledb_env(DB_ENV **envpp);
int create_person_database(DB_ENV *envp, DB **dbpp);
int create_name_index_secondary(DB_ENV *envp, DB *primary_dbp,
                            DB **secondary_dbpp);
```

These functions must be invoked to initialize the Berkeley DB environment. However, see the next bit:

```
extern DB_ENV * peopledb_envp;
extern DB *person_dbp;
extern DB *name_index_dbp;

int initialize_peopledb_environment();
```

For convenience, **db_sql_codegen** provides global variables for the environment and database, and a single initialization function that sets up the environment for you. You may choose to use the globals and the single initialization function, or you may declare your own DB_ENV and DB pointers, and invoke the individual create_* functions yourself.

The word "create" in these function names might be confusing. It means "create the environment/database if it doesn't already exist; otherwise open it."

All of the functions in the generated API return Berkeley DB error codes. If the return value is non-zero, there was an error of some kind, and an explanatory message should have been printed on stderr.

```
int person_insert_struct(DB *dbp, person_data *personp);
int person_insert_fields(DB * dbp,
        int person_id,
        char *name,
        int age);
```

These are the functions that you'd use to store a record in the database. The first form takes a pointer to the data structure that represents this record. The second form takes each field as a separate argument.

If two records with the same primary key value are stored, the first one is lost.

```
int get_person_data(DB *dbp, int person_key, person_data *data);
```

This function retrieves a record from the database. It seeks the record with the supplied key, and populates the supplied structure with the contents of the record. If no matching record is found, the function returns DB_NOTFOUND.

```
int delete_person_key(DB *dbp, int person_key);
```

This function removes the record matching the given key.

```
typedef void (*person_iteration_callback)(void *user_data,
                                         person_data *personp);

int person_full_iteration(DB *dbp,
    person_iteration_callback user_func,
    void *user_data);
```

This function performs a complete iteration over every record in the person table. The user must provide a callback function which is invoked once for every record found. The user's callback function must match the prototype provided in the typedef "person_iteration_callback." In the callback, the "user_data" argument is passed unchanged from the "user_data" argument given to person_full_iteration. This is provided so that the caller of person_full_iteration can communicate some context information to the callback function. The "personp" argument to the callback is a pointer to the record that was retrieved from the database. Personp points to data that is valid only for the duration of the callback invocation.

```
int name_index_query_iteration(DB *secondary_dbp,
    char *name_index_key,
    person_iteration_callback user_func,
    void *user_data);
```

This function performs lookups through the secondary index database. Because duplicate keys are allowed in secondary indexes, this query might return multiple instances. This function takes as an argument a pointer to a user-written callback function, which must match the function prototype typedef mentioned above (person_iteration_callback). The callback is invoked once for each record that matches the secondary key.

Test output

The test output file is useful as an example of how to invoke the generated API. It will contain calls to the functions mentioned above, to store a single record and retrieve it by primary key and through the secondary index.

To compile the test, you would issue a command such as

```
 cc -I$BDB_INSTALL/include -L$BDB_INSTALL/lib -o test_people people.c \
     test_people.c -ldb-4.8
```

This will produce the executable file test_people, which can be run to exercise the generated API. The program generated from people.sql will create a database environment in a directory named "peopledb." This directory must be created before the program is run.

dbsql

```
dbsql [OPTIONS] FILENAME SQL
```

dbsql is a command line tool that provides access to the Berkeley DB SQL interface. To build this tool, run the configure script with the `--enable-sql` option when you are building the Berkeley DB SQL interface. For more information on building this tool, see "Building for UNIX/POSIX".

FILENAME is the name of a Berkeley DB database file created with the SQL interface. A new database is created if the file does not exist. The options are as follows:

- **-init filename**

  Reads/processes named file.

- **-echo**

  Prints commands before execution.

- **-[no]header**

  Turns headers on or off.

- **-bail**

  Stops after hitting an error.

- **-interactive**

  Forces interactive I/O.

- **-batch**

  Forces batch I/O.

- **-column**

  Sets output mode to column.

- **-csv**

  Sets output mode to csv.

- **-html**

  Sets output mode to HTML.

- **-line**

  Sets output mode to line.

- **-list**

Sets output mode to list.

- **-separator 'x'**

  Sets output field separator (|).

- **-nullvalue 'text'**

  Sets text string for NULL values.

- **-version**

  Shows SQLite version.

The dbsql executable provides the same interface as the sqlite3 executable that is part of SQLite. For more information on how to use dbsql see the SQLite Documentation page.

db_stat

```
db_stat -d file [-fN] [-h home] [-P password] [-s database]

db_stat [-cEelmNrtVxZ] [-C Aclop] [-h home] [-L A] [-M Ah] [-R A]
        [-P password]
```

The **db_stat** utility displays statistics for Berkeley DB environments.

The options are as follows:

- **-C**

  Display detailed information about the locking subsystem.

  - **A**

    Display all information.

  - **c**

    Display lock conflict matrix.

  - **l**

    Display lockers within hash chains.

  - **o**

    Display lock objects within hash chains.

  - **p**

    Display locking subsystem parameters.

- **-c**

  Display locking subsystem statistics, as described in the DB_ENV->lock_stat() (page 319) method.

- **-d**

  Display database statistics for the specified file, as described in the DB->stat() (page 130) method.

  If the database contains multiple databases and the **-s** flag is not specified, the statistics are for the internal database that describes the other databases the file contains, and not for the file as a whole.

- **-E**

  Display detailed information about the database environment.

- **-e**

  Display information about the database environment, including all configured subsystems of the database environment.

- **-f**

  Display only those database statistics that can be acquired without traversing the database.

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-l**

  Display logging subsystem statistics, as described in the DB_ENV->log_stat() (page 351) method.

- **-L**

  Display all logging subsystem statistics.

  - **A**

    Display all information.

- **-M**

  Display detailed information about the cache.

  - **A**

    Display all information.

  - **h**

    Display buffers within hash chains.

- **-m**

  Display cache statistics, as described in the DB_ENV->memp_stat() (page 383) method.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, will be ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-R**

  Display detailed information about the replication subsystem.

  - **A**

    Display all information.

- **-r**

  Display replication statistics, as described in in the DB_ENV->rep_stat() (page 486) method.

- **-s**

  Display statistics for the specified database contained in the file specified with the **-d** flag.

- **-t**

  Display transaction subsystem statistics, as described in the DB_ENV->txn_stat() (page 549) method.

- **-V**

  Write the library version number to the standard output, and exit.

- **-x**

  Display mutex subsystem statistics, as described in the DB_ENV->mutex_stat() (page 447) method.

- **-Z**

  Reset the statistics after reporting them; valid only with the **-C**, **-c**, **-E**, **-e**, **-L**, **-l**, **-M**, **-m**, **-R**, **-r**, and **-t** options.

Values normally displayed in quantities of bytes are displayed as a combination of gigabytes (GB), megabytes (MB), kilobytes (KB), and bytes (B). Otherwise, values smaller than 10 million are displayed without any special notation, and values larger than 10 million are displayed as a number followed by "M".

The **db_stat** utility may be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_stat** should always be given the chance to detach from the environment and exit gracefully. To cause **db_stat** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_stat** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_upgrade

```
db_upgrade [-NsVv] [-h home] [-P password] file ...
```

The **db_upgrade** utility upgrades the Berkeley DB version of one or more files and the databases they contain to the current release version.

The options are as follows:

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, will be ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-s**

  This flag is only meaningful when upgrading databases from releases before the Berkeley DB 3.1 release.

  As part of the upgrade from the Berkeley DB 3.0 release to the 3.1 release, the on-disk format of duplicate data items changed. To correctly upgrade the format requires that applications specify whether duplicate data items in the database are sorted or not. Specifying the **-s** flag means that the duplicates are sorted; otherwise, they are assumed to be unsorted. Incorrectly specifying the value of this flag may lead to database corruption.

  Because the **db_upgrade** utility upgrades a physical file (including all the databases it contains), it is not possible to use **db_upgrade** to upgrade files where some of the databases it includes have sorted duplicate data items, and some of the databases it includes have unsorted duplicate data items. If the file does not have more than a single database, if the databases do not support duplicate data items, or if all the databases that support duplicate data items support the same style of duplicates (either sorted or unsorted), **db_upgrade** will work correctly as long as the **-s** flag is correctly specified. Otherwise, the file cannot be upgraded using **db_upgrade**, and must be upgraded manually using the db_dump (page 576) and db_load (page 582) utilities.

- **-V**

  Write the library version number to the standard output, and exit.

- **-v**

  Run in verbose mode, displaying a message for each successful upgrade.

It is important to realize that Berkeley DB database upgrades are done in place, and so are potentially destructive. This means that if the system crashes during the upgrade procedure, or if the upgrade procedure runs out of disk space, the databases may be left in an inconsistent and unrecoverable state. See Upgrading databases for more information.

The **db_upgrade** utility may be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_upgrade** should always be given the chance to detach from the environment and exit gracefully. To cause **db_upgrade** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_upgrade** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

db_verify

```
db_verify [-NoqV] [-h home] [-P password] file ...
```

The **db_verify** utility verifies the structure of one or more files and the databases they contain.

The options are as follows:

- **-h**

  Specify a home directory for the database environment; by default, the current working directory is used.

- **-o**

  Skip the database checks for btree and duplicate sort order and for hashing.

  If the file being verified contains databases with non-default comparison or hashing configurations, calling the **db_verify** utility without the **-o** flag will usually return failure. The **-o** flag causes **db_verify** to ignore database sort or hash ordering and allows **db_verify** to be used on these files. To fully verify these files, verify them explicitly using the DB->verify() (page 144) method, after configuring the correct comparison or hashing functions.

- **-N**

  Do not acquire shared region mutexes while running. Other problems, such as potentially fatal errors in Berkeley DB, will be ignored as well. This option is intended only for debugging errors, and should not be used under any other circumstances.

- **-P**

  Specify an environment password. Although Berkeley DB utilities overwrite password strings as soon as possible, be aware there may be a window of vulnerability on systems where unprivileged users can see command-line arguments or where utilities are not able to overwrite the memory containing the command-line arguments.

- **-q**

  Suppress the printing of any error descriptions, simply exit success or failure.

- **-V**

  Write the library version number to the standard output, and exit.

The db_verify utility does not perform any locking, even in Berkeley DB environments that are configured with a locking subsystem. As such, it should only be used on files that are not being modified by another thread of control.

The **db_verify** utility may be used with a Berkeley DB environment (as described for the **-h** option, the environment variable **DB_HOME**, or because the utility was run in a directory containing a Berkeley DB environment). In order to avoid environment corruption when using a Berkeley DB environment, **db_verify** should always be given the chance to detach from the environment and exit gracefully. To cause **db_verify** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_verify** utility exits 0 on success, and >0 if an error occurs.

Environment Variables

DB_HOME

If the **-h** option is not specified and the environment variable DB_HOME is set, it is used as the path of the database home, as described in the DB_ENV->open() (page 231) method.

sqlite3

Sqlite3 is a command line tool that enables you to manually enter and execute SQL commands. It is identical to the dbsql executable but named so that existing scripts for SQLite can easily work with Berkeley DB. To build this tool, run the configure script with the `--enable-sql_compat` option when you are building the Berkeley DB SQL interface.

For more information on building this tool, see the "Building for UNIX/POSIX"

For more information on how to use Sqlite3 see the SQLite Documentation page.

Appendix B. Historic Interfaces

This appendix describe the usage of several historic interfaces that previous users of Berkeley DB may have made use of.

Historic Interfaces

Historic Interfaces	Description
dbm/ndbm	Compatibility for applications written to the historic dbm or hdbm interfaces
hsearch	Compatibility for applications written to the historic hsearch interface

dbm/ndbm

```
#define DB_DBM_HSEARCH    1
#include <db.h>

typedef struct {
 char *dptr;
 int dsize;
} datum;
```

Dbm Functions

```
int
dbminit(char *file);

int
dbmclose();

datum
fetch(datum key);

int
store(datum key, datum content);

int
delete(datum key);

datum
firstkey(void);

datum
nextkey(datum key);
```

Ndbm Functions

```
DBM *
dbm_open(char *file, int flags, int mode);

void
dbm_close(DBM *db);

datum
dbm_fetch(DBM *db, datum key);

int
dbm_store(DBM *db, datum key, datum content, int flags);

int
dbm_delete(DBM *db, datum key);

datum
```

```
dbm_firstkey(DBM *db);

datum
dbm_nextkey(DBM *db);

int
dbm_error(DBM *db);

int
dbm_clearerr(DBM *db);
```

The dbm functions are intended to provide high-performance implementations and source code compatibility for applications written to historic interfaces. They are not recommended for any other purpose. The historic dbm database format **is not supported**, and databases previously built using the real dbm libraries cannot be read by the Berkeley DB functions.

To compile dbm applications, replace the application's **#include** of the dbm or ndbm include file (for example, **#include <dbm.h>** or **#include <ndbm.h>**) with the following two lines:

```
#define DB_DBM_HSEARCH    1
#include <db.h>
```

and recompile. If the application attempts to load against a dbm library (for example, **-ldbm**), remove the library from the load line.

Key and **content** parameters are objects described by the **datum** typedef. A **datum** specifies a string of **dsize** bytes pointed to by **dptr**. Arbitrary binary data, as well as normal text strings, are allowed.

Dbm Functions

Before a database can be accessed, it must be opened by dbminit. This will open and/or create the database **file.db**. If created, the database file is created read/write by owner only (as described in **chmod**(2)) and modified by the process' umask value at the time of creation (see **umask**(2)). The group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB.

A database may be closed, and any held resources released, by calling dbmclose.

Once open, the data stored under a key is accessed by fetch, and data is placed under a key by store. A key (and its associated contents) are deleted by delete. A linear pass through all keys in a database may be made, in an (apparently) random order, by using firstkey and nextkey. The firstkey method will return the first key in the databaseThe nextkey method will return the next key in satabase.

The following code will traverse the database:

```
for (key = firstkey(key);
    key.dptr != NULL; key = nextkey(key)) {
        ...
}
```

Ndbm Functions

Before a database can be accessed, it must be opened by dbm_open. This will open and/or create the database file **file.db**, depending on the flags parameter (see **open**(2)). If created, the database file is created with mode **mode** (as described in **chmod**(2)) and modified by the process' umask value at the time of creation (see **umask**(2)). The group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB.

Once open, the data stored under a key is accessed by dbm_fetch, and data is placed under a key by dbm_store. The **flags** field can be either **DBM_INSERT** or **DBM_REPLACE**. **DBM_INSERT** will only insert new entries into the database, and will not change an existing entry with the same key. **DBM_REPLACE** will replace an existing entry if it has the same key. A key (and its associated contents) are deleted by dbm_delete. A linear pass through all keys in a database may be made, in an (apparently) random order, by using dbm_firstkey and dbm_nextkey. The dbm_firstkey method will return the first key in the database. The dbm_nextkey method will return the next key in the database.

The following code will traverse the database:

```
for (key = dbm_firstkey(db);
    key.dptr != NULL; key = dbm_nextkey(db)) {
        ...
}
```

Compatibility Notes

The historic dbm library created two underlying database files, traditionally named **file.dir** and **file.pag**. The Berkeley DB library creates a single database file named **file.db**. Applications that are aware of the underlying database filenames may require additional source code modifications.

The historic dbminit function required that the underlying **.dir** and **.pag** files already exist (empty databases were created by first manually creating zero-length **.dir** and **.pag** files). Applications that expect to create databases using this method may require additional source code modifications.

The historic dbm_dirfno and dbm_pagfno macros are supported, but will return identical file descriptors because there is only a single underlying file used by the Berkeley DB hashing access method. Applications using both file descriptors for locking may require additional source code modifications.

If applications using the dbm function exits without first closing the database, it may lose updates because the Berkeley DB library buffers writes to underlying databases. Such applications will require additional source code modifications to work correctly with the Berkeley DB library.

Dbm Diagnostics

The dbminit function returns -1 on failure, setting **errno**, and 0 on success.

The fetch function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

The store function returns -1 on failure, setting **errno**, and 0 on success.

The delete function returns -1 on failure, setting **errno**, and 0 on success.

The firstkey function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

The nextkey function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

Dbm Errors

The dbminit, fetch, store, delete, firstkey, and nextkey functions may fail and return an error for errors specified for other Berkeley DB and C library or system functions.

Ndbm Diagnostics

The dbm_close method returns non-zero when an error has occurred reading or writing the database.

The dbm_close method resets the error condition on the named database.

The dbm_open function returns NULL on failure, setting **errno**, and a DBM reference on success.

The dbm_fetch function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

The dbm_store function returns -1 on failure, setting **errno**, 0 on success, and 1 if DBM_INSERT was set and the specified key already existed in the database.

The dbm_delete function returns -1 on failure, setting **errno**, and 0 on success.

The dbm_firstkey function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

The dbm_nextkey function sets the **dptr** field of the returned **datum** to NULL on failure, setting **errno**, and returns a non-NULL **dptr** on success.

The dbm_close function returns -1 on failure, setting **errno**, and 0 on success.

The dbm_close function returns -1 on failure, setting **errno**, and 0 on success.

Ndbm Errors

The dbm_open, dbm_close, dbm_fetch, dbm_store, dbm_delete, dbm_firstkey, and dbm_nextkey functions may fail and return an error for errors specified for other Berkeley DB and C library or system functions.

hsearch

```
#define DB_DBM_HSEARCH    1
#include <db.h>

typedef enum {
        FIND, ENTER
} ACTION;

typedef struct entry {
        char *key;
        void *data;
} ENTRY;

ENTRY *
hsearch(ENTRY item, ACTION action);

int
hcreate(size_t nelem);

void
hdestroy(void);
```

The hsearch functions are intended to provide a high-performance implementation and source code compatibility for applications written to the historic hsearch interface. It is not recommended for any other purpose.

To compile hsearch applications, replace the application's **#include** of the hsearch include file (for example, **#include <search.h>**) with the following two lines:

```
#define DB_DBM_HSEARCH    1
        #include <db.h>
```

and recompile.

The hcreate function creates an in-memory database. The **nelem** parameter is an estimation of the maximum number of key/data pairs that will be stored in the database.

The **hdestroy** function discards the database.

Database elements are structures of type **ENTRY**, which contain at least two fields: **key** and **data**. The field **key** is declared to be of type **char ***, and is the key used for storage and retrieval. The field **data** is declared to be of type **void ***, and is its associated data.

The hsearch function retrieves key/data pairs from, and stores key/data pairs into the database.

The **action** parameter must be set to one of two values:

- **ENTER**

If the key does not already appear in the database, insert the key/data pair into the database. If the key already appears in the database, return a reference to an **ENTRY** structure which refers to the existing key and its associated data element.

- **FIND**

  Retrieve the specified key/data pair from the database.

Compatibility Notes

Historically, hsearch required applications to maintain the keys and data in the application's memory for as long as the **hsearch** database existed. Because Berkeley DB handles key and data management internally, there is no requirement that applications maintain local copies of key and data items, although the only effect of doing so should be the allocation of additional memory.

Hsearch Diagnostics

The **hcreate** function returns 0 on failure, setting **errno**, and non-zero on success.

The **hsearch** function returns a pointer to an ENTRY structure on success, and NULL, setting **errno**, if the **action** specified was FIND and the item did not appear in the database.

Hsearch Errors

The **hsearch** function will fail, setting **errno** to 0, if the **action** specified was FIND and the item did not appear in the database.

In addition, the hcreate, hsearch and hdestroy functions may fail and return an error for errors specified for other Berkeley DB and C library or system functions.

Appendix C. Berkeley DB Application Space Static Functions

This appendix describes functionality that existed on the DB_ENV handle in releases prior to Berkeley DB 3.1. In 3.1, this functionality was moved to as series of static functions, as in this appendix.

Static Functions

Static Function	Description
db_env_set_func_close	Replace Berkeley DB calls to close() with the identified function.
db_env_set_func_dirfree	Specify function used to free memory obtained due to a directory walk.
db_env_set_func_dirlist	Specify function used to free memory obtained due to a directory list.
db_env_set_func_exists	Specify function used to determine whether a file exists.
db_env_set_func_file_map	Specify function used to map a file into memory.
db_env_set_func_free	Specify function used to free memory.
db_env_set_func_fsync	Specify function used to sync a file to disk.
db_env_set_func_ftruncate	Specify function used to truncate a file.
db_env_set_func_ioinfo	Specify function used to determine file characteristics.
db_env_set_func_malloc	Specify function used to allocate memory.
db_env_set_func_open	Specify function used to open a file.
db_env_set_func_pread	Specify function used to read data from an object.
db_env_set_func_pwrite	Specify function used to write data to an object.
db_env_set_func_read	Specify function used to read data from an object.
db_env_set_func_realloc	Specify function used to change the size of memory pointed to by a pointer.
db_env_set_func_region_map	Specify function used to created shared memory regions.
db_env_set_func_rename	Specify function used to change the name of a file.
db_env_set_func_seek	Specify function used to specify a location in a file.
db_env_set_func_unlink	Specify function used to delete a file.
db_env_set_func_write	Specify function used to write data to an object.
db_env_set_func_yield	Specify function used to yield the processor to another thread of control.

db_env_set_func_close

```
#include <db.h>

int
db_env_set_func_close(int (*func_close)(int fd));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **close** function with **func_close**, which must conform to the standard interface specification.

The `db_env_set_func_close()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_close()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_close()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_close

The **func_close** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_dirfree

```
#include <db.h>

int
db_env_set_func_dirfree(void (*func_dirfree)(char **namesp, int cnt));
```

The Berkeley DB library requires the ability to return any memory allocated as part of the routine which reads through a directory and creates a list of files that the directory contains (see db_env_set_func_dirlist (page 623)).

The `db_env_set_func_dirfree()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_dirfree()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_dirfree()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_dirfree

The **func_dirfree** parameter is a function which frees the memory returned from the db_env_set_func_dirlist (page 623) function.

The **namesp** and **cnt** parameters to this function are the same values as were returned by the db_env_set_func_dirlist (page 623) function.

See Also

Run-time configuration

db_env_set_func_dirlist

```
#include <db.h>

int
db_env_set_func_dirlist(int (*func_dirlist)(const char *dir,
    char ***namesp, int *cntp));
```

The Berkeley DB library requires the ability to read through a directory and create a list of files that the directory contains.

The db_env_set_func_dirlist method configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the db_env_set_func_dirlist method may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_dirlist()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_dirlist

The **func_dirlist** parameter is the function which reads through a directory and returns a list of the files it contains.

The **dir** parameter to this function is the name of the directory to be searched.

The function must return a pointer to an array of nul-terminated file names into the memory location to which the **namesp** parameter refers, and a count of the number of elements in the array into the memory location to which **cntp** refers.

See Also

Run-time configuration

db_env_set_func_exists

```
#include <db.h>

int
db_env_set_func_exists(int (*func_exists)(const char *path,
                         int *isdirp));
```

The Berkeley DB library requires the ability to determine whether a file exists and whether it is a file of type directory.

The `db_env_set_func_exists()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_exists()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_exists()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_exists

The **func_exists** parameter is the function which returns if a file exists and if it is a file of type directory.

The **path** parameter to this function is the pathname of the file to be checked.

If the **isdirp** parameter is non-NULL, it must be set to non-0 if **path** is a directory, and 0 if **path** is not a directory.

The **func_exists** function must return the value of **errno** on failure and 0 on success.

See Also

Run-time configuration

db_env_set_func_file_map

```
#include <db.h>

int
db_env_set_func_file_map(int (*func_file_map)(DB_ENV *dbenv, char *path,
    size_t len, int is_rdonly, void **addr),
    int (*func_file_unmap)(DB_ENV *dbenv, void *addr));
```

The Berkeley DB library optionally uses the ability to map a file into memory.

The `db_env_set_func_file_map()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_file_map()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_file_map()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_file_map

The **func_file_map** parameter is the function which maps a file into memory. The function takes 5 parameters:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle.

- **path**

  The **path** parameter is the name of file. Repeated requests for the mapping of the same name should return a reference to the same memory.

- **len**

  The **len** parameter is the length, in bytes, of the file.

- **is_rdonly**

  The **is_rdonly** parameter will be non-zero if the mapped file is read-only.

- **addr**

  The **addr** parameter is the memory location into which a pointer to the mapped file is returned.

The **func_file_map** function must return the value of **errno** on failure and 0 on success.

func_file_unmap

The **func_file_unmap** parameter is the function which unmaps a file from memory. The function takes 2 parameters:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle.

- `addr`

  The **addr** parameter is the value returned by the **func_file_map** function when the file or region was mapped into memory.

See Also

Run-time configuration

db_env_set_func_free

```
#include <db.h>

int
db_env_set_func_free(void (*func_free)(void *ptr));
```

Replace Berkeley DB calls to the ANSI C X3.159-1989 (ANSI C) standard **free** function with **func_free**, which must conform to the standard interface specification.

The `db_env_set_func_free()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_free()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_free()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_free

The **func_free** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_fsync

```
#include <db.h>

int
db_env_set_func_fsync(int (*func_fsync)(int fd));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **fsync** function with **func_fsync**, which must conform to the standard interface specification.

The `db_env_set_func_fsync()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_fsync()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_fsync` function returns a non-zero error value on failure and 0 on success.

Parameters

func_fsync

The **func_fsync** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_ftruncate

```
#include <db.h>

int
db_env_set_func_ftruncate(int (*func_ftruncate)(int fd, off_t offset));
```

The Berkeley DB library requires the ability to truncate a file.

The db_env_set_func_ftruncate function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the db_env_set_func_ftruncate function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The db_env_set_func_ftruncate() function returns a non-zero error value on failure and 0 on success.

Parameters

func_ftruncate

The **func_ftruncate** parameter is the function which truncates a file.

The **fd** parameter is an open file descriptor on the file.

The **ftruncate** function must truncate the file to the byte length specified by the **offset** parameter.

The **func_ftruncate** function must return the value of **errno** on failure and 0 on success.

See Also

Run-time configuration

db_env_set_func_ioinfo

```
#include <db.h>

int
db_env_set_func_ioinfo(int (*func_ioinfo)(const char *path,
    int fd, u_int32_t *mbytesp, u_int32_t *bytesp, u_int32_t *iosizep));
```

The Berkeley DB library requires the ability to determine the size and I/O characteristics of a file.

The `db_env_set_func_ioinfo()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_ioinfo()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_ioinfo()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_ioinfo

The **func_ioinfo** parameter is the function which returns the size and I/O characteristics of a file.

The **path** parameter is the pathname of the file to be checked, and the **fd** parameter is an open file descriptor on the file.

If the **mbytesp** and **bytesp** parameters are non-NULL, the **ioinfo** function must return in them the size of the file: the number of megabytes in the file into the memory location to which the **mbytesp** parameter refers, and the number of bytes over and above that number of megabytes into the memory location to which the **bytesp** parameter refers.

In addition, if the **iosizep** parameter is non-NULL, the **ioinfo** function must return the optimum granularity for I/O operations to the file into the memory location to which it refers.

The **func_ioinfo** function must return the value of **errno** on failure and 0 on success.

See Also

Run-time configuration

db_env_set_func_malloc

```
#include <db.h>

int
db_env_set_func_malloc(void *(*func_malloc)(size_t size));
```

Replace Berkeley DB calls to the ANSI C X3.159-1989 (ANSI C) standard **malloc** function with **func_malloc**, which must conform to the standard interface specification.

The `db_env_set_func_malloc()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_malloc()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_malloc()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_malloc

The **func_malloc** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_open

```
#include <db.h>

int
db_env_set_func_open(int (*func_open)(const char *path, int flags,
    int mode));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **open** function with **func_open**, which must conform to the standard interface specification.

The `db_env_set_func_open()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_open()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_open()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_open

The **func_open** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_pread

```
#include <db.h>

int
db_env_set_func_pread(ssize_t (*func_pread)(int fd, void *buf,
    size_t nbytes, off_t offset));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **pread** function with **func_pread**, which must conform to the standard interface specification.

The `db_env_set_func_pread()` configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_pread()` may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_pread()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_pread

The **func_pread** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_pwrite

```
#include <db.h>

int
db_env_set_func_pwrite(ssize_t (*func_pwrite)(int fd, const void *buf,
    size_t nbytes, off_t offset));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **pwrite** function with **func_pwrite**, which must conform to the standard interface specification.

The `db_env_set_func_pwrite()` configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_pwrite()` may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_pwrite()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_pwrite

The **func_pwrite** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_read

```
#include <db.h>

int
db_env_set_func_read(ssize_t (*func_read)(int fd, void *buf,
    size_t nbytes));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **read** function with **func_read**, which must conform to the standard interface specification.

The `db_env_set_func_read()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_read()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_read()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_read

The **func_read** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_realloc

```
#include <db.h>

int
db_env_set_func_realloc(void *(*func_realloc)(void *ptr, size_t size));
```

Replace Berkeley DB calls to the ANSI C X3.159-1989 (ANSI C) standard **realloc** function with **func_realloc**, which must conform to the standard interface specification.

The `db_env_set_func_realloc()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_realloc()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_realloc()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_realloc

The **func_realloc** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_region_map

```
#include <db.h>

int
db_env_set_func_region_map(int (*func_region_map)(DB_ENV *dbenv,
     char *path, size_t len, int *is_create, void **addr),
     int (*func_region_unmap)(DB_ENV *dbenv, void *addr));
```

The Berkeley DB library optionally uses the ability to create shared memory regions (which may or may not be backed by physical files). The memory will be used as a shared memory region for synchronization between Berkeley DB threads/processes; while the returned memory may be of any kind (for example, anonymous memory), it must be able to support semaphores.

The `db_env_set_func_region_map()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_region_map()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_region_map()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_region_map

The **func_region_map** parameter is the function which creates shared memory regions. The function takes 5 parameters:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle. This handle is provided to uniquely identify a shared memory region: the **dbenv** parameter and the path are a unique identifier pair for mapping any new region, and the **dbenv** parameter and the address are a unique identifier pair for unmapping any region.

- **path**

  The **path** parameter is the name of the region. Repeated requests for the shared regions of the same name, in the same database environment, should return a reference to the same memory.

- **len**

  The **len** parameter is the length, in bytes, needed for the region.

- **is_create**

The memory referenced by the **is_create** parameter will be non-zero if flags to Berkeley DB allowed creation of the mapped region; the memory referenced by the **is_create** parameter must be set to non-zero if the region is created by the **func_region_map** function, and set to zero if the region is not created by the function. This returned information will determine if the region is subsequently initialized by Berkeley DB.

- **addr**

  The **addr** parameter is the memory location into which a pointer to the region or mapped file is returned.

func_region_unmap

The **func_region_unmap** parameter is the function which unmaps a shared memory region. The function takes 2 parameters:

- **dbenv**

  The **dbenv** parameter is the enclosing database environment handle.

- **addr**

  The **addr** parameter is the value returned by the **func_region_map** function when the region was mapped into memory.

See Also

Run-time configuration

db_env_set_func_rename

```
#include <db.h>

int
db_env_set_func_rename(int (*func_rename)(const char *from,
    const char *to));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **rename** function with **func_rename**, which must conform to the standard interface specification.

The `db_env_set_func_rename()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_rename()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_rename()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_rename

The **func_rename** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_seek

```
#include <db.h>

int
db_env_set_func_seek(int (*func_seek)(int fd, off_t offset, int whence));
```

The Berkeley DB library requires the ability to specify that a subsequent read from or write to a file will occur at a specific location in that file.

The db_env_set_func_seek() function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the db_env_set_func_seek() function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The db_env_set_func_seek() function returns a non-zero error value on failure and 0 on success.

Parameters

func_seek

The **func_seek** parameter is the function which seeks to a specific location in a file.

The **fd** parameter is an open file descriptor on the file.

The **seek** function must cause a subsequent read from or write to the file to occur at the byte offset specified by the **offset** parameter.

The **whence** parameter specifies where in the file the byte offset is relative to, as described by the IEEE/ANSI Std 1003.1 (POSIX) **lseek** system call.

The **func_seek** function must return the value of **errno** on failure and 0 on success.

See Also

Run-time configuration

db_env_set_func_unlink

```
#include <db.h>

int
db_env_set_func_unlink(int (*func_unlink)(const char *path));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **unlink** function with **func_unlink**, which must conform to the standard interface specification.

The `db_env_set_func_unlink()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_unlink()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_unlink()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_unlink

The **func_unlink** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_write

```
#include <db.h>

int
db_env_set_func_write(ssize_t (*func_write)(int fd, const void *buffer,
    size_t nbytes));
```

Replace Berkeley DB calls to the IEEE/ANSI Std 1003.1 (POSIX) **write** function with **func_write**, which must conform to the standard interface specification.

The `db_env_set_func_write()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_write()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_write()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_write

The **func_write** parameter is the replacement function. It must conform to the standard interface specification.

See Also

Run-time configuration

db_env_set_func_yield

```
#include <db.h>

int
db_env_set_func_yield(int (*func_yield)(u_long secs, u_long usecs));
```

The Berkeley DB library requires the ability to yield the processor from the current thread of control to any other waiting threads of control.

The **func_yield** function must be able to cause the rescheduling of all participants in the current Berkeley DB environment, whether threaded or not. It may be incorrect to supply a thread **yield** function if more than a single process is operating in the Berkeley DB environment. This is because many thread-yield functions will not allow other processes to run, and the contested lock may be held by another process, not by another thread.

The `db_env_set_func_yield()` function configures all operations performed by a process and all of its threads of control, not operations confined to a single database environment.

Although the `db_env_set_func_yield()` function may be called at any time during the life of the application, it should normally be called before making calls to the db_env_create (page 199) or db_create (page 16) methods.

The `db_env_set_func_yield()` function returns a non-zero error value on failure and 0 on success.

Parameters

func_yield

The **func_yield** parameter is the function which yields the processor.

The **secs** parameter is the number of seconds to pause before the thread of control should run again, or 0.

The **usecs** parameter is the number of microseconds to pause before the thread of control should run again, or 0.

The **func_yield** function must return the value of **errno** on failure and 0 on success.

See Also

Run-time configuration