Oracle Berkeley DB

Berkeley DB
API Reference
for C++

12c Release 1
Library Version 12.1.6.2
Legal Notice

This documentation is distributed under an open source license. You may review the terms of this license at: http://www.oracle.com/technetwork/database/berkeleydb/downloads/oslicense-093458.html

Oracle, Berkeley DB, and Sleepycat are trademarks or registered trademarks of Oracle. All rights to these marks are reserved. No third-party use is permitted without the express prior written consent of Oracle.

Other names may be trademarks of their respective owners.

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

Published 4/13/2017
# Table of Contents

Preface ..................................................................................................... xiv  
Conventions Used in this Book ................................................................. xv  
For More Information ............................................................................. xvi  
1. Introduction to Berkeley DB APIs ........................................................ 1  
2. The Db Handle ......................................................................................... 2  
   Database and Related Methods .............................................................. 3  
   Db::associate() ....................................................................................... 6  
   Db::associate_foreign() ......................................................................... 10  
   Db::close() ........................................................................................... 13  
   Db::compact() ....................................................................................... 16  
   db_copy .............................................................................................. 20  
   Db ..................................................................................................... 21  
   Db::del() ............................................................................................. 23  
   Db::err() ............................................................................................. 26  
   Db::exists() .......................................................................................... 28  
   Db::fd() ............................................................................................. 30  
   Db::get() ............................................................................................. 31  
   Db::get_bt_minkey() ............................................................................... 36  
   Db::get_byteswapped() ............................................................................ 37  
   Db::get_cachesize() ................................................................................ 38  
   Db::get_create_dir() ............................................................................... 39  
   Db::get_dbname() .................................................................................. 40  
   Db::get_encrypt_flags() ........................................................................... 41  
   Db::get_errfile() .................................................................................... 42  
   Db::get_errpfx() .................................................................................... 43  
   Db::get_flags() ...................................................................................... 44  
   Db::get_h_ffactor() ................................................................................ 45  
   Db::get_h_nelem() ............................................................................... 46  
   Db::get_heapsize() ................................................................................. 47  
   Db::get_heap_regionsize() ....................................................................... 48  
   Db::get_lk_exclusive() ............................................................................. 49  
   Db::get_lorder() ................................................................................... 50  
   Db::get_msgfile() ................................................................................... 51  
   Db::get_msgpfx() ................................................................................... 52  
   Db::get_multiple() ............................................................................... 53  
   Db::get_open_flags() ............................................................................. 54  
   Db::get_partition_callback() ............................................................... 55  
   Db::get_partition_dirs() ......................................................................... 56  
   Db::get_partition_keys() ........................................................................ 57  
   Db::getpagesize() ............................................................................... 58  
   Db::get_priority() ................................................................................ 59  
   Db::get_q_extentsize() .......................................................................... 60  
   Db::get_re_delim() ............................................................................... 61  
   Db::get_re_len() ................................................................................. 62  
   Db::get_re_pad() ................................................................................... 63  
   Db::get_re_source() .............................................................................. 64
Db::stat() ........................................................................................... 157
Db::stat_print() .................................................................................... 165
Db::sync() .......................................................................................... 166
Db::truncate() ..................................................................................... 168
Db::upgrade() ...................................................................................... 170
Db::verify() ......................................................................................... 172
DbHeapRecordId ................................................................................... 175

3. The Dbc Handle ..................................................................................... 177
   Database Cursors and Related Methods ....................................................... 178
   Db::cursor() ........................................................................................ 179
   Dbc::close() ........................................................................................ 182
   Dbc::cmp() ......................................................................................... 184
   Dbc::count() ....................................................................................... 186
   Dbc::del() .......................................................................................... 188
   Dbc::dup() .......................................................................................... 191
   Dbc::get() .......................................................................................... 193
   Dbc::get_priority() ................................................................................ 201
   Dbc::put() .......................................................................................... 202
   Dbc::set_priority() ................................................................................ 206

4. The Dbt Handle ..................................................................................... 207
   DBT and Bulk Operations ........................................................................ 212
   DbMultipleIterator ................................................................................ 213
   DbMultipleDataIterator .......................................................................... 214
   DbMultipleKeyDataIterator ..................................................................... 216
   DbMultipleRecnoDataIterator .................................................................. 218
   DbMultipleBuilder ................................................................................. 220
   DbMultipleDataBuilder ........................................................................... 221
   DbMultipleKeyDataBuilder ....................................................................... 223
   DbMultipleRecnoDataBuilder ................................................................... 225

5. The DbEnv Handle .................................................................................. 227
   Database Environments and Related Methods ........................................... 228
   DbEnv::add_data_dir() .......................................................................... 230
   DbEnv::backup() .................................................................................. 232
   DbEnv::close() .................................................................................... 235
   DbEnv ............................................................................................... 237
   DbEnv::dbbackup() ............................................................................... 239
   DbEnv::dbremove() ............................................................................. 241
   DbEnv::dbrename() ............................................................................. 243
   DbEnv::err() ...................................................................................... 246
   DbEnv::failchk() ................................................................................ 248
   DbEnv::fileid_reset() ......................................................................... 250
   DbEnv::full_version() .......................................................................... 252
   DbEnv::get_create_dir() ....................................................................... 253
   DbEnv::get_data_dirs() ....................................................................... 254
   DbEnv::get_encrypt_flags() .................................................................... 255
   Db::get_env() .................................................................................... 256
   DbEnv::get_errfile() ........................................................................... 257
   DbEnv::get_errpfx() ............................................................................ 258
   DbEnv::get_backup_callbacks() .............................................................. 259
DbEnv::get_backup_config() ................................................................. 260
DbEnv::get_flags() ........................................................................... 261
DbEnv::get_home() .......................................................................... 262
DbEnv::get_intermediate_dir_mode() .................................................... 263
DbEnv::get_memory_init() .................................................................. 264
DbEnv::get_memory_max() .................................................................. 266
DbEnv::get_metadata_dir() ................................................................. 267
DbEnv::get_msgfile() ..................................................................... 268
DbEnv::get_msgpfx() ..................................................................... 269
DbEnv::get_open_flags() ................................................................... 270
DbEnv::getshm_key() .................................................................... 271
DbEnv::get_slices() ....................................................................... 272
DbEnv::get_slice_count() .................................................................. 273
DbEnv::get_thread_count() ............................................................... 274
DbEnv::get_timeout() .................................................................... 275
DbEnv::get_tmp_dir() ..................................................................... 276
DbEnv::getverbose() ....................................................................... 277
DbEnv::log_verify() ....................................................................... 279
DbEnv::lsn_reset() ........................................................................ 282
DbEnv::msg() ................................................................................ 284
DbEnv::open() .............................................................................. 286
DbEnv::remove() .......................................................................... 292
DbEnv::set_alloc() ......................................................................... 294
DbEnv::set_app_dispatch() ............................................................... 296
DbEnv::set_backup_callbacks() ....................................................... 298
DbEnv::set_backup_config() ............................................................. 301
DbEnv::set_data_dir() .................................................................... 303
DbEnv::set_create_dir() .................................................................. 305
DbEnv::set_encrypt() .................................................................... 307
DbEnv::set_event_notify() ............................................................... 309
DbEnv::set_errcall() ...................................................................... 315
DbEnv::set_errfile() ....................................................................... 317
DbEnv::set_error_stream() ............................................................. 319
DbEnv::set_errpfx() ...................................................................... 320
DbEnv::set_feedback() .................................................................. 321
DbEnv::set_flags() ....................................................................... 323
DbEnv::set_intermediate_dir_mode() ................................................. 330
DbEnv::set_isalive() ..................................................................... 332
DbEnv::set_memory_init() ................................................................ 334
DbEnv::set_memory_max() ................................................................ 336
DbEnv::set_metadata_dir() ............................................................... 338
DbEnv::set_message_stream() .......................................................... 339
DbEnv::set_msgcall() ................................................................... 340
DbEnv::set_msgfile() ................................................................... 342
DbEnv::set_msgpfx() .................................................................... 343
DbEnv::setshm_key() ................................................................... 344
DbEnv::set_thread_count() ............................................................. 346
DbEnv::set_thread_id() .................................................................. 348
DbEnv::set_thread_id_string() ........................................................ 350
6. The DbException Class ................................................................. 363
   DB C++ Exceptions ....................................................................... 364
   DbDeadlockException .................................................................... 365
   DbLockNotGrantedException ......................................................... 366
   DbMemoryException ...................................................................... 368
   DbRepHandleDeadException .......................................................... 369
   DbRunRecoveryException ............................................................... 370

7. The DbLock Handle ...................................................................... 371
   Locking Subsystem and Related Methods ........................................ 372
   DbEnv::get_lk_conflicts() ............................................................. 373
   DbEnv::get_lk_detect() .................................................................. 374
   DbEnv::get_lk_max_lockers() .......................................................... 375
   DbEnv::get_lk_max_locks() ............................................................. 376
   DbEnv::get_lk_max_objects() .......................................................... 377
   DbEnv::get_lk_partitions() .............................................................. 378
   DbEnv::get_lk_priority() ................................................................. 379
   DbEnv::get_lk_tablesize() ............................................................... 380
   DbEnv::set_lk_conflicts() ............................................................... 381
   DbEnv::set_lk_detect() .................................................................. 382
   DbEnv::set_lk_max_lockers() ............................................................ 383
   DbEnv::set_lk_max_locks() ............................................................... 384
   DbEnv::set_lk_max_objects() ............................................................ 385
   DbEnv::set_lk_partitions() ............................................................... 386
   DbEnv::set_lk_priority() ................................................................. 387
   DbEnv::set_lk_tablesize() ............................................................... 388
   DbEnv::lock_detect() ..................................................................... 390
   DbEnv::lock_get() ........................................................................ 391
   DbEnv::lock_id() ........................................................................... 392
   DbEnv::lock_id_free() ................................................................... 393
   DbEnv::lock_put() ......................................................................... 394
   DbEnv::lock_stat() ........................................................................ 395
   DbEnv::lock_stat_print() ............................................................... 396
   DbEnv::lock_vec() ......................................................................... 397

8. The DbLsn Handle ...................................................................... 416
   Logging Subsystem and Related Methods ........................................ 417
   DbEnv::get_lg_bsize() ................................................................... 418
   DbEnv::get_lg_dir() ....................................................................... 419
   DbEnv::get_lg_filemode() ............................................................... 420
   DbEnv::get_lg_max() ..................................................................... 421
   DbEnv::get_lg_regionmax() ............................................................. 422
   DbEnv::log_archive() .................................................................... 423
   DbEnv::log_cursor() ...................................................................... 424
   DbEnv::log_file() ......................................................................... 425
   DbEnv::log_vec() .......................................................................... 426
DbEnv::log_flush() ................................................................................ 427
DbEnv::log_get_config() .......................................................................... 428
DbEnv::log_printf() ............................................................................... 430
DbEnv::log_put() .................................................................................. 431
DbEnv::log_set_config() .......................................................................... 433
DbEnv::log_stat() ................................................................................ 437
DbEnv::log_stat_print() .......................................................................... 441
DbEnv::set_lg_bsize() ............................................................................. 442
DbEnv::set_lg_dir() ............................................................................... 444
DbEnv::set_lg_filemode() ........................................................................ 446
DbEnv::set_lg_max() .............................................................................. 447
DbEnv::set_lg_regionmax() ...................................................................... 449
The DbLogc Handle ............................................................................. 451
DbLogc::close() ................................................................................... 452
DbLogc::get() ...................................................................................... 453
DbEnv::log_compare() ............................................................................ 455
9. The DbMpoolFile Handle ........................................................................... 456
Memory Pools and Related Methods ........................................................... 457
Db::get_mpf() ..................................................................................... 459
DbEnv::get_cache_max() ........................................................................ 460
DbEnv::get_cachesize() .......................................................................... 461
DbEnv::get_mp_max_openfd() .................................................................. 462
DbEnv::get_mp_max_write() .................................................................... 463
DbEnv::get_mp_mmapsize() ..................................................................... 464
DbEnv::get_mp_mtxcount() ..................................................................... 465
DbEnv::get_mp_pagesize() ...................................................................... 466
DbEnv::get_mp_tablesize() ...................................................................... 467
DbEnv::memp_fcreate() .......................................................................... 468
DbEnv::memp_register() ......................................................................... 469
DbEnv::memp_stat() .............................................................................. 471
DbEnv::memp_stat_print() ...................................................................... 477
DbEnv::memp_sync() ............................................................................... 478
DbEnv::memp_trickle() ........................................................................... 479
DbEnv::set_cache_max() ......................................................................... 480
DbEnv::set_cachesize() .......................................................................... 482
DbEnv::set_mp_max_openfd() .................................................................. 484
DbEnv::set_mp_max_write() .................................................................... 485
DbEnv::set_mp_mmapsize() ..................................................................... 487
DbEnv::set_mp_mtxcount() ...................................................................... 489
DbEnv::set_mp_pagesize() ...................................................................... 490
DbEnv::set_mp_tablesize() ...................................................................... 491
DbEnv::set_region_dir() ........................................................................ 492
DbMpoolFile::close() .............................................................................. 494
DbMpoolFile::get() ................................................................................ 495
DbMpoolFile::open() ............................................................................... 498
DbMpoolFile::put() .............................................................................. 500
DbMpoolFile::sync() ............................................................................... 502
DbMpoolFile::get_clear_len() .................................................................. 503
DbMpoolFile::get_fileid() ........................................................................ 504
DbMpoolFile::get_flags() ................................................................. 505
DbMpoolFile::get_fype() ................................................................. 506
DbMpoolFile::get_lsn_offset() ....................................................... 507
DbMpoolFile::get_maxsize() .......................................................... 508
DbMpoolFile::get_pgcookie() ......................................................... 509
DbMpoolFile::get_priority() .......................................................... 510
DbEnv::get_region_dir() ............................................................... 511
DbMpoolFile::set_clear_len() ......................................................... 512
DbMpoolFile::set_fileid() ............................................................. 513
DbMpoolFile::set_flags() .............................................................. 515
DbMpoolFile::set_fype() ............................................................... 517
DbMpoolFile::set_lsn_offset() ....................................................... 518
DbMpoolFile::set_maxsize() .......................................................... 519
DbMpoolFile::set_pgcookie() ......................................................... 520
DbMpoolFile::set_priority() .......................................................... 521
10. Mutex Methods .......................................................................... 523
    Mutex Methods ........................................................................... 524
    DbEnv::mutex_alloc() ............................................................... 525
    DbEnv::mutex_free() ............................................................... 527
    DbEnv::mutex_get_align() ......................................................... 528
    DbEnv::mutex_get_increment() ................................................. 529
    DbEnv::mutex_get_init() .......................................................... 530
    DbEnv::mutex_get_max() .......................................................... 531
    DbEnv::mutex_get_tas_spins() .................................................. 532
    DbEnv::mutex_lock() ............................................................... 533
    DbEnv::mutex_set_align() ......................................................... 534
    DbEnv::mutex_set_increment() ................................................. 536
    DbEnv::mutex_set_init() .......................................................... 538
    DbEnv::mutex_set_max() .......................................................... 539
    DbEnv::mutex_set_tas_spins() .................................................. 541
    DbEnv::mutex_stat() ............................................................... 542
    DbEnv::mutex_stat_print() ...................................................... 545
    DbEnv::mutex_unlock() ............................................................ 546
11. Replication Methods ................................................................. 547
    Replication and Related Methods .............................................. 548
    The DbSite Handle .................................................................... 550
    The DbChannel Handle ............................................................. 551
    DbChannel::close() ................................................................. 552
    DbChannel::send_msg() .......................................................... 553
    DbChannel::send_request() ...................................................... 555
    DbChannel::set_timeout() ...................................................... 557
    DbSite::get_config() .............................................................. 558
    DbSite::get_address() ............................................................ 559
    DbSite::get_eid() ................................................................. 560
    DbSite::remove() ................................................................... 561
    DbSite::set_config() .............................................................. 562
    DbEnv::rep_elect() ................................................................. 564
    DbEnv::rep_get_clockskew() .................................................... 567
    DbEnv::rep_get_config() ........................................................ 568
DbEnv::rep_get_limit() ........................................................................... 569
DbEnv::rep_get_nsites() .......................................................................... 570
DbEnv::rep_get_priority() ....................................................................... 571
DbEnv::rep_get_request() ....................................................................... 572
DbEnv::rep_get_timeout() ....................................................................... 573
DbEnv::rep_process_message() ................................................................. 574
DbEnv::rep_set_clockskew() ..................................................................... 577
DbEnv::rep_set_config() ......................................................................... 579
DbEnv::rep_set_limit() ........................................................................... 583
DbEnv::rep_set_nsites() .......................................................................... 585
DbEnv::rep_set_priority() ........................................................................ 587
DbEnv::rep_set_request() ........................................................................ 589
DbEnv::rep_set_timeout() ....................................................................... 591
DbEnv::rep_set_transport() ...................................................................... 594
DbEnv::rep_set_view() ........................................................................... 597
DbEnv::rep_start() ................................................................................ 599
DbEnv::rep_stat() ................................................................................ 601
DbEnv::rep_stat_print() .......................................................................... 608
DbEnv::rep_sync() ................................................................................. 609
DbEnv::repmgr_channel() ........................................................................ 611
DbEnv::repmgr_local_site() ...................................................................... 613
DbEnv::repmgr_get_ack_policy() ............................................................... 614
DbEnv::repmgr_get_incoming_queue_max()  ................................................ 615
DbEnv::repmgr_msg_dispatch() ................................................................. 616
DbEnv::repmgr_set_ack_policy() ............................................................... 618
DbEnv::repmgr_set_incoming_queue_max()  ................................................ 620
DbEnv::repmgr_set_socket() .................................................................... 622
DbEnv::repmgr_site() ............................................................................. 624
DbEnv::repmgr_site_by_eid() ................................................................... 626
DbEnv::repmgr_site_list() ........................................................................ 627
DbEnv::repmgr_start() ........................................................................... 629
DbEnv::repmgr_stat() ............................................................................. 632
DbEnv::repmgr_stat_print() ..................................................................... 635
DbSite::close() ..................................................................................... 636
DbEnv::txn_applied() ............................................................................. 637
DbTnx::set_commit_token() ....................................................................... 639

12. The DbSequence Handle ......................................................................... 640
Sequences and Related Methods ............................................................... 641
DbSequence ........................................................................................ 642
DbSequence::close() .............................................................................. 644
DbSequence::get() ................................................................................ 645
DbSequence::get_cachesize() ................................................................... 647
DbSequence::get_dbp() .......................................................................... 648
DbSequence::get_flags() ......................................................................... 649
DbSequence::get_key() ........................................................................... 650
DbSequence::get_range() ........................................................................ 651
DbSequence::initial_value() ..................................................................... 652
DbSequence::open() .............................................................................. 653
DbSequence::remove() ........................................................................... 655
13. The DbTxn Handle .............................................................................. 663
Transaction Subsystem and Related Methods ........................................... 664
Db::get_transactional() ....................................................................... 665
DbEnv::cdsgroup_begin() ...................................................................... 666
DbEnv::get_tx_max() .......................................................................... 667
DbEnv::get_tx_timestamp() .................................................................. 668
DbEnv::set_tx_max() .......................................................................... 669
DbEnv::set_tx_timestamp() .................................................................. 671
DbEnv::txn_recover() ......................................................................... 672
DbEnv::txn_begin() ........................................................................... 674
DbEnv::txn_checkpoint() ..................................................................... 678
DbEnv::txn_stat() ............................................................................ 680
DbEnv::txn_stat_print() ..................................................................... 684
DbTxn::abort() ................................................................................. 685
DbTxn::commit() .............................................................................. 686
DbTxn::discard() .............................................................................. 689
DbTxn::get_name() .......................................................................... 691
DbTxn::get_priority() ....................................................................... 692
DbTxn::id() .................................................................................... 693
DbTxn::prepare() .............................................................................. 694
DbTxn::set_name() .......................................................................... 696
DbTxn::set_priority() ....................................................................... 697
DbTxn::set_timeout() ....................................................................... 698
14. External Files ................................................................................... 700
External Files and Related Methods ....................................................... 701
Db::get_ext_file_dir() ........................................................................ 702
Db::get_ext_file_threshold() ................................................................. 703
Db::set_ext_file_dir() ........................................................................ 704
Db::set_ext_file_threshold() ................................................................. 705
Dbc::db_stream() ............................................................................... 707
DbStream::close() ............................................................................ 709
DbStream::read() ............................................................................. 710
DbStream::size() ............................................................................... 712
DbStream::write() ............................................................................ 713
DbEnv::get_ext_file_dir() .................................................................... 715
DbEnv::get_ext_file_threshold() ........................................................... 716
DbEnv::set_ext_file_dir() .................................................................... 717
DbEnv::set_ext_file_threshold() ............................................................ 718
A. Berkeley DB Command Line Utilities ..................................................... 720
Utilities ............................................................................................ 721
db_archive ...................................................................................... 722
db_checkpoint .................................................................................. 724
db_convert ....................................................................................... 726
db_deadlock ..................................................................................... 728
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>slice</td>
<td>808</td>
</tr>
<tr>
<td>log_set_config</td>
<td>809</td>
</tr>
<tr>
<td>set_mp_max_openfd</td>
<td>810</td>
</tr>
<tr>
<td>set_mp_max_write</td>
<td>811</td>
</tr>
<tr>
<td>set_mp_mmapsize</td>
<td>812</td>
</tr>
<tr>
<td>set_open_flags</td>
<td>813</td>
</tr>
<tr>
<td>set_region_dir</td>
<td>814</td>
</tr>
<tr>
<td>set_shm_key</td>
<td>815</td>
</tr>
<tr>
<td>set_thread_count</td>
<td>816</td>
</tr>
<tr>
<td>set_timeout</td>
<td>817</td>
</tr>
<tr>
<td>set_tmp_dir</td>
<td>818</td>
</tr>
<tr>
<td>set_tx_max</td>
<td>819</td>
</tr>
<tr>
<td>set.verbose</td>
<td>820</td>
</tr>
</tbody>
</table>
Preface

Welcome to Berkeley DB 12c Release 1 (DB). This document describes the C++ API for DB library version 12.1.6.2. 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 this manual:

Class names are represented in monospaced font, as are method names. For example: "Db::open() is a Db class method."

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:

```c
typedef struct vendor {
    char name[MAXFIELD];         // Vendor name
    char street[MAXFIELD];       // Street name and number
    char city[MAXFIELD];         // City
    char state[3];               // Two-digit US state code
    char zipcode[6];             // US zipcode
    char phone_number[13];       // Vendor phone number
} VENDOR;
```

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 Reference Guide
- Berkeley DB STL API Reference Guide
- Berkeley DB TCL API Reference Guide
- Berkeley DB Installation and Build Guide
- Berkeley DB Programmer’s Reference Guide
- Berkeley DB Getting Started with the SQL APIs

To download the latest Berkeley DB documentation along with white papers and other collateral, visit http://www.oracle.com/technetwork/indexes/documentation/index.html.


Contact Us


For sales or support information, email to: berkeleydb-info_us@oracle.com You can subscribe to a low-volume email announcement list for the Berkeley DB product family by sending email to: bdb-join@oss.oracle.com
Chapter 1. Introduction to Berkeley DB APIs


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 DbLsn Handle (page 416) 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 (page 21) constructor. For most database activities, you must then open the handle using the Db::open() (page 75) 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 88), Db::remove() (page 86) or Db::verify() (page 172) 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 DbEnv 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 75) method for information on specifying database types.
## Database and Related Methods

<table>
<thead>
<tr>
<th>Database Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db::associate()</td>
<td>Associate a secondary index</td>
</tr>
<tr>
<td>Db::associate_foreign()</td>
<td>Associate a foreign index</td>
</tr>
<tr>
<td>Db::close()</td>
<td>Close a database</td>
</tr>
<tr>
<td>Db::compact()</td>
<td>Compact a database</td>
</tr>
<tr>
<td>Db</td>
<td>Create a database handle</td>
</tr>
<tr>
<td>Db::del()</td>
<td>Delete items from a database</td>
</tr>
<tr>
<td>Db::err()</td>
<td>Error message</td>
</tr>
<tr>
<td>Db::exists()</td>
<td>Return if an item appears in a database</td>
</tr>
<tr>
<td>Db::fd()</td>
<td>Return a file descriptor from a database</td>
</tr>
<tr>
<td>Db::get()</td>
<td>Get items from a database</td>
</tr>
<tr>
<td>Db::get_byteswapped()</td>
<td>Return if the underlying database is in host order</td>
</tr>
<tr>
<td>Db::get_dbname()</td>
<td>Return the file and database name</td>
</tr>
<tr>
<td>Db::get_multiple()</td>
<td>Return if the database handle references multiple databases</td>
</tr>
<tr>
<td>Db::get_open_flags()</td>
<td>Returns the flags specified to Db::open</td>
</tr>
<tr>
<td>Db::get_slices()</td>
<td>Returns an array of slice databases.</td>
</tr>
<tr>
<td>Db::get_type()</td>
<td>Return the database type</td>
</tr>
<tr>
<td>Db::join()</td>
<td>Perform a database join on cursors</td>
</tr>
<tr>
<td>Db::key_range()</td>
<td>Return estimate of key location</td>
</tr>
<tr>
<td>Db::msg()</td>
<td>Informational message</td>
</tr>
<tr>
<td>Db::open()</td>
<td>Open a database</td>
</tr>
<tr>
<td>Db::put()</td>
<td>Store items into a database</td>
</tr>
<tr>
<td>Db::remove()</td>
<td>Remove a database</td>
</tr>
<tr>
<td>Db::rename()</td>
<td>Rename a database</td>
</tr>
<tr>
<td>Db::set_priority(), Db::get_priority()</td>
<td>Set/get cache page priority</td>
</tr>
<tr>
<td>Db::slice_lookup()</td>
<td>Return the slice database that contains the key</td>
</tr>
<tr>
<td>Db::stat()</td>
<td>Database statistics</td>
</tr>
<tr>
<td>Db::stat_print()</td>
<td>Display database statistics</td>
</tr>
<tr>
<td>Db::sync()</td>
<td>Flush a database to stable storage</td>
</tr>
<tr>
<td>Db::truncate()</td>
<td>Empty a database</td>
</tr>
<tr>
<td>Db::upgrade()</td>
<td>Upgrade a database</td>
</tr>
<tr>
<td>Db::verify()</td>
<td>Verify/salvage a database</td>
</tr>
</tbody>
</table>
### Database Operations

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Db::cursor()</strong></td>
</tr>
<tr>
<td>Create a cursor handle</td>
</tr>
</tbody>
</table>

### Database Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Db::get_partition_callback()</strong></td>
<td>Return the database partition callback</td>
</tr>
<tr>
<td><strong>Db::get_partition_keys()</strong></td>
<td>Returns the array of keys used for the database partition</td>
</tr>
<tr>
<td><strong>Db::set_alloc()</strong></td>
<td>Set local space allocation functions</td>
</tr>
<tr>
<td><strong>Db::set_cachesize(), Db::get_cachesize()</strong></td>
<td>Set/get the database cache size</td>
</tr>
<tr>
<td><strong>Db::set_create_dir(), Db::get_create_dir()</strong></td>
<td>Set/get the directory in which a database is placed</td>
</tr>
<tr>
<td><strong>Db::set_dup_compare()</strong></td>
<td>Set a duplicate comparison function</td>
</tr>
<tr>
<td><strong>Db::set_encrypt(), Db::get_encrypt_flags()</strong></td>
<td>Set/get the database cryptographic key</td>
</tr>
<tr>
<td><strong>Db::set_errcall()</strong></td>
<td>Set error message callback</td>
</tr>
<tr>
<td><strong>Db::set_errfile(), Db::get_errfile()</strong></td>
<td>Set/get error message FILE</td>
</tr>
<tr>
<td><strong>Db::set_error_stream()</strong></td>
<td>Set C++ ostream used for error messages</td>
</tr>
<tr>
<td><strong>Db::set_errpfx(), Db::get_errpfx()</strong></td>
<td>Set/get error message prefix</td>
</tr>
<tr>
<td><strong>Db::set_feedback()</strong></td>
<td>Set feedback callback</td>
</tr>
<tr>
<td><strong>Db::set_flags(), Db::get_flags()</strong></td>
<td>Set/get general database configuration</td>
</tr>
<tr>
<td><strong>Db::set_lk_exclusive(), Db::get_lk_exclusive()</strong></td>
<td>Set/get exclusive database locking</td>
</tr>
<tr>
<td><strong>Db::set_lorder(), Db::get_lorder()</strong></td>
<td>Set/get the database byte order</td>
</tr>
<tr>
<td><strong>Db::set_message_stream()</strong></td>
<td>Set C++ ostream used for informational messages</td>
</tr>
<tr>
<td><strong>Db::set_msgcall()</strong></td>
<td>Set informational message callback</td>
</tr>
<tr>
<td><strong>Db::set_msgfile(), Db::get_msgfile()</strong></td>
<td>Set/get informational message FILE</td>
</tr>
<tr>
<td><strong>Db::set_msgpfx(), Db::get_msgpfx()</strong></td>
<td>Set/get informational message prefix</td>
</tr>
<tr>
<td><strong>Db::set_pagesize(), Db::get_pagesize()</strong></td>
<td>Set/get the underlying database page size</td>
</tr>
<tr>
<td><strong>Db::set_partition()</strong></td>
<td>Set database partitioning</td>
</tr>
<tr>
<td><strong>Db::set_partition_dirs(), Db::get_partition_dirs()</strong></td>
<td>Set/get the directories used for database partitions</td>
</tr>
<tr>
<td><strong>Db::set_slice_callback()</strong></td>
<td>Set the callback used to identify slices</td>
</tr>
</tbody>
</table>

### Btree/Recno Configuration

<table>
<thead>
<tr>
<th>Description</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Db::set_append_recno()</strong></td>
<td>Set record append callback</td>
</tr>
<tr>
<td><strong>Db::set_bt_compare()</strong></td>
<td>Set a Btree comparison function</td>
</tr>
<tr>
<td><strong>Db::set_bt_compress()</strong></td>
<td>Set Btree compression functions</td>
</tr>
<tr>
<td><strong>Db::set_bt_minkey(), Db::get_bt_minkey()</strong></td>
<td>Set/get the minimum number of keys per Btree page</td>
</tr>
<tr>
<td><strong>Db::set_bt_prefix()</strong></td>
<td>Set a Btree prefix comparison function</td>
</tr>
<tr>
<td>Database Operations</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>Db::set_re_delim(), Db::get_re_delim()</td>
<td>Set/get the variable-length record delimiter</td>
</tr>
<tr>
<td>Db::set_re_len(), Db::get_re_len()</td>
<td>Set/get the fixed-length record length</td>
</tr>
<tr>
<td>Db::set_re_pad(), Db::get_re_pad()</td>
<td>Set/get the fixed-length record pad byte</td>
</tr>
<tr>
<td>Db::set_re_source(), Db::get_re_source()</td>
<td>Set/get the backing Recno text file</td>
</tr>
<tr>
<td><strong>Hash Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>Db::set_h_compare()</td>
<td>Set a Hash comparison function</td>
</tr>
<tr>
<td>Db::set_h_ffactor(), Db::get_h_ffactor()</td>
<td>Set/get the Hash table density</td>
</tr>
<tr>
<td>Db::set_h_hash()</td>
<td>Set a hashing function</td>
</tr>
<tr>
<td>Db::set_h_nelem(), Db::get_h_nelem()</td>
<td>Set/get the Hash table size</td>
</tr>
<tr>
<td><strong>Queue Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>Db::set_q_extentsize(),</td>
<td>Set/get Queue database extent size</td>
</tr>
<tr>
<td>Db::get_q_extentsize()</td>
<td></td>
</tr>
<tr>
<td><strong>Heap</strong></td>
<td></td>
</tr>
<tr>
<td>Db::set_heapsize(), Db::get_heapsize()</td>
<td>Set/get the database heap size</td>
</tr>
<tr>
<td>Db::set_heap_regionsize(),</td>
<td>Set/get the database region size</td>
</tr>
<tr>
<td>Db::get_heap_regionsize()</td>
<td></td>
</tr>
<tr>
<td><strong>DbHeapRecordId</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Database Utilities</strong></td>
<td></td>
</tr>
<tr>
<td>db_copy</td>
<td>Copy a named database to a target directory</td>
</tr>
</tbody>
</table>
Db::associate()

```cpp
#include <db_cxx.h>

int Db::associate(DbTxn *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.

When called on a database that has been opened with sliced support, the Db::associate() method is automatically called on each supporting slice database.

The Db::associate() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**txnid**

If the operation is part of an application-specified transaction, the txnid parameter is a transaction handle returned from DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException (page 369)` (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

`DbDeadlockException (page 365)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.

**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()

```c++
#include <db_cxx.h>

int
DB::associate_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 can not be used with a sliced database.

The Db::associate_foreign() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw
exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**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; if this method is called on a sliced database.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::close()

#include <db_cxx.h>

int Db::close(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 Dbc 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 DbTxn 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 166) 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 called on a database that has been opened with sliced support, the Db::close() method is automatically called on each supporting slice database.

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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success. The error values that Db::close() method returns include the error values of Dbc::close() and the following:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.
DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

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 throw a DbException exception, encapsulating one of the following non-zero errors, or 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:
DbDeadlockException or DB_LOCK_DEADLOCK

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

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::compact()

```c
#include <db_cxx.h>

int
Db::compact(DbTxn *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 is non-atomic when used on a sliced database. If the operation fails, it is possible for a subset of the supporting sliced databases to be compacted.

The Db::compact() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**txnid**

If the operation is part of an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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. The implementation does not guarantee an exact match to the number of pages requested.

- 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 filesystem. A repeated call to the Db::compact() method with a low compact_fillpercent may be used to return pages in this case.

dend

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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

DbDeadlockException or DB_LOCK DEADLOCK

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.
DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**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_copy

```c
#include <db.h>

int db_copy(DB_ENV *dbenv, const char *dbfile, const char *target,
             const char *password);
```

The `db_copy()` routine copies the named database file to the target directory. An optional password can be specified for encrypted database files. This routine can be used on operating systems that do not support atomic file system reads to create a hot backup of a database file. If the specified database file is for a QUEUE database with extents, all extent files for that database will be copied as well.

**Parameters**

**dbenv**
An open environment handle for the environment containing the database file.

**dbfile**
The path name to the file to be backed up. The file name is resolved using the usual BDB library name resolution rules.

**target**
The directory to which you want the database copied. This is specified relative to the current directory of the executing process or as an absolute path.

**password**
Specified only if the database file is encrypted. The resulting backup file will be encrypted as well.
The Db handle is the handle for a Berkeley DB database, which may or may not be part of a database environment.

Db handles are free-threaded if the DB_THREAD flag is specified to the Db::open() (page 75) 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 86), Db::rename() (page 88), or Db::verify() (page 172) methods are called, the handle may not be accessed again, regardless of the method's return.

The constructor creates a Db object that is the handle for a Berkeley DB database. The constructor allocates memory internally; calling the Db::close() (page 13), Db::remove() (page 86), or Db::rename() (page 88) methods will free that memory.

Note that destroying the Db object is synonymous with calling Db::close(0).

Each Db object has an associated DB struct, which is used by the underlying implementation of Berkeley DB and its C-language API. The Db::get_DB() method returns a pointer to this struct. Given a const Db object, Db::get_const_DB() returns a const pointer to the same struct.

Given a DB struct, the Db::get_DB() method returns the corresponding Db object, if there is one. If the DB object was not associated with a Db (that is, it was not returned from a call to the Db::get_DB() method), then the result of Db::get_DB() is undefined. Given a const DB struct, Db::get_const_Db() returns the associated const Db object, if there is one.

These methods may be useful for Berkeley DB applications including both C and C++ language software. It should not be necessary to use these calls in a purely C++ application.
Parameters

**dbenv**

If no `dbenv` value is specified, the database is standalone; that is, it is not part of any Berkeley DB environment.

If a `dbenv` value is specified, 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 must be set to 0 or the following value:

- **DB_CXX_NO_EXCEPTION**

  The Berkeley DB C++ API supports two different error behaviors. By default, whenever an error occurs, an exception is thrown that encapsulates the error information. This generally allows for cleaner logic for transaction processing because a try block can surround a single transaction. However, if this flag is specified, exceptions are not thrown; instead, each individual function returns an error code.

  If a `dbenv` value is specified, this flag is ignored, and the error behavior of the specified environment is used instead.

Class

**Db**

See Also

*Database and Related Methods (page 3)*
Db::del()

```c
#include <db_cxx.h>

int Db::del(DbTxn *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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

### Parameters

**txnid**

If the operation is part of an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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 DbMultipleDataBuilder (page 221). To delete records in bulk by record number, construct a bulk buffer in the key Dbt using DbMultipleRecnoDataBuilder (page 225) 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 23) for each one.

See the DBT and Bulk Operations (page 212) 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 DbMultipleKeyDataBuilder (page 223). To delete records in bulk with the recno or hash access methods, construct a bulk buffer in the key Dbt using DbMultipleRecnoDataBuilder (page 225).

See the DBT and Bulk Operations (page 212) 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 throw a DbException exception, encapsulating one of the following non-zero errors, or 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.

DbDeadlockException or DB_LOCK_DEADLOCK

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.
DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

DbDeadlockException or DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

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_cxx.h>

Db::err(int error, const char *fmt, ...);
Db::errx(const char *fmt, ...);

The DbEnv::err() (page 246), DbEnv::errx(), Db::err() and Db::errx() methods provide error-messaging functionality for applications written using the Berkeley DB library.

The Db::err() and DbEnv::err() (page 246) methods construct an error message consisting of the following elements:

- **An optional prefix string**
  
  If no error callback function has been set using the DbEnv::set_errcall() (page 315) method, any prefix string specified using the DbEnv::set_errpfx() (page 320) 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 DbEnv::strerror() (page 361) method.

The Db::errx() and DbEnv::errx() methods are the same as the Db::err() and DbEnv::err() (page 246) 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 109) and DbEnv::set_errcall() (page 315)), that function is called with two parameters: any prefix string specified (see Db::set_errpfx() (page 114) and DbEnv::set_errpfx() (page 320)) and the error message.

- If a C library FILE * has been set (see Db::set_errfile() (page 111) and DbEnv::set_errfile() (page 317)), the error message is written to that output stream.

- If a C++ ostream has been set (see DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113)), the error message is written to that stream.

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

The total length of the message written by this method, including the prefix string, can be no longer than 4096.

Parameters

error

The error parameter is the error value for which the `DbEnv::err()` (page 246) 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_cxx.h>

int Db::exists(DbTxn *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 DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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()

```cpp
#include <db_cxx.h>

int Db::fd(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 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_cxx.h>

int
Db::get(DbTxn *txnid, Dbt *key, Dbt *data, u_int32_t flags);

int
Db::pget(DbTxn *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 Dbc::get() (page 193) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**txnid**

If the operation is part of an application-specified transaction, the txnid parameter is a transaction handle returned from DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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.
If `DB_DBT_PARTIAL` is set for the Dbt used for this parameter, and if the `flags` parameter is not set to `DB_CONSUME DB_CONSUME_WAIT`, or `DB_SET_RECNO`, then this method will fail and return `EINVAL`.

**pkey**

The `pkey` parameter is the return key from the primary database. If `DB_DBT_PARTIAL` is set for the Dbt used for this parameter, then this method will fail and return `EINVAL`.

**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 return 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, Recno or Heap 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 212)** 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbMemoryException or DB_BUFFER_SMALL**

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

`DbMemoryException` (page 368) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_BUFFER_SMALL` is returned.

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

`DbLockNotGrantedException` (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**DbLockNotGrantedException or 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.

`DbLockNotGrantedException` (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DB_REPLEASE_EXPIRED**

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

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**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 75) 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()

```cpp
#include <db_cxx.h>

int Db::get_bt_minkey(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 99) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 99)
Db::get_byteswapped()  

```c++
#include <db_cxx.h>

int Db::get_byteswapped(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()` method is called.

The `Db::get_byteswapped()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called before `Db::open()` 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()

```cpp
#include <db_cxx.h>

int Db::get_cachesize(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 102) method.

The Db::get_cachesize() method may be called at any time during the life of the application.

The Db::get_cachesize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 102)
Db::get_create_dir()

```cpp
#include <db_cxx.h>

int
Db::get_create_dir(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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

dirp

The dirp will be set to the directory specified in the call to Db::set_create_dir() (page 104) method on this handle or to the directory that the database was found in after Db::open() (page 75) has been called.

Errors

The Db::get_create_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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()

```cpp
#include <db_cxx.h>

int Db::get_dbname(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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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()

```cpp
#include <db_cxx.h>

int Db::get_encrypt_flags(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 107) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 107)
Db::get_errfile()

```cpp
#include <db_cxx.h>

void Db::get_errfile(FILE **errfilep);
```

The `Db::get_errfile()` method returns the `FILE *`, as set by the `Db::set_errfile()` (page 111) 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 111)
Db::get_errpfx()

#include <db_cxx.h>

void Db::get_errpfx(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 114)
Db::get_flags()

```cpp
#include <db_cxx.h>

int Db::get_flags(u_int32_t *flagsp);
```

The `Db::get_flags()` method returns the current database flags as set by the `Db::set_flags()` (page 117) method.

The `Db::get_flags()` method may be called at any time during the life of the application.

The `Db::get_flags()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 117)
Db::get_h_ffactor()

```cpp
#include <db_cxx.h>

int Db::get_h_ffactor(u_int32_t *h_ffactorp);
```

The `Db::get_h_ffactor()` method returns the hash table density as set by the `Db::set_h_ffactor()` 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 125)
Db::get_h_nelem()

```c
#include <db_cxx.h>

int Db::get_h_nelem(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 127) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 127)*
Db::get_heapsize()

```cpp
#include <db_cxx.h>

int Db::get_heapsize(u_int32_t *gbytesp, u_int32_t *bytesp);
```

Used when the underlying database is configured to use the Heap access method. This method returns the maximum size of the database's heap file. This value may be set using the Db::set_heapsize() (page 128) method.

The Db::get_heapsize() method may be called at any time during the life of the application.

The Db::get_heapsize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**gbytesp**

The gbytesp parameter references memory into which is copied the maximum number of gigabytes in the heap.

**bytesp**

The bytesp parameter references memory into which is copied the additional bytes in the heap.

**Class**

Db

**See Also**

Database and Related Methods (page 3), Db::set_heapsize() (page 128)
Db::get_heap_regionsize()

```
#include <db_cxx.h>

int
Db::get_heap_regionsize(u_int32_t *npagesp);
```

Used when the underlying database is configured to use the Heap access method. This method returns the number of pages in a region. This value may be set using the Db::set_heap_regionsize() (page 130) method.

The Db::get_heap_regionsize() method may be called at any time during the life of the application.

The Db::get_heap_regionsize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

* npagesp

The npagesp parameter references memory into which is copied the number of pages in a region.

**Class**

Db

**See Also**

Database and Related Methods (page 3), Db::set_heap_regionsize() (page 130)
Db::get_lk_exclusive()

#include <db_cxx.h>

int
Db::get_lk_exclusive(int *onoff, int *nowait);

Returns whether the database handle is configured to obtain a write lock on the entire
database. This can be set using the Db::set_lk_exclusive() (page 131) method.

The Db::get_lk_exclusive() method may be called at any time during the life of the
application.

The Db::get_lk_exclusive() always returns 0.

Parameters

onoff

Indicates whether the handle is configured for exclusive database locking. If 0, it is not
configured for exclusive locking. If 1, then it is configured for exclusive locking.

nowait

Indicates whether the handle is configured for immediate locking. If 0, then the locking
operation will block until it can obtain an exclusive database lock. If 1, then the locking
operation will error out if it cannot immediately obtain an exclusive lock.

Class

Db

See Also

Database and Related Methods (page 3), Db::set_lk_exclusive() (page 131)
Db::get_lorder()  

```c++
#include <db_cxx.h>

int Db::get_lorder(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 133) method.

The Db::get_lorder() method may be called at any time during the life of the application.

The Db::get_lorder() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 133)
Db::get_msgfile()

#include <db_cxx.h>

void Db::get_msgfile(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 137) 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 137)
Db::get_msgpfx()

```cpp
#include <db_cxx.h>

void Db::get_msgpfx(const char **msgpfxp);
```

The `Db::get_msgpfx()` method returns the informational prefix.

The `Db::get_msgpfx()` method may be called at any time during the life of the application.

**Parameters**

- `msgpfxp`
  
  The `Db::get_msgpfx()` method returns a reference to the error prefix in `msgpfxp`.

**Class**

- `Db`

**See Also**

- Database and Related Methods (page 3), `Db::set_msgpfx()` (page 138)
Db::get_multiple()

```cpp
#include <db_cxx.h>

int Db::get_multiple()
```

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 75) method is called.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::get_open_flags()

```cpp
#include <db_cxx.h>

int Db::get_open_flags(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 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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()

```cpp
#include <db_cxx.h>

int Db::get_partition_callback(u_int32_t *partsp,
    u_int32_t (**callback_fcn) (DB *dbp, DBT *key);
```

The Db::get_partition_callback() method returns the database partitioning callback as set by the Db::set_partition() (page 140) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **partsp**
  The partsp parameter returns the number of partitions used by the database.

- **callback_fcn**
  The callback_fcn parameter returns the partitioning callback.

**Class**

Db

**See Also**

Database and Related Methods (page 3), Db::set_partition() (page 140)
Db::get_partition_dirs()

```
#include <db_cxx.h>

int
Db::get_partition_dirs(const char **dirsp);
```

Identify the directories used to store the database partitions.

The Db::get_partition_dirs() method may be called at any time.

The Db::get_partition_dirs() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

dirsp

The dirsp will be set to the array of directories specified in the call to Db::set_partition_dirs() method on this handle or to the directoreies that the database partitions were found in after Db::open() has been called.

**Errors**

The Db::get_partition_dirs() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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()

```cpp
#include <db_cxx.h>

int Db::get_partition_keys(u_int32_t *partsp, DBT *keysp);
```

The `Db::get_partition_keys()` method returns the range of keys used to specify the values placed in each of a database's partitions. This information is set using the `Db::set_partition()` (page 140) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

* `partsp`
  The `partsp` parameter returns the number of partitions in the database.

* `keysp`
  The `keysp` parameter returns the set of keys used to place values in the database partitions.

**Class**

`Db`

**See Also**

`Database and Related Methods` (page 3), `Db::set_partition()` (page 140)
Db::get_pagesize()

```
#include <db_cxx.h>

int Db::get_pagesize(u_int32_t *pagesizep);
```

The `Db::get_pagesize()` method returns the database's current page size, as set by the `Db::set_pagesize()` (page 139) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 139)
Db::get_priority()

```cpp
#include <db_cxx.h>

int Db::get_priority(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 143) method.

The `Db::get_priority()` method may be called only after the database has been opened.

The `Db::get_priority()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`priorityp`

The `Db::get_priority()` method returns a reference to the cache priority in `priorityp`. See `Db::set_priority()` (page 143) for a list of possible priorities.

**Class**

Db

**See Also**

Database and Related Methods (page 3), `Db::set_priority()` (page 143)
Db::get_q_extentsize()\

```c++
#include <db_cxx.h>

int Db::get_q_extentsize(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()` method.

The `Db::get_q_extentsize()` method may be called only after the database has been opened.

The `Db::get_q_extentsize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 145)
Db::get_re_delim()

```
#include <db_cxx.h>

int Db::get_re_delim(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 146) method.

The Db::get_re_delim() method may be called only after the database has been opened.

The Db::get_re_delim() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 146) for details.

**Class**

Db

**See Also**

Database and Related Methods (page 3), Db::set_re_delim() (page 146)
Db::get_re_len()

```
#include <db_cxx.h>

int Db::get_re_len(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 147) method.

The `Db::get_re_len()` method may be called only after the database has been opened.

The `Db::get_re_len()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 147), then 0 is returned.

**Class**

`Db`

**See Also**

Database and Related Methods (page 3), `Db::set_re_len()` (page 147)
Db::get_re_pad()

#include <db_cxx.h>

int
Db::get_re_pad(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 149) method.

The Db::get_re_pad() method may be called only after the database has been opened.

The Db::get_re_pad() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 149) method description for what that default value is.

Class

Db

See Also

Database and Related Methods (page 3), Db::set_re_pad() (page 149)
Db::get_re_source()

#include <db_cxx.h>

int Db::get_re_source(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 150) method.

The Db::get_re_source() method may be called only after the database has been opened.

The Db::get_re_source() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 150)
# Db::get_slices()

```cpp
#include <db_cxx.h>

int Db::get_slices(Db ***slicepp);
```

The `Db::get_slices()` method returns a NULL-terminated array of slice databases. Each element in the array represents one slice. Note that you should not explicitly close these database handles; each slice database is automatically closed when the parent database handle is closed.

The `Db::get_slices()` method may be called at any time during the life of the application.

The `Db::get_slices()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

slicepp

The `Db::get_slices()` method returns a reference to a NULL-terminated array of slice databases in `slicepp`.

**Class**

`Db`

**See Also**

Database and Related Methods (page 3)
Db::get_type()

```cpp
#include <db_cxx.h>

int Db::get_type(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 75) with the `type` parameter set to DB_UNKNOWN.

The `Db::get_type()` method may not be called before the `Db::open()` (page 75) method is called.

The `Db::get_type()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called before `Db::open()` (page 75) 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_cxx.h>

int Db::join(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 Dbc::get() (page 193) and Dbc::close() (page 182) cursor functions:

- **Dbc::get() (page 193)**

  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.

  - **Dbc::close() (page 182)**

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

The Db::join() method can not be used with a sliced database.
The `Db::join()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `Dbc::get()` (page 193) 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException` (page 369) (if your application is configured to throw
exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**DB_SECONDARY_BAD**

A secondary index references a nonexistent primary key.

**EINVAL**

If cursor methods other than Dbc::get() (page 193) or Dbc::close() (page 182) were called; if this method is called on a sliced database; or if an invalid flag value or parameter was specified.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::key_range()

```c
#include <db_cxx.h>

int Db::key_range(DbTxn *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 can not be used with a sliced database.

The Db::key_range() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**txnid**
If the operation is part of an application-specified transaction, the txnid parameter is a transaction handle returned from DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

DbDeadlockException or DB_LOCK_DEADLOCK

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

DbDeadlockException or DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.
EINVAL

If the underlying database was not of type Btree; if this method is called on a sliced database; or if an invalid flag value or parameter was specified.

Class

Db

See Also

Database and Related Methods (page 3)
Db::msg()  

```c++
#include <db_cxx.h>
Db::msg(const char *fmt, ...);
```

The `DbEnv::msg()` (page 284) and `Db::msg()` methods provide informational messaging functionality for applications written using the Berkeley DB library.

The `Db::msg()` and `DbEnv::msg()` (page 284) methods construct an informational message consisting of the following elements:

- **An optional prefix string**
  
  If no informational callback function has been set using the `DbEnv::set_msgcall()` (page 340) method, any prefix string specified using the `DbEnv::set_msgpfx()` (page 343) method.

- **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.

This constructed informational message is then handled as follows:

- If an informational callback function has been set (see `Db::set_msgcall()` (page 135) and `DbEnv::set_msgcall()` (page 340)), that function is called with two parameters: any prefix string specified (see `Db::set_msgpfx()` (page 138) and `DbEnv::set_msgpfx()` (page 343)) and the informational message.

- If a C library FILE * has been set (see `Db::set_msgfile()` (page 137) and `DbEnv::set_msgfile()` (page 342)), the informational message is written to that output stream.

- If a C++ ostream has been set (see `DbEnv::set_message_stream()` (page 339) and `Db::set_message_stream()` (page 134)), the informational message is written to that stream.

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

**Note**

The total length of the message written by this method, including the prefix string, can be no longer than 4096.

**Parameters**

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

Db

**See Also**

Database and Related Methods (page 3)
Db::open()

```c
#include <db_cxx.h>

int Db::open(DbTxn *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, Heap, 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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.
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 or Heap 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_HEAP, 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 66) 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 DbEnv::set_mp_mmapsiz() (page 487) 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_SLICED**

  Open the database with slice support. If the set_slice_count (page 807) DB_CONFIG parameter is not set, this flag is ignored. In addition, slice support must be configured for the database at compile time, or this flag is ignored.

  If this flag is specified, then the **file** parameter must not be NULL and the **database** parameter must be NULL. Also, the **type** parameter must be either DB_BTREE or DB_HASH, or DB_UNKNOWN when opening an existing database.

  This flag is optional when opening an existing database.

  This flag can not be used if the supporting environment was opened with the DB_INIT_REP and DB_FAILCHK flags. Also, distributed transactions can not be used with sliced databases.

- **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. You should use this flag only in the absence of an encompassing environment.

  When opening the database within an encompassing environment, the database inherits the state of this flag from the environment. That is, if the encompassing environment is threaded, then the database will also be threaded. Note that it is an error to specify this flag to the database open if the encompassing environment is not threaded.

  Note that this flag is incompatible with the Db::set_lk_exclusive() method.

  Be aware that enabling this flag will serialize calls to DB when using the handle across threads. If concurrent scaling is important to your application we recommend opening separate handles for each thread (and not specifying this flag), rather than sharing handles between threads.
• **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 `DbEnv::add_data_dir()` (page 230) method, or by setting the "add_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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

`DbDeadlockException (page 365)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.
You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

`DbLockNotGrantedException (page 366)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**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.

**DB_META_CHKSUM_FAIL**
Checksum mismatch detected on a database metadata page. Either the database is corrupted or the file is not a Berkeley DB database file.

**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 `DbEnv::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; a Heap database is in use and `Db::set_heapsize()` (page 128) was used to set a heap size that is different from the value used to create the database or an invalid heap region size was set using `Db::set_heap_regionsize()` (page 130); or if an invalid flag value or parameter was specified.

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException` (page 369) (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**
The operation was blocked by client/master synchronization.
DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
# Db::put()

```cpp
#include <db_cxx.h>

int Db::put(DbTxn *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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

## Parameters

**txnid**

If the operation is part of an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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.

If creating a new record in a Heap database, the key `Dbt` must be empty. The `put` method will return the new record’s Record ID (RID) in the key `Dbt`.

**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 Heap, 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.

For a Heap database, if the put operation results in the creation of a new record, then this flag is required.

- **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 204) 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 204) 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 DbMultipleDataBuilder (page 221). 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 DbMultipleRecnoDataBuilder (page 225) 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 81) for each one.

  See DBT and Bulk Operations (page 212) 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 bulk buffer in the key Dbt using DbMultipleKeyDataBuilder (page 223). To put records in bulk with the recno or queue access methods, construct a bulk buffer in the key Dbt using DbMultipleRecnoDataBuilder (page 225).

See DBT and Bulk Operations (page 212) 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 throw a DbException exception, encapsulating one of the following non-zero errors, or 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_HEAP_FULL**

An attempt was made to add or update a record in a Heap database. However, the size of the database was constrained using the `Db::set_heapsize()` (page 128) method, and that limit has been reached.

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**EACCESS**

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 on a database not configured for it (such as a database configured for duplicate records); 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()

```cpp
#include <db_cxx.h>

int Db::remove(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 `DbEnv::dbremove()` (page 241) method should be used instead.

The `Db::remove()` method can not be called for sliced databases; the `DbEnv::dbremove()` (page 241) method should be used instead.

The `Db::remove()` method may not be called after calling the `Db::open()` (page 75) method on any Db handle. If the `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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
DbEnv::add_data_dir() (page 230) method, or by setting the “add_data_dir” string in the
environment’s DB_CONFIG file.

Errors

The Db::remove() method may fail and throw a DbException exception, encapsulating one of
the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() (page 75) was called; if this method was called on a
sliced database; or if an invalid flag value or parameter was specified.

**ENOENT**

The file or directory does not exist.

**DB_META_CHKSUM_FAIL**

Checksum mismatch detected on a database metadata page. Either the database is corrupted
or the file is not a Berkeley DB database file.

Class

Db

See Also

Database and Related Methods (page 3)
Db::rename()

```cpp
#include <db_cxx.h>

int
Db::rename(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 `DbEnv::dbrename()` (page 243) method should be used instead.

The `Db::rename()` method can not be used for sliced databases; the `DbEnv::dbrename()` (page 243) method should be used instead.

The `Db::rename()` method may not be called after calling the `Db::open()` (page 75) method on any `Db` handle. If the `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::add_data_dir() (page 230) method, or by setting the “add_data_dir” string in the environment’s DB_CONFIG file.

Errors

The Db::rename() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL
If the method was called after Db::open() (page 75) was called; if this method was called on a sliced database; or if an invalid flag value or parameter was specified.

ENOENT
The file or directory does not exist.

DB_META_CHKSUM_FAIL
Checksum mismatch detected on a database metadata page. Either the database is corrupted or the file is not a Berkeley DB database file.

Class

Db

See Also

Database and Related Methods (page 3)
Db::set_alloc()

#include <db_cxx.h>

int Db::set_alloc(db_malloc_fcn_type app_malloc,
                 db_realloc_fcn_type app_realloc,
                 db_free_fcn_type app_free);

Set the allocation functions used by the DbEnv 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 157), DbEnv::lock_stat() (page 404), DbEnv::log_archive() (page 423), DbEnv::log_stat() (page 437), DbEnv::memp_stat() (page 471), and DbEnv::txn_stat() (page 680). 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 DbEnv::set_alloc() 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 75) method is called.

The Db::set_alloc() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Errors**

The Db::set_alloc() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If called in a database environment, or called after Db::open() (page 75) 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()

```c
#include <db_cxx.h>

int
Db::set_append_recno(int (*db_append_recno_fcn)(DB *dbp, Dbt *data,
                      db_recno_t recno));
```

When using the DB_APPEND option of the Db::put() (page 81) 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 75) method is called.

The Db::set_append_recno() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 \texttt{errno} or a value outside of the Berkeley DB error name space on failure.

**Errors**

The \texttt{Db::set_append_recno() } method may fail and throw a \texttt{DbException} exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after \texttt{Db::open()} (page 75) was called; or if an invalid flag value or parameter was specified.

**Class**

\textbf{Db}

**See Also**

Database and Related Methods (page 3)
Db::set_bt_compare()

```cpp
#include <db_cxx.h>
extern "C" {
    typedef int (*bt_compare_fcn_type)(DB *db, const DBT *dbt1,
                                       const DBT *dbt2, size_t *locp);
};
int Db::set_bt_compare(bt_compare_fcn_type bt_compare_fcn);
```

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 75) method is called. If the database already exists when Db::open() (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

- **bt_compare_fcn**

  The bt_compare_fcn function is the application-specified Btree comparison function. The comparison function takes four 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.

  - **locp**

    The locp parameter is currently unused, and must be set to NULL or corruption can occur.
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 100) 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` (page 75) 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()**

```cpp
#include <db_cxx.h>

extern "C" {
    typedef int (*bt_compress_fcn_type)(DB *db, const DBT *prevKey,
        const DBT *prevData, const DBT *key, const DBT *data, DBT *dest);
    typedef int (*bt_decompress_fcn_typ)(DB *db, const DBT *prevKey,
        const DBT *prevData, DBT *compressed, DBT *destKey,
        DBT *destData);
};

int
Db::set_bt_compress(bt_compress_fcn_type bt_compress_fcn,
    bt_decompress_fcn_type bt_decompress_fcn);
```

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.

This method is only compatible with prefix-based compression routines. This callback is mostly intended for compressing keys. From a performance perspective, it is better to perform compression of the data portion of your records outside of the Berkeley DB library.

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 75) method is called. If the database already exists when `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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->set_data()` (the size of which is returned by `dest->get_ulen()`), the function should identify the required buffer size in `dest->set_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->set_data() or data->set_data() (the size of which is returned by the Dbt's get_ulen() method), the function should identify the required buffer size using the Dbt's set_size() method and return DB_BUFFER_SMALL.

Errors

The Db::set_bt_compress() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after Db::open() (page 75) 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()**

```cpp
#include <db_cxx.h>

int
Db::set_bt_minkey(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 75) method is called. If the database already exists when `Db::open()` (page 75) is called, the information specified to `Db::set_bt_minkey()` will be ignored.

The `Db::set_bt_minkey()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after `Db::open()` (page 75) 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()

```cpp
#include <db_cxx.h>
extern "C" {
    typedef size_t (*bt_prefix_fcn_type)(DB *, const DBT *dbt1,
                                        const DBT *dbt2);
};

int Db::set_bt_prefix(bt_prefix_fcn_type bt_prefix_fcn);
```

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 75) method is called. If the database already exists when Db::open() (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` (page 75) 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_cxx.h>

int Db::set_cachesize(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 75) method is called.

The Db::set_cachesize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or 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 75) 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()

```cpp
#include <db_cxx.h>

int Db::set_create_dir(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 75) method is called.

The `Db::set_create_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`dir`

The `dir` will be used to create or locate the database file specified in the `Db::open()` (page 75) method call. The directory must be one of the directories in the environment list specified by `DbEnv::add_data_dir()` (page 230).

**Errors**

The `Db::set_create_dir()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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()

```cpp
#include <db_cxx.h>

extern "C" {
    typedef int (*dup_compare_fcn_type)(DB *db, const DBT *dbt1,
                                       const DBT *dbt2, size_t *locp);
};

int Db::set_dup_compare(dup_compare_fcn_type dup_compare_fcn);
```

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 117) 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 75) method is called. If the database already exists when `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

### Parameters

**dup_compare_fcn**

The `dup_compare_fcn` function is the application-specified duplicate data item comparison function. The function takes four 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.

- **locp**
  - The `locp` parameter is currently unused, and must be set to NULL or corruption can occur.

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 throw a DbException exception, encapsulating one of the following non-zero errors, or 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()

```cpp
#include <db_cxx.h>

int Db::set_encrypt(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 75) method is called.

The Db::set_encrypt() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  If the method was called after Db::open() (page 75) 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()

```cpp
#include <db_cxx.h>

void Db::set_errcall(void (*db_errcall_fcn)(const DbEnv *dbenv, const char *errpfx, const char *msg));
```

When an error occurs in the Berkeley DB library, an exception is thrown 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 `DbEnv::set_errcall()` (page 315) 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 `DbEnv::set_error_stream()` (page 319) and `Db::set_error_stream()` (page 113) methods to display the additional information via an output stream, or the `Db::set_errfile()` (page 111) or `Db::set_errfile()` (page 317) methods to display the additional information via a C library `FILE *`. You should not mix these approaches.

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 `DbEnv::set_errcall()` (page 315) 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 114) or DbEnv::set_errpfx() (page 320)).

• msg

The msg parameter is the error message string.

Class

Db

See Also

Database and Related Methods (page 3)
Db::set_errfile()

# include <db_cxx.h>

void Db::set_errfile(FILE *errfile);

When an error occurs in the Berkeley DB library, an exception is thrown 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 DbEnv::set_errfile() (page 317) 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 DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113) methods to display the additional messages via an output stream, or the DbEnv::set_errcall() (page 315) or Db::set_errcall() (page 109) methods to capture the additional error information in a way that does not use C library FILE *'s. You should not mix these approaches.

The error message will consist of the prefix string and a colon ("":") (if a prefix string was previously specified using Db::set_errpfx() (page 114) or DbEnv::set_errpfx() (page 320)), an error string, and a trailing <newline> character.

The default configuration when applications first create Db or DbEnv handles is as if the DbEnv::set_errfile() (page 317) 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 DbEnv::set_errfile() (page 317) 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()
- DbEnv::set_errfile() (page 317)
- DbEnv::set_errcall() (page 315)
- Db::set_errcall() (page 109)
- DbEnv::set_error_stream() (page 319)
- Db::set_error_stream() (page 113)

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 DbEnv::set_errfile() (page 317) 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_error_stream()

```cpp
#include <db_cxx.h>

void Db::set_error_stream(class ostream*);
```

When an error occurs in the Berkeley DB library, an exception is thrown or an `errno` 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 `DbEnv::set_error_stream()` (page 319) and `Db::set_error_stream()` methods are used to enhance the mechanism for reporting error messages to the application by setting the C++ `ostream` 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 stream.

The error message will consist of the prefix string and a colon (":") (if a prefix string was previously specified using `Db::set_errpfx()` (page 114), an error string, and a trailing `<newline>` character.

Setting `stream` to NULL unconfigures the interface.

Alternatively, you can use the `DbEnv::set_errfile()` (page 317) or `Db::set_errfile()` (page 111) methods to display the additional information via a C Library `FILE *`, or the `DbEnv::set_errcall()` (page 315) and `Db::set_errcall()` (page 109) methods to capture the additional error information in a way that does not use either output streams or C Library `FILE *`s. You should not mix these approaches.

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_error_stream()` method affects the entire environment and is equivalent to calling the `DbEnv::set_error_stream()` (page 319) method.

The `Db::set_error_stream()` method may be called at any time during the life of the application.

**Parameters**

- `stream`

  The `stream` parameter is the application-specified output stream to be used for additional error information.

**Class**

- `Db`

**See Also**

- Database and Related Methods (page 3)
**Db::set_errpfx()**

```cpp
#include <db_cxx.h>

void Db::set_errpfx(const char *errpfx);
```

Set the prefix string that appears before error messages issued by Berkeley DB. The length of this string can be no longer than 200.

The `Db::set_errpfx()` and `DbEnv::set_errpfx()` (page 320) 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 `DbEnv::set_errpfx()` (page 320) 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()

```cpp
#include <db_cxx.h>

int Db::set_feedback(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 progress percentage.
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()

```cpp
#include <db_cxx.h>

int Db::set_flags(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 75) method is called.

The `Db::set_flags()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 75) is called, the DB_CHKSUM flag will be ignored.

- **DB_ENCRYPT**

  Encrypt the database using the cryptographic password specified to the `DbEnv::set_encrypt()` (page 307) or `Db::set_encrypt()` (page 107) 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 75) 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 DbEnv::txn_checkpoint() (page 678) or DbEnv::memp_sync() (page 478) 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 75) 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 105) 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 75) 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 31) and `Dbc::get()` (page 193) 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 75) 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 75) is called, the DB_DUP flag must be the same as the existing database or an error will be returned.

- 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 duplicate comparison function. If the application does not specify a comparison function using the Db::set_dup_compare() (page 105) method, a default lexical comparison will be used.

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 75) 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 105) 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 75) 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 31) 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 31) call returns.

This flag modifies the Db::get() (page 31) 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_f1ags()` 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 81) or `Dbc::put()` (page 202) 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 `Dbc::get()` (page 193) 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 `Dbc::get()` (page 193) 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_f1ags()` 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 75) 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 75) is called. If this flag is not specified, the `re_source` file may be read
lazily.

See the `Db::set_re_source()` (page 150) 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 throw a `DbException` exception, encapsulating
one of the following non-zero errors, or 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_cxx.h>

extern "C" {
    typedef int (*compare_fcn_type)(DB *db, const DBT *dbt1,
        const DBT *dbt2, size_t *locp);
};

int Db::set_h_compare(compare_fcn_type compare_fcn);
```

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 75) method is called. If the database already exists when `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **compare_fcn**

  The `compare_fcn` function is the application-specified Hash comparison function. The comparison function takes four 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.

  - **locp**
    
    The `locp` parameter is currently unused, and must be set to NULL or corruption can occur.

  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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() (page 75) 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()

```cpp
#include <db_cxx.h>

int Db::set_h_ffactor(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:

\[
\frac{(\text{pagesize} - 32)}{\text{(average_key_size + average_data_size + 8)}}
\]

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 75) method is called. If the database already exists when Db::open() (page 75) is called, the information specified to Db::set_h_ffactor() will be ignored.

The Db::set_h_ffactor() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() (page 75) 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_cxx.h>

extern "C" {
    typedef u_int32_t (*h_hash_fcn_type)(DB *, const void *bytes, u_int32_t length);
};

int Db::set_h_hash(h_hash_fcn_type h_hash_fcn);
```

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 75) method is called. If the database already exists when `Db::open()` (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` (page 75) 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_cxx.h>

int Db::set_h_nelem(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 125) 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 75) method
is called. If the database already exists when Db::open() (page 75) is called, the information
specified to Db::set_h_nelem() will be ignored.

The Db::set_h_nelem() method either returns a non-zero error value or throws an exception
that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating
one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after Db::open() (page 75) was called; or if an invalid flag value or
parameter was specified.

Class

Db

See Also

Database and Related Methods (page 3)
Db::set_heapsize()

#include <db_cxx.h>

int Db::set_heapsize(u_int32_t gbytes, u_int32_t bytes, u_int32_t flags);

Sets the maximum on-disk database file size used by a database configured to use the Heap access method. If this method is never called, the database's file size can grow without bound. If this method is called, then the heap file can never grow larger than the limit defined by this method. In that case, attempts to update or create records in a Heap database that has reached its maximum size will result in a DB_HEAP_FULL error return.

The size specified to this method must be at least three times the database page size. That is, a Heap database must contain at least three database pages. You can set the database page size using the Db::set_pagesize() (page 139) method.

The Db::set_heapsize() method may not be called after the Db::open() (page 75) method is called. Further, if this method is called on an existing Heap database, the size specified here must match the size used to create the database. Note, however, that specifying an incorrect size to this method will not result in an error return (EINVAL) until the database is opened.

The Db::set_heapsize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

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

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

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

Errors

The Db::set_heapsize() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL
If the specified heap size was too small; the method was called after Db::open() (page 75) was called; or if an invalid flag value or parameter was specified.

Class

Db
See Also

Database and Related Methods (page 3)
**Db::set_heap_regionsize()**

```cpp
#include <db_cxx.h>

int
Db::set_heap_regionsize(u_int32_t npages);
```

Sets the number of pages in a region of a database configured to use the Heap access method. If this method is never called, the default region size for the database's page size will be used. You can set the database page size using the `Db::set_pagesize()` (page 139) method.

The `Db::set_heap_regionsize()` method may not be called after the `Db::open()` (page 75) method is called. If the database already exists when `Db::open()` (page 75) is called, the information specified to `Db::set_heap_regionsize()` will be ignored. If the specified region size is larger than the maximum region size for the database's page size, an error will be returned when `Db::open()` (page 75) is called. The maximum allowable region size will be included in the error message.

The `Db::set_heap_regionsize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `npages`

  The `npages` parameter is the number of pages in a Heap database region.

**Errors**

The `Db::set_heap_regionsize()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the specified region size was too small; the method was called after `Db::open()` (page 75) was called; or if an invalid flag value or parameter was specified.

**Class**

- `Db`

**See Also**

- Database and Related Methods (page 3), `Db::get_heap_regionsize()` (page 48)
Db::set_lk_exclusive()

#include <db_cxx.h>

int
Db::set_lk_exclusive(int nowait_onoff);

Configures the database handle to obtain a write lock on the entire database when it is opened. This gives the handle exclusive access to the database, because the write lock will block all other threads of control for both read and write access.

Use this method to improve the throughput performance on your database for the thread that is controlling this handle. When configured with this method, operations on the database do not acquire page locks as they perform read and/or write operations. Also, the exclusive lock means that operations performed on the database handle will never be blocked waiting for lock due to another thread's activities. The application will also be immune to deadlocks.

On the other hand, use of this method means that you can only have a single thread accessing the database until the handle is closed. For some applications, the loss of multiple threads concurrently operating on the database will result in performance degradation.

Also, use of this method means that you can only have one transaction active for the handle at a time.

Note

This method is incompatible with the DB_THREAD (page 77) and DB_SLICED (page 77) configuration flags.

The Db::set_lk_exclusive() method may not be called after the Db::open() (page 75) method is called.

The Db::set_lk_exclusive() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Replication Notes

Replication applications that use exclusive database handles need to be written with caution. This is because replication clients cannot process updates on an exclusive database until all local handles on the database are closed. Also, attempting to open an exclusive database handle on a currently operating client will result in the open call failing with the error EINVAL.

Also, opening an exclusive database handle on a replication master will result in all clients being locked out of the database. On clients, existing handles on the exclusive database will return the error DB_REP_DEAD_HANDLE when accessed, and must be closed. New handles opened on the exclusive database will block until the master closes its exclusive database handle.
Parameters

`nowait_onoff`

If set to 0, this method will block until it can obtain the exclusive lock on the database. If set to some value other than 0, `DB_LOCK_NOTGRANTED` is returned when the handle is opened if the exclusive database lock cannot be immediately obtained.

Errors

The `Db::set_lk_exclusive()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` (page 75) was called; the method was called on a currently operating replication client; this method was called on a sliced database; or if an invalid flag value or parameter was specified.

Class

Db

See Also

Database and Related Methods (page 3)
Db::set_lorder()

```cpp
#include <db_cxx.h>

int
Db::set_lorder(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() method is called. If the database already exists when Db::open() 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() was called; or if an invalid flag value or parameter was specified.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::set_message_stream()

#include <db_cxx.h>

void Db::set_message_stream(class ostream*);

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, the DbEnv::set_verbose() (page 357) and DbEnv::stat_print() (page 360) methods.

The DbEnv::set_message_stream() (page 339) and Db::set_message_stream() methods are used to display these messages for the application. In this case, the message will include a trailing <newline> character.

Setting stream to NULL unconfigures the interface.

Alternatively, you can use the DbEnv::set_msgfile() (page 342) or Db::set_msgfile() (page 137) methods to display the additional information via a C Library FILE *, or the DbEnv::set_msgcall() (page 340) and Db::set_msgcall() (page 135) methods to capture the additional error information in a way that does not use either output streams or C Library FILE *'s. You should not mix these approaches.

For Db handles opened inside of Berkeley DB environments, calling the Db::set_message_stream() method affects the entire environment and is equivalent to calling the DbEnv::set_message_stream() (page 339) method.

The Db::set_message_stream() method configures operations performed using the specified Db handle, not all operations performed on the underlying database.

The Db::set_message_stream() method may be called at any time during the life of the application.

Parameters

stream

The stream parameter is the application-specified output stream to be used for additional message information.

Class

Db

See Also

Database and Related Methods (page 3)
Db::set_msgcall()

```cpp
#include <db_cxx.h>

void Db::set_msgcall(void (*db_msgcall_fcn)(const DbEnv *dbenv,
                                  const char *msgpfx,
                                  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, DbEnv::set_verbose() (page 357) and DbEnv::stat_print() (page 360).

The DbEnv::set_msgcall() (page 340) 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 DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113) methods to display the messages via an output stream, or the Db::set_msgfile() (page 137) or Db::set_msgfile() (page 342) methods to display the messages via a C library FILE *. You should not mix these approaches.

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 DbEnv::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 three parameters:

- dbenv

  The dbenv parameter is the enclosing database environment.
- `msgpfx`

  The `msgpfx` parameter is the message prefix string that appears before informational messages. This value is the environmental default, and can be changed at any time using `DbEnv::set_msgpfx()` (page 343) or `Db::set_msgpfx()` (page 138).

- `msg`

  The `msg` parameter is the message string.

**Class**

`Db`

**See Also**

Database and Related Methods (page 3)
Db::set_msgfile()

```cpp
#include <db_cxx.h>

void Db::set_msgfile(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, DbEnv::setVerbose() (page 357) and DbEnv::stat_print() (page 360).

The DbEnv::set_msgfile() (page 342) 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 DbEnv::set_message_stream() (page 339) and Db::set_message_stream() (page 134) methods to display the messages via an output stream, or the DbEnv::set_msgcall() (page 340) or Db::set_msgcall() (page 135) methods to capture the additional error information in a way that does not use C library FILE *’s. You should not mix these approaches.

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 DbEnv::set_msgfile() (page 342) 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_msgpfx()

```cpp
#include <db_cxx.h>
void Db::set_msgpfx(const char *msgpfx);
```

Set the prefix string that appears before informational messages issued by Berkeley DB. The length of this string can be no longer than 200.

The Db::set_msgpfx() and DbEnv::set_msgpfx() (page 343) methods do not copy the memory to which the msgpfx parameter refers; rather, they maintain a reference to it. Although this allows applications to modify the informational 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_msgpfx() method affects the entire environment and is equivalent to calling the DbEnv::set_msgpfx() (page 343) method.

The Db::set_msgpfx() method configures operations performed using the specified Db handle, not all operations performed on the underlying database.

The Db::set_msgpfx() method may be called at any time during the life of the application.

**Parameters**

msgpfx

The msgpfx parameter is the application-specified informational prefix for additional informational messages.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::set_pagesize()

```c
#include <db_cxx.h>

int Db::set_pagesize(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 75) method is called. If the database already exists when Db::open() (page 75) 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

pagesize

The pagesize parameter sets the database page size.

**Errors**

The Db::set_pagesize() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() (page 75) 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()

```cpp
#include <db_cxx.h>

int Db::set_partition(u_int32_t parts, DBT *kys,
                      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 implemented as separate database files and can help reduce contention within a logical database. Contention can come from multiple threads of control accessing database pages simultaneously. 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.

The Db::set_partition() method can not be used with a sliced database.

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 greater than or equal to 2, and smaller than 1000000.

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_cxx.h>

int Db::set_partition_dirs(const char **dirs);
```

Specify which directories will contain the database extents. 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()** method is called.

The **Db::set_partition_dirs()** method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 to the **Db::open()** method. The directories must be included in the environment list specified by **DbEnv::add_data_dir()**.

**Errors**

The **Db::set_partition_dirs()** method may fail and throw a **DbException** exception, encapsulating one of the following non-zero errors, or 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()

```cpp
#include <db_cxx.h>

int Db::set_priority(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.

When called on a database that has been opened with sliced support, the Db::set_priority() method is automatically called on each supporting slice database.

The Db::set_priority() method may be called at any time during the life of the application.

The Db::set_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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()

```cpp
#include <db_cxx.h>

int Db::set_q_extentsize(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 75) method is called. If the database already exists when Db::open() (page 75) is called, the information specified to Db::set_q_extentsize() will be ignored.

The Db::set_q_extentsize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  If the method was called after Db::open() (page 75) 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()**

```cpp
#include <db_cxx.h>

int Db::set_re_delim(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()` 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()` method is called. If the database already exists when `Db::open()` is called, the information specified to `Db::set_re_delim()` will be ignored.

The `Db::set_re_delim()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` 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()

```cpp
#include <db_cxx.h>

int
Db::set_re_len(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 149) 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 75) method is called. If the database already exists when Db::open() (page 75) is called, the information specified to Db::set_re_len() will be ignored.

The Db::set_re_len() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after Db::open() (page 75) 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()

```cpp
#include <db_cxx.h>

int db::set_re_pad(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()` method is called. If the database already exists when `db::open()` is called, the information specified to `db::set_re_pad()` will be ignored.

The `db::set_re_pad()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `db::open()` 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_cxx.h>

int
Db::set_re_source(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 146). 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 166) 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 75) 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 75) 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 166) 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()` method is called. If the database already exists when `Db::open()` 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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `Db::open()` was called; or if an invalid flag value or parameter was specified.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
**Db::set_slice_callback()**

```c
#include <db_cxx.h>

Db->set_slice_callback(int (*func)(const Db *db, const Dbt *key, Dbt *slice));
```

The `Db::set_slice_callback()` method is optionally used to identify slices for applications using a sliced environment. The callback identifies the portion of a record's key that is used to allocate records across slices. This is the *slice-relevant* portion of the key. If no callback is specified, then the entire key is used to allocate records across slices.

The `Db::set_slice_callback()` method must be called on an opened database handle. Changing this callback after the database has been created results in undefined behavior.

The `Db::set_backup_callbacks()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**func**

The `func` parameter is the function used to determine which part of a key is slice relevant.

This function takes three parameters:

- **db**
  
  The `db` parameter is the sliced database for which the callback is set.

- **key**
  
  The `key` parameter is the source of the slice specification fields. That is, this is the key portion of the database's records. Some portion of this key is used to determine slice location, and that portion of the key is determined by the function implemented by this callback.

- **slice**
  
  The `slice` parameter is the `Dbt` into which the slice-relevant portion of the key will be written. This `Dbt` is user-owned.

You should zero this `Dbt`, with the exception of the size and data fields. When initializing the structure, set size to `DB_SLICE_BUFLEN`, and data to the address of a 32-bit word aligned buffer of `DB_SLICE_BUFLEN` bytes.

Within the context of the callback, you can set these two fields in the following ways:

- If the slice-relevant portion of the key is one contiguous region, the set data to the first relevant byte in that region, and size to the number of significant bytes. Change no other fields.
• If the slice-relevant portion of the key is more than one contiguous region, then calculate the total size required to hold the slice-relevant data. If this total size is less than DB_SLICE_BUFLEN, then leave the initialized buffer unchanged. If the total size is greater than DB_SLICE_BUFLEN, then:

1. Allocate enough memory to contain all of the slice-relevant data. By default, malloc() is used to allocate memory. However, if your application configured a custom memory allocator using DbEnv::set_alloc() (page 294), then use that function.

2. Set the flags parameter to DB_DBT_APPMALLOC. DB will deallocate the memory before returning from the API call, using the free function corresponding to the allocator just used.

Once you have created a large enough buffer, set the slice Dbt handle's size to the new buffer's size.

After that, concatenate all of the slice-relevant data together into the newly created buffer.

Be aware that the slice Dbt handle's size can be zero, in which case the record will be placed in slice 0. Also, this size may be larger than the key's size, if it is appropriate for your application.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::slice_lookup()

```cpp
#include <db_cxx.h>
Db->slice_lookup(const Dbt *key, Db **result u_int32_t flags));
```

The `Db::slice_lookup()` method returns the subordinate database that contains the records for the provided key.

The `Db::slice_lookup()` method must be called on an opened database handle.

The `Db::slice_lookup()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**key**

The `key` parameter is the `Dbt` for the record that you want to look up.

**result**

The subordinate database containing the record identified by `key` is returned in the `result` parameter.

**flags**

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

**Class**

`Db`

**See Also**

`Database and Related Methods (page 3)`
Db::sort_multiple()

```cpp
#include <db_cxx.h>

int Db::sort_multiple(Dbt *key, Dbt *data, u_int32_t flags);
```

The Db::sort_multiple() method is used to sort a set of 
Dbs into database insert order.

If specified the application specific btree comparison and duplicate comparison functions will
be used if they are configured.

The key and data parameters must contain pairs of items. That is the n-th entry in key must
correspond to the n-th entry in data.

The Db::sort_multiple() method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

**key**

The key parameter must contain a set of Dbt entries in DB_MULTIPLE or DB_MULTIPLE_KEY
format.

The sorted entries will be returned in the key parameter.

**data**

If non-NULL, the data parameter must contain a set of Dbs entries in DB_MULTIPLE format.
Each entry must correspond to an entry in the key parameter.

**flags**

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

- **DB_MULTIPLE**
  
  Sorts one or two DB_MULTIPLE format Dbs. Assumes that key and data specify pairs of key
  and data items to sort together. If the data parameter is NULL the API will sort the key
  arrays according to the btree comparison function.

- **DB_MULTIPLE_KEY**
  
  Sorts a DB_MULTIPLE_KEY format Dbt.

Errors

The Db::sort_multiple() method may fail and throw a DbException exception, encapsulating
one of the following non-zero errors, or return one of the following non-zero errors:
**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)

DBT and Bulk Operations (page 212)
Db::stat()

```
#include <db_cxx.h>

int Db::stat(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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**txnid**

If the operation is part of an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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 `DbEnv::set_alloc()` (page 294) 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 75) 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:

- `uintmax_t hash_bfree;
  
  The number of bytes free on bucket pages.

- `u_int32_t hash_bigpages;
  
  The number of hash overflow pages (created when key/data is too big for the page).

- `uintmax_t hash_big_bfree;
  
  The number of bytes free on hash overflow (big item) pages.

- `u_int32_t hash_buckets;
  
  The number of hash buckets. Returned if DB_FAST_STAT is set.

- `u_int32_t hash_dup;
  
  The number of duplicate pages.

- `uintmax_t hash_dup_free;
  
  The number of bytes free on duplicate pages.

- `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_free;
  The number of pages on the free list.
• 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_metaflags;
  Reports internal flags. For internal use only.
• u_int32_t hash_ext_files;
  The number of external files.
• 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_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_overflows;
  The number of bucket overflow pages (bucket overflow pages are created when items did
  not fit on the main bucket page).
• uintmax_t hash_ovfl_free;
  The number of bytes free on bucket overflow pages.
• 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_version;
  The version of the Hash database. Returned if DB_FAST_STAT is set.

**Heap Statistics**

In the case of a Heap database, the statistics are stored in a structure of type DB_HEAP_STAT.
The following fields will be filled in:
• u_int32_t heap_magic;
  Magic number that identifies the file as a Heap file. Returned if DB_FAST_STAT is set.

• u_int32_t heap_metaflags;
  Reports internal flags. For internal use only.

• u_int32_t heap_ext_files;
  The number of external files.

• u_int32_t heap_nrecs;
  Reports the number of records in the Heap database. Returned if DB_FAST_STAT is set.

• u_int32_t heap_pagecnt;
  The number of pages in the database. Returned if DB_FAST_STAT is set.

• u_int32_t heap_pagesize;
  The underlying database page (and bucket) size, in bytes. Returned if DB_FAST_STAT is set.

• u_int32_t heap_nregions;
  The number of regions in the Heap database. Returned if DB_FAST_STAT is set.

• u_int32_t heap_regionsize;
  The number of pages in a region in the Heap database. Returned if DB_FAST_STAT is set.

• u_int32_t heap_version;
  The version of the Heap database. Returned if DB_FAST_STAT is set.

**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_dup_pg;
  Number of database duplicate pages.

• uintmax_t bt_dup_pgfree;
  Number of bytes free in database duplicate pages.

• u_int32_t bt_empty_pg;
  Number of empty database pages.
• u_int32_t bt_free;
  Number of pages on the free list.

• u_int32_t bt_int_pg;
  Number of database internal pages.

• uintmax_t bt_int_pgfree;
  Number of bytes free in database internal pages.

• u_int32_t bt_leaf_pg;
  Number of database leaf pages.

• uintmax_t bt_leaf_pgfree;
  Number of bytes free in database leaf pages.

• u_int32_t bt_levels;
  Number of levels in the database.

• 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_metaflags;
  Reports internal flags. For internal use only.

• u_int32_t bt_minkey;
  The minimum keys per page. Returned if DB_FAST_STAT is set.

• u_int32_t bt_ext_files;
  The number of external files.

• u_int32_t btndata;

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_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_over_pg;`

  Number of database overflow pages.

- `uintmax_t bt_over_pgfree;`

  Number of bytes free in database overflow pages.

- `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_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_version;`

  The version of the Btree database. Returned if DB_FAST_STAT is set.

**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_cur_recno;`
Next available record number. 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_first_recno;
  
  First undeleted record in the database. Returned if DB_FAST_STAT is set.

- 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_metaflags;
  
  Reports internal flags. For internal use only.

- 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_pages;
  
  Number of pages in the database.

- u_int32_t qs_pagesize;
  
  Underlying database page size, in bytes. Returned if DB_FAST_STAT is set.

- u_int32_t qs_pgfree;
  
  Number of bytes free in database pages.

- 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_version;
The version of the Queue file type. Returned if DB_FAST_STAT is set.

**Errors**

The `Db::stat()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException (page 369)` (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

`DbDeadlockException (page 365)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
Db::stat_print()

#include <db_cxx.h>

int Db::stat_print(u_int32_t flags);

The Db::stat_print() method displays the database statistical information, as described for the Db::stat() (page 157) method. The information is printed to a specified output channel (see the DbEnv::set_msgfile() (page 342) method for more information), or passed to an application callback function (see the DbEnv::set_msgcall() (page 340) method for more information).

The Db::stat_print() method may not be called before the Db::open() (page 75) method is called.

The Db::stat_print() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

For Berkeley DB SQL table or index statistics, see Command Line Features Unique to dbsql (page 760).

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)
The Db::sync() method flushes any cached information to disk. This method operates on the database file level, so if the file contains multiple database handles then this method will flush to disk any information that is cached for any of those handles.

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.

When called on a database that has been opened with sliced support, the Db::sync() method is automatically called on each supporting slice database.

The Db::sync() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

DbDeadlockException or DB_REP_LOCKOUT

The operation was blocked by client/master synchronization.
DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

Db

**See Also**

Database and Related Methods (page 3)
**Db::truncate()**

```c++
#include <db_cxx.h>

int Db::truncate(DbTxn *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 is non-atomic when used on a sliced database. If the operation fails, it is possible for the resulting database files to be in an inconsistent state.

The `Db::truncate()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **txnId**
  
  If the operation is part of an application-specified transaction, the `txnId` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnId` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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 throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **DbDeadlockException or DB_LOCK_DEADLOCK**
  
  A transactional database environment operation was selected to resolve a deadlock.
DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

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()

```c++
#include <db_cxx.h>

int Db::upgrade(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 is non-atomic when used on a sliced database. If the operation fails, it is possible for the underlying database files to be in an inconsistent state.

The `Db::upgrade()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**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), \texttt{Db::upgrade()} will work correctly as long as the \texttt{DB\_DUPSORT} flag is correctly specified. Otherwise, the file cannot be upgraded using \texttt{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 \texttt{DB\_HOME} may be used as the path of the database environment home.

\texttt{Db::upgrade()} is affected by any database directory specified using the \texttt{DbEnv::add\_data\_dir()} method, or by setting the "add\_data\_dir" string in the environment’s \texttt{DB\_CONFIG} file.

**Errors**

The \texttt{Db::upgrade()} method may fail and throw a \texttt{DbException} exception, encapsulating one of the following non-zero errors, or 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**

\texttt{Db}

**See Also**

Database and Related Methods (page 3)
# Db::verify()

```cpp
#include <db_cxx.h>

int Db::verify(const char *file,
               const char *database, ostream *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()` method is called.

The `Db` handle may not be accessed again after `Db::verify()` is called, regardless of its return.

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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

When called on a database that has been opened with sliced support, the `Db::verify()` method is automatically called on each supporting slice database.

## 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 DbEnv::add_data_dir() (page 230) method, or by setting the "add_data_dir" string in the environment's DB_CONFIG file.

Errors

The Db::verify() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL
If the method was called after Db::open() (page 75) 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)
DbHeapRecordId

```cpp
#include <db_cxx.h>

class _exported DbHeapRecordId : private DB_HEAP_RID
{
public:
    db_pgno_t get_pgno() const { return pgno; }
    void set_pgno(db_pgno_t value) { pgno = value; }

    db_indx_t get_indx() const { return indx; }
    void set_indx(db_indx_t value) { indx = value; }

    DB_HEAP_RID *get_DB_HEAP_RID() { return (DB_HEAP_RID *)this; }
    const DB_HEAP_RID *get_const_DB_HEAP_RID() const
        { return (const DB_HEAP_RID *)this; }

    static DbHeapRecordId* get_DbHeapRecordId(DB_HEAP_RID *rid)
        { return (DbHeapRecordId *)rid; }
    static const DbHeapRecordId* get_const_DbHeapRecordId(DB_HEAP_RID *rid)
        { return (const DbHeapRecordId *)rid; }

    DbHeapRecordId(db_pgno_t pgno, db_indx_t indx);
    DbHeapRecordId();
    ~DbHeapRecordId();
    DbHeapRecordId(const DbHeapRecordId &);
    DbHeapRecordId &operator = (const DbHeapRecordId &);
};
```

Content used for the key in a Heap database record. Berkeley DB instantiates an object of this class for you when you create a record in a Heap database. You should never instantiate an object of this class or modify the contents of this class yourself; Berkeley DB must create and manage it for you.

This object is returned in the key Dbt parameter of the method that you use to add a record to the Heap database.

**Class Methods**

`get_pgno()`
Returns the database page number where the record is stored.

`get_indx()`
Returns the index in the offset table where the record can be found.

`get_DB_HEAP_RID()`
Returns a pointer to the underlying C-language structure used to store the database page number and offset table index information.
set_pgno()

For internal use only. Changing the offset index has unpredictable results.

set_indx()

For internal use only. Changing the offset index has unpredictable results.

See Also

Database and Related Methods (page 3),
Chapter 3. The Dbc Handle

A Dbc object is a handle for a cursor into a Berkeley DB database.

Dbc handles are not free-threaded. Cursor handles may be shared by multiple threads if access is serialized by the application.

You create a Dbc using the Db::cursor() (page 179) 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 Dbc::close() (page 182) has been called, the handle may not be accessed again, regardless of the method's return.
# Database Cursors and Related Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db::cursor()</td>
<td>Create a cursor handle</td>
</tr>
<tr>
<td>Dbc::close()</td>
<td>Close a cursor handle</td>
</tr>
<tr>
<td>Dbc::cmp()</td>
<td>Compare two cursors for equality.</td>
</tr>
<tr>
<td>Dbc::count()</td>
<td>Return count of duplicates for current key</td>
</tr>
<tr>
<td>Dbc::del()</td>
<td>Delete current key/data pair</td>
</tr>
<tr>
<td>Dbc::dup()</td>
<td>Duplicate the cursor handle</td>
</tr>
<tr>
<td>Dbc::get()</td>
<td>Retrieve by cursor</td>
</tr>
<tr>
<td>Dbc::put()</td>
<td>Store by cursor</td>
</tr>
<tr>
<td>Dbc::set_priority(), Dbc::get_priority()</td>
<td>Set/get the cursor's cache priority</td>
</tr>
</tbody>
</table>
Db::cursor()

```cpp
#include <db_cxx.h>

int Db::cursor(DbTxn *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 either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_flags()` (page 323) or `Db::open()` (page 75) methods. If cursor operations are to be transaction-protected, the `txnid` parameter must be a transaction handle returned from `DbEnv::txn_begin()` (page 674); 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_CURSOR_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, this flag 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. Snapshot isolation is not supported with replication.

**Errors**

The `Db::cursor()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException` (page 369) (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`Db`
See Also

Database Cursors and Related Methods (page 178)
Dbc::close()

```cpp
#include <db_cxx.h>

int
Dbc::close(void);
```

The `Dbc::close()` method discards the cursor.

It is possible for the `Dbc::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 `Dbc::close()` method has been called, regardless of its return value, you cannot 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 `Dbc::close()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Errors

The `Dbc::close()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

`DbDeadlockException (page 365)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

`DbLockNotGrantedException (page 366)` is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**EINVAL**

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

Dbc

See Also

Database Cursors and Related Methods (page 178)
Dbc::cmp()

```cpp
#include <db_cxx.h>

int Dbc::cmp(Dbc *other_cursor, int *result, u_int32_t flags);
```

The `Dbc::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 `Dbc::cmp()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `Dbc::cmp()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

Dbc
See Also

Database Cursors and Related Methods (page 178)


Dbc::count()

```cpp
#include <db_cxx.h>

int
Dbc::count(db_recno_t *countp, u_int32_t flags);
```

The `Dbc::count()` method returns a count of the number of data items for the key to which
the cursor refers.

The `Dbc::count()` method either returns a non-zero error value or throws an exception that
encapsulates a non-zero error value on failure, and returns 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 `Dbc::count()` method may fail and throw a `DbException` exception, encapsulating one of
the following non-zero errors, or return one of the following non-zero errors:

- **DbRepHandleDeadException or 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 throw
a `DbRepHandleDeadException` (page 369) (if your application is configured to throw
exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle
and open a new one in order to continue processing.

- **DbDeadlockException or DB_REP_LOCKOUT**

  The operation was blocked by client/master synchronization.

  `DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw
exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.

- **EINVAL**

  If the cursor has not been initialized; or if an invalid flag value or parameter was specified.

**Class**

Dbc
See Also

Database Cursors and Related Methods (page 178)
Dbc::del()

```cpp
#include <db_cxx.h>

int
Dbc::del(u_int32_t flags);
```

The `Dbc::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()` method, the `Dbc::del()` 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 `Dbc::del()` method will return DB_KEYEMPTY if the element has already been deleted. The `Dbc::del()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `Dbc::del()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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** 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.

**DbDeadlockException or DB_LOCK_DEADLOCK**

A transactional database environment operation was selected to resolve a deadlock.
**DbDeadlockException (page 365)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

**DbLockNotGrantedException (page 366)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException (page 369)` (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

**DbDeadlockException (page 365)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.

**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 `DbEnv::open()` (page 286).

**Class**

Dbc
See Also

Database Cursors and Related Methods (page 178)
Dbc::dup()

```cpp
#include <db_cxx.h>

int Dbc::dup(Dbc **cursorp, u_int32_t flags);
```

The `Dbc::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 `Dbc::dup()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

### Parameters

**cursorp**

The `Dbc::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 179) method.

### Errors

The `Dbc::dup()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbRepHandleDeadException or 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 throw a `DbRepHandleDeadException` (page 369) (if your application is configured to throw exceptions), or return `DB_REP_HANDLE_DEAD`. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_REP_LOCKOUT` is returned.
EINVAL

An invalid flag value or parameter was specified.

Class

Dbc

See Also

Database Cursors and Related Methods (page 178)
**Dbc::get()**

```cpp
#include <db_cxx.h>

int Dbc::get(Dbt *key, Dbt *data, u_int32_t flags);

int Dbc::pget(Dbt *key, Dbt *pkey, Dbt *data, u_int32_t flags);
```

The `Dbc::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 `Dbc::get()` and `Dbc::pget()` methods return the key from the secondary index and the data item from the primary database. In addition, the `Dbc::pget()` method returns the key from the primary database. In databases that are not secondary indices, the `Dbc::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 `Dbc::get()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

If `Dbc::get()` fails for any reason, the state of the cursor will be unchanged.

**Parameters**

**key**

The key `Dbt` operated on.

If `DB_DBT_PARTIAL` is set for the `Dbt` used for this parameter, and if the `flags` parameter is set to `DB_GET_BOTH`, `DB_GET_BOTH_RANGE`, `DB_SET`, or `DB_SET_RECNO`, then this method will fail and return `EINVAL`.

**pkey**

The return key from the primary database. If `DB_DBT_PARTIAL` is set for the `Dbt` used for this parameter, then this method will fail and return `EINVAL`.

**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 `Dbc::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, `Dbc::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 `Dbc::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 `Dbc::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 `Dbc::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 Dbc::get() and Dbc::pget() methods return the record number of the primary database in data. In addition, the Dbc::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 67) method.

This flag is not supported for Heap databases.

- 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, Dbc::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 Dbc::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, Dbc::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 Dbc::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 `Dbc::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.

If using a Heap database, this flag results in this method returning DB_NOTFOUND.

- **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, `Dbc::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 `Dbc::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.

  If using a Heap database, this flag is identical to the `DB_NEXT` flag.

- **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, `Dbc::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 `Dbc::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 `Dbc::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.

  If using a Heap database, this flag results in this method returning DB_NOTFOUND.

- **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, `Dbc::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 `Dbc::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.

If using a Heap database, this flag is identical to the DB_PREV flag.

• DB_SET

Move the cursor to the specified key/data pair of the database, and return the datum associated with the given key.

The `Dbc::get()` method will return DB_NOTFOUND if DB_SET is set and no matching keys are found. The `Dbc::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, `Dbc::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_IGNORELEASE

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, Recno, or Heap databases, data items starting at the current cursor position are entered into the buffer. The record number (or the RID, in the case of Heap) of the first record will be returned in the `key` parameter. For Queue and Recno, the record number of each subsequent returned record must be calculated from this value. For Heap databases, the RID of subsequent returned records cannot be known. 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_BT_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 `DbMultipleDataIterator` (page 214) class.

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, Hash or Heap databases, the multiple key and data pairs can be iterated over using the `DbMultipleKeyDataIterator` (page 216) class.

In the case of Queue or Recno databases, the multiple record number and data pairs can be iterated over using the `DbMultipleRecnoDataIterator` (page 218) class.

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_RECNOS` 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 `Db::get()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **DbMemoryException or DB_BUFFER_SMALL**

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

  `DbMemoryException` (page 368) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_BUFFER_SMALL` is returned.

- **DbDeadlockException or DB_LOCK_DEADLOCK**

  A transactional database environment operation was selected to resolve a deadlock.
DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or 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.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**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 Dbc::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**

Dbc

**See Also**

Database Cursors and Related Methods (page 178)
Dbc::get_priority()

#include <db_cxx.h>

int 
Dbc::get_priority(DB_CACHE_PRIORITY *priorityp);

The Dbc::get_priority() method returns the cache priority for pages referenced by the Dbc handle.

The Dbc::get_priority() method may be called at any time during the life of the application.

The Dbc::get_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

priorityp

The Dbc::get_priority() method returns a reference to the cache priority for pages referenced by the Dbc handle in priorityp.

Class

Dbc

See Also

Database Cursors and Related Methods (page 178)
Dbc::put()

```cpp
#include <db_cxx.h>

int
Dbc::put(Dbt *key, Dbt *data, u_int32_t flags);
```

The `Dbc::put()` method stores key/data pairs into the database.

Unless otherwise specified, the `Dbc::put()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

If `Dbc::put()` fails for any reason, the state of the cursor will be unchanged. If `Dbc::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.

If creating a new record in a Heap database, the key `Dbt` must be empty. The `put` method will return the new record's Record ID (RID) in the key `Dbt`.

**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 75) for more information.

  The `DB_AFTER` flag may not be specified to the Queue access method.
The `Dbc::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 75) for more information.

  The DB_BEFORE flag may not be specified to the Queue access method.

  The `Dbc::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 `Dbc::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 204) 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 Dbc::put() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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_HEAP_FULL

An attempt was made to add or update a record in a Heap database. However, the size of the database was constrained using the Db::set_heapsize() (page 128) method, and that limit has been reached.

DbDeadlockException or DB_LOCK_DEADLOCK

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set lk exclusive() (page 131) for more information.
DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**DbRepHandleDeadException or 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 throw a DbRepHandleDeadException (page 369) (if your application is configured to throw exceptions), or return DB_REP_HANDLE_DEAD. The application will need to discard the handle and open a new one in order to continue processing.

**DbDeadlockException or DB_REP_LOCKOUT**

The operation was blocked by client/master synchronization.

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_REP_LOCKOUT is returned.

**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 DbEnv::open() (page 286).

**Class**

Dbc

**See Also**

Database Cursors and Related Methods (page 178)
Dbc::set_priority()

```cpp
#include <db_cxx.h>

int
Dbc::set_priority(DB_CACHE_PRIORITY priority);
```

Set the cache priority for pages referenced by the Dbc 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 Dbc::set_priority() method is only advisory, and does not guarantee pages will be treated in a specific way.

The Dbc::set_priority() method may be called at any time during the life of the application.

The Dbc::set_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

Dbc

**See Also**

Database Cursors and Related Methods (page 178)
Chapter 4. The Dbt Handle

#include <db_cxx.h>

class Dbt {
public:
    Dbt(void *data, size_t size);
    Dbt();
    Dbt(const Dbt &);
    Dbt &operator = (const Dbt &);
    ~Dbt();

    void *get_data() const;
    void set_data(void *);

    u_int32_t get_size() const;
    void set_size(u_int32_t);

    u_int32_t get_ulen() const;
    void set_ulen(u_int32_t);

    u_int32_t get_dlen() const;
    void set_dlen(u_int32_t);

    u_int32_t get_doff() const;
    void set_doff(u_int32_t);

    u_int32_t get_flags() const;
    void set_flags(u_int32_t);

    DBT *Dbt::get_DBT();
    const DBT *Dbt::get_const_DBT() const;
    static Dbt *Dbt::get_Dbtx(DBT *dbt);
    static const Dbt *Dbt::get_const_Dbtx(const DBT *dbt);
};

The Dbt class is used to encode key and data items in a Berkeley DB database.

Storage and retrieval for the Db access methods are based on key/data pairs. Both key and data items are represented by Dbt objects. 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.

In the case when the flags structure element is set to 0, when the application is providing Berkeley DB a key or data item to store into the database, Berkeley DB expects the data object 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 object 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 Dbtxs is only compatible with single threaded usage of Berkeley DB.
Access to Dbt objects is not re-entrant. In particular, if multiple threads simultaneously access
the same Dbt object using Db API calls, the results are undefined, and may result in a crash.
One easy way to avoid problems is to use Dbt objects that are constructed as stack variables.

Each Dbt object has an associated DBT struct, which is used by the underlying implementation
of Berkeley DB and its C-language API. The Dbt::get_DBT() method returns a pointer to this
struct. Given a const Dbt object, Dbt::get_const_DBT() returns a const pointer to the same
struct.

Given a DBT struct, the Dbt::get_Dbt() method returns the corresponding Dbt object, if
there is one. If the DBT object was not associated with a Dbt (that is, it was not returned from
a call to Dbt::get_DBT()), then the result of Dbt::get_Dbt() is undefined. Given a const
DBT struct, Dbt::get_const_Dbt() returns the associated const Dbt object, if there is one.

These methods may be useful for Berkeley DB applications including both C and C++ language
software. It should not be necessary to use these calls in a purely C++ application.

• Dbt::set_data(void *data)
  Set the data array.
  The data parameter is an array of bytes to be used to set the content for the Dbt.

• Dbt::get_data()
  Return the data array.

• Dbt::set_size(u_int32_t size)
  Sets the byte size of the data array, in bytes.

• Dbt::get_size()
  Return the data array size.

• Dbt::set_ulen(u_int32_t value)
  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.

• Dbt::get_ulen()
  Return the length in bytes 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.

• Dbt::set_dlen(u_int32_t dlen)
Set the length of the partial record being read or written by the application, in bytes. See the DB_DBT_PARTIAL flag for more information.

- `Dbt::get_dlen()`
  Return the length of the partial record, in bytes.

- `Dbt::set_doff(u_int32_t value)`
  Sets the offset of the partial record being read or written by the application, in bytes. See the DB_DBT_PARTIAL flag for more information.

- `Dbt::get_doff()`
  Return the offset of the partial record, in bytes.

- `Dbt::set_flags(u_int32_t flags)`
  Set the object flag value.

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

  - `DB_DBT_EXT_FILE`
    Set this flag on a Dbt used for the data portion of a record to indicate that the Dbt stores external file data. If this flag is set, and if the database otherwise supports external files, then the data contained by this Dbt will be stored as an external file, regardless of whether it exceeds the external file threshold in size.

  - `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.

    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.

If DB_DBT_MALLOC or DB_DBT_REALLOC is specified, Berkeley DB allocates a properly sized byte array to contain the data. This can be convenient if you know little about the nature of the data, specifically the size of data in the database. However, if your application makes repeated calls to retrieve keys or data, you may notice increased garbage collection due to this allocation. If you know the maximum size of data you are retrieving, you might decrease the memory burden and speed your application by allocating your own byte array and using DB_DBT_USERMEM. Even if you don't know the maximum size, you can use this option and reallocate your array whenever your retrieval API call returns an DB_BUFFER_SMALL error or throws an exception encapsulating an DB_BUFFER_SMALL.

• **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 81](#)) method in a database that supports duplicate records. Partial puts in databases supporting duplicate records must be done using a Dbc::put() ([page 202](#)) 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.

This flag is ignored when used with the pkey parameter on DB->pget() or DBcursor->pget().

- **DB_DBT_APPMALLOC**
  
  After an application-supplied callback routine passed to either Db::associate() (page 6) or Db::set_append_recno() (page 92) 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.

- **DB_DBT_READONLY**
  
  When this flag is set Berkeley DB will not write into the DBT. This may be set on key values in cases where the key is a static string that cannot be written and Berkeley DB might try to update it because the application has set a user defined comparison function.
## DBT and Bulk Operations

<table>
<thead>
<tr>
<th>DBT and Bulk Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db::sort_multiple()</td>
<td>Sort a set of DBTs</td>
</tr>
<tr>
<td>DbMultipleIterator</td>
<td>Base class for bulk get retrieval</td>
</tr>
<tr>
<td>DbMultipleDataIterator</td>
<td>Bulk retrieval iterator for data items</td>
</tr>
<tr>
<td>DbMultipleKeyDataIterator</td>
<td>Bulk retrieval iterator for key/data pairs</td>
</tr>
<tr>
<td>DbMultipleRecnoDataIterator</td>
<td>Bulk retrieval iterator for record number / data item pairs</td>
</tr>
<tr>
<td>DbMultipleBuilder</td>
<td>Base class for bulk buffer building</td>
</tr>
<tr>
<td>DbMultipleDataBuilder</td>
<td>Bulk buffer builder for data items</td>
</tr>
<tr>
<td>DbMultipleKeyDataBuilder</td>
<td>Bulk buffer builder for key/data pairs</td>
</tr>
<tr>
<td>DbMultipleRecnoDataBuilder</td>
<td>Bulk buffer builder for record number / data pairs</td>
</tr>
</tbody>
</table>
DbMultipleIterator

```cpp
#include <db_cxx.h>

class DbMultipleIterator
{
};
```

The DbMultipleIterator class is a shared package-private base class for the three types of bulk-return iterator; it should never be instantiated directly, but it handles the functionality shared by its subclasses.

**Class**

DbMultipleIterator

**See Also**

DBT and Bulk Operations (page 212)
DbMultipleDataIterator

```cpp
#include <db_cxx.h>

class DbMultipleDataIterator
{
public:
    DbMultipleDataIterator(const Dbt &dbt);
    
    bool next(Dbt &data);
};
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the Db::get() (page 31) or Dbc::get() (page 193) 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 classes.

The DbMultipleDataIterator class is used to iterate through data returned using the DB_MULTIPLE flag from a database belonging to any access method.

The constructor takes the  The Dbt Handle (page 207) returned by the call to Db::get() (page 31) or Dbc::get() (page 193) that used the DB_MULTIPLE flag.

**Note**

All instances of the bulk retrieval classes may be used only once, and to traverse the bulk retrieval buffer in the forward direction only. However, they are nondestructive, so multiple iterators can be instantiated and used on the same returned data Dbt.

Parameters are:

- `dbt`

  The `dbt` parameter is a data  The Dbt Handle (page 207) returned by the call to Db::get() (page 31) or Dbc::get() (page 193) that used the DB_MULTIPLE flag.

**DbMultipleDataIterator.next()**

The DbMultipleDataIterator.next() method returns the next data item in the original bulk retrieval buffer.

The DbMultipleDataIterator.next() method method returns false if no more data are available, and true otherwise.

Parameters are:

- `data`

  The `data` parameter is a  The Dbt Handle (page 207) that will be filled in with a reference to a buffer, a size, and an offset that together yield the next data item in the original bulk retrieval buffer.
Class

DbMultipleIterator

See Also

DBT and Bulk Operations (page 212)
DbMultipleKeyDataIterator

```cpp
#include <db_cxx.h>

class DbMultipleKeyDataIterator
{
public:
   DbMultipleKeyDataIterator(const Dbt &dbt);
   bool next(Dbt &key, Dbt &data);
};
```

If either of the DB_MULTIPLE or DB_MULTIPLE_KEY flags were specified to the Db::get() (page 31) or Dbc::get() (page 193) methods, the data Dbt returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through these classes.

The DbMultipleDataIterator class is used to iterate through data returned using the DB_MULTIPLE_KEY flag from a database belonging to Btree or Hash access methods.

The constructor takes the The Dbt Handle (page 207) returned by the call to Db::get() (page 31) or Dbc::get() (page 193) that used the DB_MULTIPLE_KEY flag.

**Note**

All instances of the bulk retrieval classes may be used only once, and to traverse the bulk retrieval buffer in the forward direction only. However, they are nondestructive, so multiple iterators can be instantiated and used on the same returned data Dbt.

Parameters are:

- dbt

  The dbt parameter is a data The Dbt Handle (page 207) returned by the call to Db::get() (page 31) or Dbc::get() (page 193) that used the DB_MULTIPLE_KEY flag.

DbMultipleKeyDataIterator.next()

The DbMultipleKeyDataIterator.next() method returns the next data item in the original bulk retrieval buffer.

The DbMultipleKeyDataIterator.next() method method returns false if no more data are available, and true otherwise.

Parameters are:

- key

  The key parameter is a The Dbt Handle (page 207) that will be filled in with a reference to a buffer, a size, and an offset that together yield the next data item in the original bulk retrieval buffer.
• data

The data parameter is a The Dbt Handle (page 207) that will be filled in with a reference to a buffer, a size, and an offset that together yield the next data item in the original bulk retrieval buffer.

Class

DbMultipleIterator

See Also

DBT and Bulk Operations (page 212)
### DbMultipleRecnoDataIterator

```cpp
#include <db_cxx.h>

class DbMultipleRecnoDataIterator
{
public:
    DbMultipleRecnoDataIterator(const Dbt &dbt);

    bool next(db_recno_t &key, Dbt &data);
};
```

If either of the `DB_MULTIPLE` or `DB_MULTIPLE_KEY` flags were specified to the `Db::get()` (page 31) or `Dbc::get()` (page 193) methods, the data `Dbt` returned by those interfaces will refer to a buffer that is filled with data. Access to that data is through these classes.

The `DbMultipleDataIterator` class is used to iterate through data returned using the `DB_MULTIPLE_KEY` flag from a database belonging to Queue or Recno access methods.

The constructor takes the `The Dbt Handle` (page 207) returned by the call to `Db::get()` (page 31) or `Dbc::get()` (page 193) that used the `DB_MULTIPLE_KEY` flag.

**Note**

All instances of the bulk retrieval classes may be used only once, and to traverse the bulk retrieval buffer in the forward direction only. However, they are nondestructive, so multiple iterators can be instantiated and used on the same returned data `Dbt`.

Parameters are:

- `dbt`

  The `dbt` parameter is a `The Dbt Handle` (page 207) returned by the call to `Db::get()` (page 31) or `Dbc::get()` (page 193) that used the `DB_MULTIPLE_KEY` flag.

#### DbMultipleRecnoDataIterator.next()

The `DbMultipleRecnoDataIterator.next()` method returns the next data item in the original bulk retrieval buffer.

The `DbMultipleRecnoDataIterator.next()` method method returns `false` if no more data are available, and `true` otherwise.

Parameters are:

- `key`

  The `key` parameter is a `The Dbt Handle` (page 207) that will be filled in with a reference to a buffer, a size, and an offset that together yield the next data item in the original bulk retrieval buffer.
• data

The \texttt{data} parameter is a \textit{The Dbt Handle (page 207)} that will be filled in with a reference to a buffer, a size, and an offset that together yield the next data item in the original bulk retrieval buffer.

\textbf{Class}

\texttt{DbMultipleIterator}

\textbf{See Also}

\textit{DBT and Bulk Operations (page 212)
DbMultipleBuilder

```cpp
#include <db_cxx.h>

class DbMultipleBuilder
{
};
```

The DbMultipleBuilder class is a shared package-private base class for the three types of bulk buffer builders; it should never be instantiated directly, but it handles the functionality shared by its subclasses.

**Class**

Dbt

**See Also**

DBT and Bulk Operations (page 212)
DbMultipleDataBuilder

```cpp
#include <db_cxx.h>

class DbMultipleDataBuilder
{
public:
    DbMultipleDataBuilder(Dbt &dbt);

    bool append(void *dbuf, size_t dlen);
    bool reserve(void *ddest, size_t dlen);
};
```

This class builds a bulk buffer for use when the `DB_MULTIPLE` flag is specified to either the `Db::put()` (page 81) or `Db::del()` (page 23) methods. The buffer in the `Dbt` passed to the constructor is filled by calls to `DbMultipleDataBuilder.append()` (page 221) or `DbMultipleDataBuilder.reserve()` (page 222).

The constructor takes a The Dbt Handle (page 207) that must be configured to contain a buffer managed by the application, with the `ulen` field set to the size of the buffer.

**Note**

All instances of the bulk retrieval classes may be used only once, and to build the bulk buffer in the forward direction only.

Parameters are:

- `dbt`

  The `dbt` parameter is a The Dbt Handle (page 207) that must already be configured to contain a buffer managed by the application, with the `ulen` field set to the size of the buffer, which must be a multiple of 4.

**DbMultipleDataBuilder.append()**

The `DbMultipleDataBuilder.append()` method copies a data item to the end of the buffer.

The `DbMultipleDataBuilder.append()` method returns `false` if the data does not fit in the buffer and `true` otherwise.

Parameters are:

- `dbuf`

  A pointer to the data to be copied into the bulk buffer.

- `dlen`

  The number of bytes to be copied.
DbMultipleDataBuilder.reserve()

The DbMultipleDataBuilder.reserve() method reserves space for the next data item in the bulk buffer. Unlike the append(), no data is actually copied into the bulk buffer by reserve(): copying the data is the responsibility of the application.

The DbMultipleDataBuilder.reserve() method returns false if the data does not fit in the buffer and true otherwise.

Parameters are:

• ddest
  Set to a pointer to the position in the bulk buffer reserved for the data item, if enough space is available.

• dlen
  The number of bytes to reserve.

Class

Dbt

See Also

DBT and Bulk Operations (page 212)
DbMultipleKeyDataBuilder

```cpp
#include <db_cxx.h>

class DbMultipleKeyDataBuilder
{
public:
    DbMultipleKeyDataBuilder(Dbt &dbt);
    bool append(void *kbuf, size_t klen, void *dbuf, size_t dlen);
    bool reserve(void *&kdest, size_t klen, void *&ddest, size_t dlen);
};
```

This class builds a bulk buffer for use when the `DB_MULTIPLE_KEY` flag is specified to either the `Db::put()` (page 81) or `Db::del()` (page 23) methods with the btree or hash access methods. The buffer in the `Dbt` passed to the constructor is filled by calls to `DbMultipleKeyDataBuilder.append()` (page 223) or `DbMultipleKeyDataBuilder.reserve()` (page 224).

The constructor takes a The Dbt Handle (page 207) that must be configured to contain a buffer managed by the application, with the `ulen` field set to the size of the buffer.

**Note**

All instances of the bulk retrieval classes may be used only once, and to build the bulk buffer in the forward direction only.

Parameters are:

- `dbt`

  The `dbt` parameter is a The Dbt Handle (page 207) that must already be configured to contain a buffer managed by the application, with the `ulen` field set to the size of the buffer, which must be a multiple of 4.

**DbMultipleKeyDataBuilder.append()**

The `DbMultipleKeyDataBuilder.append()` method copies a key/data pair to the end of the buffer.

The `DbMultipleKeyDataBuilder.append()` method returns `false` if the key/data pair does not fit in the buffer and `true` otherwise.

Parameters are:

- `kbuf`

  A pointer to the key to be copied into the bulk buffer.

- `klen`
The number of bytes of the key to be copied.

- `dbuf`

A pointer to the data item to be copied into the bulk buffer.

- `dlen`

The number of bytes of the data item to be copied.

### `DbMultipleKeyDataBuilder.reserve()`

The `DbMultipleKeyDataBuilder.reserve()` method reserves space for the next key/data pair in the bulk buffer. Unlike the `append()`, no data is actually copied into the bulk buffer by `reserve()`: copying the data is the responsibility of the application.

The `DbMultipleKeyDataBuilder.reserve()` method returns `false` if the data does not fit in the buffer and `true` otherwise.

Parameters are:

- `kdest`

  Set to a pointer to the position in the bulk buffer reserved for the key, if enough space is available.

- `klen`

  The number of bytes to reserve for the key.

- `ddest`

  Set to a pointer to the position in the bulk buffer reserved for the data item, if enough space is available.

- `dlen`

  The number of bytes to reserve for the data item.

### Class

`DbMultipleBuilder`

### See Also

`DBT and Bulk Operations (page 212)`
DbMultipleRecnoDataBuilder

```cpp
#include <db_cxx.h>

class DbMultipleRecnoDataBuilder
{
 public:
  DbMultipleRecnoDataBuilder(Dbt &dbt);

  bool append(db_recno_t recno, void *dbuf, size_t dlen);
  bool reserve(db_recno_t recno, void *&ddest, size_t dlen);
};
```

This class builds a bulk buffer for use when the DB_MULTIPLE_KEY flag is specified to either
the Db::put() (page 81) or Db::del() (page 23) methods with the recno or queue access
methods, or for the key when the DB_MULTIPLE flag is used. The buffer in the Dbt passed
to the constructor is filled by calls to DbMultipleRecnoDataBuilder.append() (page 225) or
DbMultipleRecnoDataBuilder.reserve() (page 226).

The constructor takes a The Dbt Handle (page 207) that must be configured to contain a
buffer managed by the application, with the ulen field set to the size of the buffer.

Note

All instances of the bulk retrieval classes may be used only once, and to build the bulk
buffer in the forward direction only.

Parameters are:

• dbt

  The dbt parameter is a The Dbt Handle (page 207) that must already be configured to
  contain a buffer managed by the application, with the ulen field set to the size of the
  buffer, which must be a multiple of 4.

  bool append(db_recno_t recno, void *dbuf, size_t dlen);

DbMultipleRecnoDataBuilder.append()

The DbMultipleRecnoDataBuilder.append() method copies a record number / data pair to
the end of the buffer.

The DbMultipleRecnoDataBuilder.append() method returns false if the record number /
data pair does not fit in the buffer and true otherwise.

Parameters are:

• recno

  The record number to append.
• **dbuf**
  A pointer to the data item to be copied into the bulk buffer.

• **dlen**
  The number of bytes of the data item to be copied.

### DbMultipleRecnoDataBuilder.reserve()

The `DbMultipleRecnoDataBuilder.reserve()` method reserves space for the next record number / data pair in the bulk buffer. The record number is appended, but unlike the `append()`, the data is not copied into the bulk buffer by `reserve()`: copying the data is the responsibility of the application.

The `DbMultipleRecnoDataBuilder.reserve()` method returns `false` if the record does not fit in the buffer and `true` otherwise.

Parameters are:

• **recno**
  The record number to append.

• **ddest**
  Set to a pointer to the position in the bulk buffer reserved for the data item, if enough space is available.

• **dlen**
  The number of bytes to reserve for the data item.

### Class

**DbMultipleBuilder**

### See Also

[DBT and Bulk Operations (page 212)](#)
Chapter 5. The DbEnv Handle

The DbEnv 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 DbEnv handle are used to configure the environment as well as to operate on subsystems and databases in the environment.

DbEnv handles are opened using the DbEnv::open() (page 286) method.

When you are done using your environment, close it using the DbEnv::close() (page 235) 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

<table>
<thead>
<tr>
<th>Database Environment Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbEnv::backup()</td>
<td>Hot back up an entire environment</td>
</tr>
<tr>
<td>DbEnv::close()</td>
<td>Close an environment</td>
</tr>
<tr>
<td>DbEnv</td>
<td>Create an environment handle</td>
</tr>
<tr>
<td>DbEnv::dbackup()</td>
<td>Hot back up a single environment file</td>
</tr>
<tr>
<td>DbEnv::dbremove()</td>
<td>Remove a database</td>
</tr>
<tr>
<td>DbEnv::dbrename()</td>
<td>Rename a database</td>
</tr>
<tr>
<td>DbEnv::err()</td>
<td>Error message</td>
</tr>
<tr>
<td>DbEnv::failchk()</td>
<td>Check for thread failure</td>
</tr>
<tr>
<td>DbEnv::fileid_reset()</td>
<td>Reset database file IDs</td>
</tr>
<tr>
<td>DbEnv::full_version()</td>
<td>Return full version information</td>
</tr>
<tr>
<td>Db::get_env()</td>
<td>Return the Db's underlying DbEnv handle</td>
</tr>
<tr>
<td>DbEnv::get_home()</td>
<td>Return environment's home directory</td>
</tr>
<tr>
<td>DbEnv::get_open_flags()</td>
<td>Return flags with which the environment was opened</td>
</tr>
<tr>
<td>DbEnv::get_slices()</td>
<td>Returns an array of slice environments</td>
</tr>
<tr>
<td>DbEnv::log_verify()</td>
<td>Verify log files of an environment</td>
</tr>
<tr>
<td>DbEnv::lsn_reset()</td>
<td>Reset database file LSNs</td>
</tr>
<tr>
<td>DbEnv::msg()</td>
<td>Informational message</td>
</tr>
<tr>
<td>DbEnv::open()</td>
<td>Open an environment</td>
</tr>
<tr>
<td>DbEnv::remove()</td>
<td>Remove an environment</td>
</tr>
<tr>
<td>DbEnv::stat_print()</td>
<td>Environment statistics</td>
</tr>
<tr>
<td>DbEnv::strerror()</td>
<td>Error strings</td>
</tr>
<tr>
<td>DbEnv::version()</td>
<td>Return version information</td>
</tr>
</tbody>
</table>

**Environment Configuration**

| DbEnv::add_data_dir()           | Add an environment data directory |
| DbEnv::get_slice_count()        | Returns the number of slices configured for the environment |
| DbEnv::set_alloc()              | Set local space allocation functions |
| DbEnv::set_app_dispatch()       | Configure application recovery callback |
| DbEnv::set_backup_callbacks(), | Set/get callbacks used for environment hot backups |
| DbEnv::get_backup_callbacks()   | |
| DbEnv::set_backup_config(),    | Set/get environment hot backup configuration options |
| DbEnv::get_backup_config()      | |
| DbEnv::set_data_dir(),          | Set/get the environment data directory |
| DbEnv::get_data_dirs()          | |
### Database Environment Operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbEnv::set_create_dir(), DbEnv::get_create_dir()</td>
<td>Add an environment data directory</td>
</tr>
<tr>
<td>DbEnv::set_encrypt(), DbEnv::get_encrypt_flags()</td>
<td>Set/get the environment cryptographic key</td>
</tr>
<tr>
<td>DbEnv::set_event_notify()</td>
<td>Set event notification callback</td>
</tr>
<tr>
<td>DbEnv::set_errno()</td>
<td>Set error message callbacks</td>
</tr>
<tr>
<td>DbEnv::set_errfile(), DbEnv::get_errfile()</td>
<td>Set/get error message FILE</td>
</tr>
<tr>
<td>DbEnv::set_error_stream()</td>
<td>Set C++ ostream used for error messages</td>
</tr>
<tr>
<td>DbEnv::set_errprefix(), DbEnv::get_errprefix()</td>
<td>Set/get error message prefix</td>
</tr>
<tr>
<td>DbEnv::set_feedback()</td>
<td>Set feedback callback</td>
</tr>
<tr>
<td>DbEnv::set_flags(), DbEnv::get_flags()</td>
<td>Environment configuration</td>
</tr>
<tr>
<td>DbEnv::set_intermediate_dir_mode(), DbEnv::get_intermediate_dir_mode()</td>
<td>Set/get intermediate directory creation mode</td>
</tr>
<tr>
<td>DbEnv::set_isalive()</td>
<td>Set thread is-alive callback</td>
</tr>
<tr>
<td>DbEnv::set_memory_init(), DbEnv::get_memory_init()</td>
<td>Set/get initial memory allocation</td>
</tr>
<tr>
<td>DbEnv::set_memory_max(), DbEnv::get_memory_max()</td>
<td>Set/get maximum memory allocation</td>
</tr>
<tr>
<td>DbEnv::set_metadata_dir(), DbEnv::get_metadata_dir()</td>
<td>Set/get the directory containing environment metadata</td>
</tr>
<tr>
<td>DbEnv::set_message_stream()</td>
<td>Set C++ ostream used for informational messages</td>
</tr>
<tr>
<td>DbEnv::set_msgcallback()</td>
<td>Set informational message callback</td>
</tr>
<tr>
<td>DbEnv::set_msgfile(), DbEnv::get_msgfile()</td>
<td>Set/get informational message FILE</td>
</tr>
<tr>
<td>DbEnv::set_msgprefix(), DbEnv::get_msgprefix()</td>
<td>Set/get informational message prefix</td>
</tr>
<tr>
<td>DbEnv::set_shm_key(), DbEnv::get_shm_key()</td>
<td>Set/get system memory shared segment ID</td>
</tr>
<tr>
<td>DbEnv::set_thread_count(), DbEnv::get_thread_count()</td>
<td>Set/get approximate thread count</td>
</tr>
<tr>
<td>DbEnv::set_thread_id()</td>
<td>Set thread of control ID function</td>
</tr>
<tr>
<td>DbEnv::set_thread_id_string()</td>
<td>Set thread of control ID format function</td>
</tr>
<tr>
<td>DbEnv::set_timeout(), DbEnv::get_timeout()</td>
<td>Set/get lock and transaction timeout</td>
</tr>
<tr>
<td>DbEnv::set_tmp_dir(), DbEnv::get_tmp_dir()</td>
<td>Set/get the environment temporary file directory</td>
</tr>
<tr>
<td>DbEnv::set_verbose(), DbEnv::get_verbose()</td>
<td>Set/get verbose messages</td>
</tr>
<tr>
<td>DbEnv::set_cachesize(), DbEnv::get_cachesize()</td>
<td>Set/get the environment cache size</td>
</tr>
</tbody>
</table>
DbEnv::add_data_dir()

```cpp
#include <db_cxx.h>

int
DbEnv::add_data_dir(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 75) 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. Note that if you use this method for your application, and you also want to use the db_recover (page 748) or db_archive (page 722) utilities, then you should create a DB_CONFIG file and set the “add_data_dir” parameter in it.

The DbEnv::add_data_dir() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::add_data_dir() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::add_data_dir() must be consistent with the existing environment or corruption can occur.

The DbEnv::add_data_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dir**

The *dir* parameter is a directory to be used as a location for database files. This directory must currently exist at environment open time.

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 DbEnv::add_data_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::backup()

```c
#include <db_cxx.h>

int DbEnv::backup(const char *target, u_int32_t flags);
```

The DbEnv::backup() method performs a hot backup of the open environment. All files used by the environment are backed up, so long as the normal rules for file placement are followed. For information on how files are normally placed relative to the environment directory, see Berkeley DB File Naming in the Berkeley DB Programmer's Reference Guide.

By default, data directories and the log directory specified relative to the home directory will be recreated relative to the target directory. If absolute path names are used, then specify DB_BACKUP_SINGLE_DIR to the flags parameter.

This method provides the same functionality as the db_hotbackup (page 734) utility. However, this method does not perform the housekeeping actions performed by the db_hotbackup utility. In particular, you may want to run a checkpoint before calling this method. To run a checkpoint, use the DbEnv::txn_checkpoint() (page 678) method. For more information on checkpoints, see Checkpoints in the Berkeley DB Programmer’s Reference Guide.

To back up a single database file contained within the environment, use the DbEnv::dbbackup() (page 239) method.

This method's default behavior can be changed by setting backup callbacks. See DbEnv::set_backup_callbacks() (page 298) for more information. Additional tuning parameters can also be set using the DbEnv::set_backup_config() (page 301) method.

The DbEnv::backup() method may only be called after the environment handle has been opened.

The DbEnv::backup() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**target**

Identifies the directory in which the back up will be placed. Any subdirectories required to contain the backup must be placed relative to this directory. Note that if the backup callbacks are set, then the value specified to this parameter is passed on to the open_func() callback. If this parameter is NULL, then the target must be specified to the open_func() callback.

This directory, and any required subdirectories, will be created for you if you specify the DB_CREATE flag on the call to this method. Otherwise, if the target does not exist, this method exits with an ENOENT error return.

**flags**

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

  Before performing the backup, first remove all files from the target backup directory tree.

• **DB_BACKUP_FILES**

  Back up all ordinary files that might exist in the environment, and the environment's subdirectories.

• **DB_BACKUP_NO_LOGS**

  Back up only the *.db files. Do not backup the log files.

• **DB_BACKUP_SINGLE_DIR**

  Regardless of the directory structure used by the source environment, place all back up files in the single directory identified by the target parameter. Use this option if absolute path names to your environment directory and the files within that directory are required by your application.

• **DB_BACKUP_UPDATE**

  Perform an incremental back up, instead of a full back up. When this option is specified, only log files are copied to the target directory.

• **DB_CREATE**

  If the target directory does not exist, create it and any required subdirectories.

• **DB_EXCL**

  Return an EEXIST error if a target backup file already exists.

• **DB_VERB_BACKUP**

  Run in verbose mode, listing operations as they are completed.

**Errors**

The DbEnv::backup() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EEXIST**

DB_EXCL was specified for the flags parameter, and an existing target file was discovered when attempting to back up a source file.

**ENOENT**

The target directory does not exist and DB_CREATE was not specified for the flags parameter.

**EINVAL**

An invalid flag value or parameter was specified.
Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::close()

```c++
#include <db_cxx.h>

DbEnv::close(u_int32_t flags);
```

The `DbEnv::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 `DbEnv::close()` method, all open `Db` handles and `Dbc` 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 `DbEnv::close()` method, by default, the database is not synchronized and is similar to calling the `Db::close(DB_NOSYNC)` method. This is to avoid unnecessary 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 `DbEnv::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 `DbEnv` 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 `DbTxn`, `DbLogc` and `DbMpoolFile` handles.

Where the environment was initialized with the `DB_INIT_LOCK` flag, calling `DbEnv::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 `DbEnv::lock_vec()` (page 412) call.

Where the environment was initialized with the `DB_INIT_MPOOL` flag, calling `DbEnv::close()` implies calls to `DbMpoolFile::close()` (page 494) for any remaining open files in the memory pool that were returned to this process by calls to `DbMpoolFile::open()` (page 498). It does not imply a call to `DbMpoolFile::sync()` (page 502) for those files.

Where the environment was initialized with the `DB_INIT_TXN` flag, calling `DbEnv::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 `DbEnv::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 `DbEnv::log_cursor()` method, calling `DbEnv::close()` does not imply closing those cursors.

In multithreaded applications, only a single thread may call the `DbEnv::close()` method.

After `DbEnv::close()` has been called, regardless of its return, the Berkeley DB environment handle may not be accessed again.

The `DbEnv::close()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

### Parameters

**flags**

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

- **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.

- **DB_FORCESYNCENV**
  
  When closing the environment, flush memory mapped environment regions to disk. Specifying this flag may help prevent loss of updates when `__db.00*` files are on NFS storage. However, there is a risk that this flag will significantly slow down this method call.

### Class

DbEnv

### See Also

*Database Environments and Related Methods (page 228)*
DbEnv

```
#include <db_cxx.h>

class DbEnv {
public:
    DbEnv(u_int32 flags);
    ~DbEnv();

    DB_ENV *DbEnv::get_DB_ENV();
    const DB_ENV *DbEnv::get_const_DB_ENV() const;
    static DbEnv *DbEnv::get_DbEnv(DB_ENV *dbenv);
    static const DbEnv *DbEnv::get_const_DbEnv(const DB_ENV *dbenv);
    ...
};
```

The DbEnv 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 DbEnv handle are used to configure the environment as well as to operate on subsystems and databases in the environment.

DbEnv handles are free-threaded if the DB_THREAD flag is specified to the DbEnv::open() (page 286) method when the environment is opened. The DbEnv handle should not be closed while any other handle remains open that is using it as a reference (for example, Db or DbTxn). Once either the DbEnv::close() (page 235) or DbEnv::remove() (page 292) methods are called, the handle may not be accessed again, regardless of the method's return.

The constructor creates the DbEnv object. The constructor allocates memory internally; calling the DbEnv::close() (page 235) or DbEnv::remove() (page 292) methods will free that memory.

Before the handle may be used, you must open it using the DbEnv::open() (page 286) method.

The flags parameter must be set to 0.

- DB_CXX_NO_EXCEPTIONS

The Berkeley DB C++ API supports two different error behaviors. By default, whenever an error occurs, an exception is thrown that encapsulates the error information. This generally allows for cleaner logic for transaction processing because a try block can surround a single transaction. However, if DB_CXX_NO_EXCEPTIONS is specified, exceptions are not thrown; instead, each individual function returns an error code.

Each DbEnv object has an associated DB_ENV structure, which is used by the underlying implementation of Berkeley DB and its C-language API. The DbEnv::get_DB_ENV() method returns a pointer to this struct. Given a const DbEnv object, DbEnv::get_const_DB_ENV() returns a const pointer to the same struct.
Given a DB_ENV struct, the DbEnv::get_DbEnv() method returns the corresponding DbEnv object, if there is one. If the DB_ENV struct was not associated with a DbEnv (that is, it was not returned from a call to DbEnv::get_DB_ENV()), then the result of DbEnv::get_DbEnv() is undefined. Given a const DB_ENV struct, DbEnv::get_const_Db_Env() returns the associated const DbEnv object, if there is one.

These methods may be useful for Berkeley DB applications including both C and C++ language software. It should not be necessary to use these calls in a purely C++ application.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::dbbackup()

```cpp
#include <db_cxx.h>

int DbEnv::dbbackup(const char *dbfile, const char *target,
                     u_int32_t flags);
```

The DbEnv::dbbackup() method performs a hot backup of a single database file contained within the environment.

To back up an entire environment, use the DbEnv::backup() (page 232) method.

This method's default behavior can be changed by setting backup callbacks. See DbEnv::set_backup_callbacks() (page 298) for more information. Additional tuning parameters can also be set using the DbEnv::set_backup_config() (page 301) method.

The DbEnv::dbbackup() method may only be called after the environment handle has been opened.

The DbEnv::dbbackup() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dbfile**

Identifies the database file that you want to back up.

**target**

Identifies the directory in which the back up will be placed. This target must exist; otherwise this method exits with an ENOENT error return.

Note that if the backup callbacks are set, then the value specified to this parameter is passed on to the open_func() callback. If this parameter is NULL, then the target must be specified directly to the open_func() callback.

**flags**

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

- **DB_EXCL**

  Return an EEXIST error if a target backup file already exists.

**Errors**

The DbEnv::dbbackup() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
**EEXIST**

DB_EXCL was specified for the flags parameter, and an existing target file was discovered when attempting to back up a source file.

**ENOENT**

The target directory does not exist.

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::dbremove()

    #include <db_cxx.h>

    int
    DbEnv::dbremove(DbTxn *txnid,
                    const char *file, const char *database, u_int32_t flags);

The DbEnv::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 DbEnv::dbremove() method is not atomic in a sliced environment. If the operation fails, it is possible for some, but not all, of the sliced databases to still exist.

The DbEnv::dbremove() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

DbEnv::dbremove() is affected by any database directory specified using the DbEnv::add_data_dir() (page 230) method, or by setting the add_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 DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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 DbEnv::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 DbEnv::dbremove() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**EINVAL**

If the method was called before DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**ENOENT**

The file or directory does not exist.

**DB_META_CHKSUM_FAIL**

Checksum mismatch detected on a database metadata page. Either the database is corrupted or the file is not a Berkeley DB database file.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
The `DbEnv::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 `DbEnv::dbrename()` method is called.

The `DbEnv::dbrename()` method is not atomic in a sliced environment. If the operation fails, it is possible for the names of the various sliced databases to be in an inconsistent state.

The `DbEnv::dbrename()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

`DbEnv::dbrename()` is affected by any database directory specified using the `DbEnv::add_data_dir()` (page 230) method, or by setting the `add_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 `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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 DbEnv::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 DbEnv::dbrename() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

EINVAL
If the method was called before DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

ENOENT
The file or directory does not exist.

DB_META_CHKSUM_FAIL
Checksum mismatch detected on a database metadata page. Either the database is corrupted or the file is not a Berkeley DB database file.
Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::err()

#include <db_cxx.h>

DbEnv::err(int error, const char *fmt, ...);
DbEnv::errx(const char *fmt, ...);

The DbEnv::err(), DbEnv::errx(), Db::err() (page 26) and Db::errx() methods provide error-messaging functionality for applications written using the Berkeley DB library.

The Db::err() (page 26) and DbEnv::err() (page 246) methods constructs an error message consisting of the following elements:

- **An optional prefix string**
  
  If no error callback function has been set using the DbEnv::set_errcall() (page 315) method, any prefix string specified using the DbEnv::set_errpfx() (page 320) 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 DbEnv::strerror() (page 361) method.

This constructed error message is then handled as follows:

- If an error callback function has been set (see Db::set_errcall() (page 109) and DbEnv::set_errcall() (page 315)), that function is called with two parameters: any prefix string specified (see Db::set_errpfx() (page 114) and DbEnv::set_errpfx() (page 320)) and the error message.

- If a C library FILE * has been set (see Db::set_errfile() (page 111) and DbEnv::set_errfile() (page 317)), the error message is written to that output stream.

- If a C++ ostream has been set (see DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113)), the error message is written to that stream.

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

**Note**

The total length of the message written by this method, including the prefix string, can be no longer than 4096.
Parameters

**error**

The *error* parameter is the error value for which the `DbEnv::err()` and `Db::err()` (page 26) methods will display an explanatory string.

**fmt**

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

Class

`DbEnv`

See Also

*Database Environments and Related Methods (page 228)*
The DbEnv::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 DbEnv::failchk() method is used in conjunction with the DbEnv::set_thread_count() (page 346), DbEnv::set_isalive() (page 332) and DbEnv::set_thread_id() (page 348) methods. Before calling the failchk() method, applications must:

1. Configure their database using the DbEnv::set_thread_count() (page 346) method.
2. Establish an is_alive() function and invoke DbEnv::set_isalive() (page 332) with that function as the is_alive parameter.
3. Establish a thread_id function and invoke DbEnv::set_thread_id() (page 348) with that function as the thread_id parameter.

If any of these methods are omitted, a program may be unable to allocate a thread control block. 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 DbEnv::failchk() method along with its associated methods before running the utility.

If DbEnv::failchk() determines a thread of control exited while holding database read locks, it will release those locks. If DbEnv::failchk() determines a thread of control exited with an unresolved transaction, the transaction will be aborted. In either of these cases, DbEnv::failchk() will return 0 and the application may continue to use the database environment.

In either of these cases, the DbEnv::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 DbEnv::set_msgfile() (page 342) method for more information), or passed to an application callback function (see the DbEnv::set_msgcall() (page 340) method for more information).

If DbEnv::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*. 

```
#include <db_cxx.h>

int
DbEnv::failchk(u_int32_t flags);
```
In multiprocess applications, it is recommended that the `DbEnv` handle used to invoke the `DbEnv::failchk()` method not be shared and therefore not free-threaded.

The `DbEnv::failchk()` method may not be called by the application before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::failchk()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**flags**

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

**Errors**

The `DbEnv::failchk()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

Database Environments and Related Methods (page 228)
# DbEnv::fileid_reset()

```cpp
#include <db_cxx.h>

int DbEnv::fileid_reset(const char *file, u_int32_t flags);
```

The `DbEnv::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 `DbEnv::fileid_reset()` method creates new ID strings for all of the databases in the physical file.

The `DbEnv::fileid_reset()` method modifies the physical file, in-place. Applications should not reset IDs in files that are currently in use.

The `DbEnv::fileid_reset()` method may be called at any time during the life of the application.

The `DbEnv::fileid_reset()` method is a non-atomic operation. If the operation fails, it is possible for the copied database to be in an inconsistent state.

The `DbEnv::fileid_reset()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::fileid_reset()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.
Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::full_version()

```
#include <db_cxx.h>

static char *
DbEnv::full_version(int *family, int *release, int *major, int *minor,
                   int *patch);
```

The DbEnv::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**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::get_create_dir()

```cpp
#include <db_cxx.h>

int
DbEnv::get_create_dir(const char **dirp);
```

The `DbEnv::get_create_dir()` method returns a pointer to the name of the directory to create databases in.

The `DbEnv::get_create_dir()` method may be called at any time during the life of the application.

The `DbEnv::get_create_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **dirp**

  The `DbEnv::get_create_dir()` method returns a pointer to the name of the directory in `dirp`.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::get_data_dirs()

```cpp
#include <db_cxx.h>

int DbEnv::get_data_dirs(const char ***dirpp);
```

The DbEnv::get_data_dirs() method returns the NULL-terminated array of directories.

The DbEnv::get_data_dirs() method may be called at any time during the life of the application.

The DbEnv::get_data_dirs() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dirpp**

The DbEnv::get_data_dirs() method returns a reference to the NULL-terminated array of directories in `dirpp`.

**Class**

`DbEnv`

**See Also**

*Database Environments and Related Methods (page 228)*
DbEnv::get_encrypt_flags()

```c
#include <db_cxx.h>

int DbEnv::get_encrypt_flags(u_int32_t *flagsp);
```

The DbEnv::get_encrypt_flags() method returns the encryption flags.

The DbEnv::get_encrypt_flags() method may be called at any time during the life of the application.

The DbEnv::get_encrypt_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

flagsp

The DbEnv::get_encrypt_flags() method returns the encryption flags in flagsp.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
The DbEnv Handle

### Db::get_env()

```cpp
#include <db_cxx.h>

DbEnv *
Db::get_env();
```

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))
DbEnv::get_errfile()

```cpp
#include <db_cxx.h>

void
DbEnv::get_errfile(FILE **errfilep);
```

The `DbEnv::get_errfile()` method returns the FILE * used for displaying additional Berkeley DB error messages. This C library is set using the `DbEnv::set_errfile()` method.

The `DbEnv::get_errfile()` method may be called at any time during the life of the application.

**Parameters**

- **errfilep**

  The `DbEnv::get_errfile()` method returns the FILE * in `errfilep`.

**Class**

`DbEnv`

**See Also**

- Database Environments and Related Methods (page 228)
DbEnv::get_errpfx()

```cpp
#include <db_cxx.h>

void
DbEnv::get_errpfx(const char **errpfxp);
```

The DbEnv::get_errpfx() method returns the error prefix that appears before error messages issued by Berkeley DB. This error prefix is set using the DbEnv::set_errpfx() (page 320) method.

The DbEnv::get_errpfx() method may be called at any time during the life of the application.

**Parameters**

`errpfxp`

The DbEnv::get_errpfx() method returns a reference to the error prefix in `errpfxp`.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::get_backup_callbacks()

#include <db_cxx.h>

static int (**open_func)(DB_ENV *, const char *dbname, const char *target, void **handle),
static int (**write_func)(DB_ENV *, u_int32_t offset_gbytes, u_int32_t offset_bytes, u_int32_t size, u_int8_t *buf, void *handle),
static int (**close_func)(DB_ENV *, const char *dbname, void *handle))

The DbEnv::get_backup_callbacks() method retrieves the three callback functions which can be used by the DbEnv::backup() (page 232) or DbEnv::dbbackup() (page 239) methods to override their default behavior. These callbacks are configured using the DbEnv::set_backup_callbacks() (page 298) method.

The DbEnv::get_backup_callbacks() method may be called at any time during the life of the application.

The DbEnv::get_backup_callbacks() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

open_func
The open_func parameter is the function used when a target location is opened during a backup.

write_func
The write_func parameter is the function used to write data during a backup.

close_func
The close_func parameter is the function used when ending a backup and closing a backup target.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228), DbEnv::set_backup_callbacks() (page 298), DbEnv::backup() (page 232), and DbEnv::dbbackup() (page 239).
DbEnv::get_backup_config()

```c
#include <db_cxx.h>

DB_ENV->get_backup_config(DB_BACKUP_CONFIG option, u_int32_t *valuep);
```

The `DbEnv::get_backup_config()` method retrieves the value set for hot backup tuning parameters. See the `DbEnv::backup()` (page 232) and `DbEnv::dbbackup()` (page 239) methods for a description of the hot backup APIs. These tuning parameters can be set using the `DbEnv::set_backup_config()` (page 301) method.

The `DbEnv::get_backup_config()` method may be called at any time during the life of the application.

The `DbEnv::get_backup_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**option**

The `option` parameter identifies the backup parameter to be retrieved. It must be one of the following:

- `DB_BACKUP_WRITE_DIRECT`
  
  Turning this on causes direct I/O to be used when writing pages to the disk.

- `DB_BACKUP_READ_COUNT`
  
  Configures the number of pages to read before pausing.

- `DB_BACKUP_READ_SLEEP`
  
  Configures the number of microseconds to sleep between batches of reads.

- `DB_BACKUP_SIZE`
  
  Configures the size of the buffer, in megabytes, to read from the database.

**valuep**

The `valuep` parameter references memory into which is copied the current value of the backup tuning parameter identified by the `option` parameter.

**Class**

`DbEnv`

**See Also**

`Database Environments and Related Methods (page 228)`, `DbEnv::set_backup_config()` (page 301), `DbEnv::backup()` (page 232), `DbEnv::dbbackup()` (page 239)
DbEnv::get_flags()

```cpp
#include <db_cxx.h>

int DbEnv::get_flags(u_int32_t *flagsp)
```

The `DbEnv::get_flags()` method returns the configuration flags set for a `DbEnv` handle. These flags are set using the `DbEnv::set_flags()` (page 323) method.

The `DbEnv::get_flags()` method may be called at any time during the life of the application.

The `DbEnv::get_flags()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **flagsp**

  The `DbEnv::get_flags()` method returns the configuration flags in `flagsp`.

**Class**

`DbEnv`

**See Also**

- Database Environments and Related Methods (page 228)
DbEnv::get_home()

#include <db_cxx.h>

int DbEnv::get_home(const char **homep);

The DbEnv::get_home() method returns the database environment home directory. This directory is normally identified when the DbEnv::open() (page 286) method is called.

The DbEnv::get_home() method may be called at any time during the life of the application.

The DbEnv::get_home() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
### DbEnv::get_intermediate_dir_mode()

```cpp
#include <db_cxx.h>

int DbEnv::get_intermediate_dir_mode(u_int32_t *modep);
```

The `DbEnv::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 `DbEnv::set_intermediate_dir_mode()` (page 330) method is called.

The `DbEnv::get_intermediate_dir_mode()` method may be called at any time during the life of the application.

The `DbEnv::get_intermediate_dir_mode()` method returns a non-zero error value on failure and 0 on success.

**Parameters**

- **modep**

  The `DbEnv::get_intermediate_dir_mode()` method returns a reference to the intermediate directory permissions in `modep`.

**Class**

- `DbEnv`

**See Also**

- [Database Environments and Related Methods (page 228)](#)
DbEnv::get_memory_init()

```c
#include <db_cxx.h>

int
DbEnv::get_memory_init(DB_MEM_CONFIG type, u_int32_t *countp);
```

The `DbEnv::get_memory_init()` method returns the number of objects to allocate and initialize when an environment is created. The count is returned for a specific named structure. The count for each structure is set using the `DbEnv::set_memory_init()` method.

The `DbEnv::get_memory_init()` method may be called at any time during the life of the application.

The `DbEnv::get_memory_init()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**type**

The `struct` parameter identifies the structure for which you want an object count returned. It must be one of the following values:

- **DB_MEM_LOCK**
  
  Initialize locks. A thread uses this structure to lock a page (or record for the QUEUE access method) and hold it to the end of a transaction.

- **DB_MEM_LOCKOBJECT**
  
  Initialize lock objects. For each page (or record) which is locked in the system, a lock object will be allocated.

- **DB_MEM_LOCKER**
  
  Initialize lockers. Each thread which is active in a transactional environment will use a locker structure either for each transaction which is active, or for each non-transactional cursor that is active.

- **DB_MEM_LOGID**
  
  Initialize the log fileid structures. For each database handle which is opened for writing in a transactional environment, a log fileid structure is used.

- **DB_MEM_TRANSACTION**
  
  Initialize transaction structures. Each active transaction uses a transaction structure until it either commits or aborts.

- **DB_MEM_THREAD**
Initialize thread identification structures. If thread tracking is enabled then each active thread will use a structure. Note that since a thread does not signal the BDB library that it will no longer be making calls, unused structures may accumulate until a cleanup is triggered either using a high water mark or by running `DbEnv::failchk()` (page 248).

**countp**

The `countp` parameter references memory into which the object count for the specified structure is copied.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::get_memory_max()

```c
#include <db_cxx.h>

int DbEnv::get_memory_max(u_int32_t *gbytesp, u_int32_t *bytesp);
```

The `DbEnv::get_memory_max()` method returns the maximum amount of memory to be used by shared structures other than mutexes and the page cache (memory pool). This value is set using the `DbEnv::set_memory_max()` method.

The `DbEnv::get_memory_max()` method may be called at any time during the life of the application.

The `DbEnv::get_memory_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **gbytesp**
  
  The `gbytesp` parameter references memory into which is copied the maximum number of gigabytes of memory that can be allocated.

- **bytesp**
  
  The `bytesp` parameter references memory into which is copied the additional bytes of memory that can be allocated.

- **sizep**
  
  The `sizep` parameter references memory into which is copied the maximum number of bytes to be allocated.

**Class**

`DbEnv`

**See Also**

- Database Environments and Related Methods (page 228)
DbEnv::get_metadata_dir()

```
#include <db_cxx.h>

int DbEnv::get_metadata_dir(const char **dirp);
```

The `DbEnv::get_metadata_dir()` method returns the directory where persistent metadata is stored. This location can be set using the `DbEnv::set_metadata_dir()` method.

The `DbEnv::get_metadata_dir()` directory may be called at any time during the life of the application.

The `DbEnv::get_metadata_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`dirp`

The `dirp` parameter references memory into which is copied the directory which contains persistent metadata files.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), `DbEnv::set_metadata_dir()` (page 338)
DbEnv::get_msgfile()

```cpp
#include <db_cxx.h>

void
DbEnv::get_msgfile(FILE **msgfilep);
```

The DbEnv::get_msgfile() method returns the FILE * used for displaying messages. This is set using the DbEnv::set_msgfile() (page 342) method.

The DbEnv::get_msgfile() method may be called at any time during the life of the application.

**Parameters**

- **msgfilep**

  The DbEnv::get_msgfile() method returns the FILE * in `msgfilep`.

**Class**

(DbEnv)

**See Also**

Database Environments and Related Methods (page 228), DbEnv::set_msgfile() (page 342)
DbEnv::get_msgpfx()

```cpp
#include <db_cxx.h>

void
DbEnv::get_msgpfx(const char **msgpfxp);
```

The `DbEnv::get_msgpfx()` method returns the information prefix that appears before informational messages issued by Berkeley DB. This prefix is set using the `DbEnv::set_msgpfx()` method.

The `DbEnv::get_msgpfx()` method may be called at any time during the life of the application.

**Parameters**

- `msgpfxp`

  The `DbEnv::get_msgpfx()` method returns a reference to the informational prefix in `msgpfxp`.

**Class**

- `DbEnv`

**See Also**

- [Database Environments and Related Methods (page 228)](#)
The DbEnv Handle

# include <db_cxx.h>

int DbEnv::get_open_flags(u_int32_t *flagsp);

The DbEnv::get_open_flags() method returns the open method flags originally used to create the database environment.

The DbEnv::get_open_flags() method may not be called before the DbEnv::open() method is called.

The DbEnv::get_open_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

flagsp

The DbEnv::get_open_flags() method returns the open method flags originally used to create the database environment in flagsp.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228), DbEnv::open() (page 286)
DbEnv::get_shm_key()

```c
#include <db_cxx.h>

int DbEnv::get_shm_key(long *shm_keyp);
```

The `DbEnv::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 `DbEnv::set_shm_key()` (page 344) method.

The `DbEnv::get_shm_key()` method may be called at any time during the life of the application.

The `DbEnv::get_shm_key()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**shm_keyp**

The `DbEnv::get_shm_key()` method returns the base segment ID in `shm_keyp`.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), `DbEnv::set_shm_key()` (page 344)
**DbEnv::get_slices()**

```cpp
#include <db_cxx.h>

int DbEnv::get_slices(DbEnv ***slicepp);
```

The `DbEnv::get_slices()` method returns a NULL-terminated array of slice environments. Each element in the array represents one slice.

The `DbEnv::get_slices()` method may only be called after the environment has been opened.

The `DbEnv::get_slices()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **slicepp**

  The `DbEnv::get_slices()` method returns a reference to a NULL-terminated array of slice environments in `slicepp`.

**Class**

DbEnv

**See Also**

- Database Environments and Related Methods (page 228)
DbEnv::get_slice_count()

```c
#include <db_cxx.h>

int DbEnv::get_slice_count(u_int32_t *nslicesp);
```

The `DbEnv::get_slice_count()` method returns the number of slices configured for the environment. The number of slices is configured using the `set_slice_count` DB_CONFIG parameter.

The `DbEnv::get_slice_count()` method may only be called after the environment has been opened.

The `DbEnv::get_slice_count()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**nslicep**

The `DbEnv::get_slice_count()` method returns the slice count in `nslicep`.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::get_thread_count()

```
#include <db_cxx.h>

int DbEnv::get_thread_count(u_int32_t *countp);
```

The DbEnv::get_thread_count() method returns the thread count as set by the 
DbEnv::set_thread_count() (page 346) method.

The DbEnv::get_thread_count() method may be called at any time during the life of the 
application.

The DbEnv::get_thread_count() method either returns a non-zero error value or throws an 
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **countp**

  The DbEnv::get_thread_count() method returns the thread count in countp.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), DbEnv::set_thread_count() (page
346)
DbEnv::get_timeout()

```cpp
#include <db_cxx.h>

int DbEnv::get_timeout(db_timeout_t *timeoutp, u_int32_t flag);
```

The `DbEnv::get_timeout()` method returns a value, in microseconds, representing either lock or transaction timeouts. These values are set using the `DbEnv::set_timeout()` (page 352) method.

The `DbEnv::get_timeout()` method may be called at any time during the life of the application.

The `DbEnv::get_timeout()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_REG_TIMEOUT`
  
  Return the timeout value for how long to wait for processes to exit the environment before recovery is started. This flag only has meaning when the `DbEnv::open()` (page 286) method was called with the `DB_REGISTER` flag and recovery must be performed.

- `DB_SET_TXN_TIMEOUT`
  
  Return the timeout value for transactions in this database environment.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), `DbEnv::set_timeout()` (page 352)
DbEnv::get_tmp_dir()

```cpp
#include <db_cxx.h>

int
DbEnv::get_tmp_dir(const char **dirp);
```

The DbEnv::get_tmp_dir() method returns the database environment temporary file directory.

The DbEnv::get_tmp_dir() method may be called at any time during the life of the application.

The DbEnv::get_tmp_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*dirp*

The DbEnv::get_tmp_dir() method returns a reference to the database environment temporary file directory in `dirp`.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), DbEnv::set_tmp_dir() (page 355)
DbEnv::get_verbose()

```
#include <db_cxx.h>

int
DbEnv::get_verbose(u_int32_t which, int *onoffp);
```

The DbEnv::get_verbose() method returns whether the specified which parameter is currently set or not. These parameters are set using the DbEnv::set_verbose() (page 357) method.

The DbEnv::get_verbose() method may be called at any time during the life of the application.

The DbEnv::get_verbose() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::open() (page 286) 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_REPLEASE
  Display detailed information about replication master leases.

• DB_VERB_REPMISC
  Display detailed information about general replication processing not covered by the other
  DB_VERB_REP_* values.

• DB_VERB_REPMSGS
  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**

DbType

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::log_verify()

#include <db_cxx.h>

int DbEnv::log_verify(DB_ENV *dbenv, const DB_LOG_VERIFY_CONFIG *config);

The DbEnv::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 DbEnv::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 DbEnv::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 DbEnv::log_verify() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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

struct __db_logvrfy_config {
  int continue_after_fail, verbose;
  u_int32_t cachesize;
  const char *temp_envhome;
  const char *dbfile, *dbname;
  DB_LSN start_lsn, end_lsn;
  time_t start_time, end_time;
};

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.
cachesize

The `cachesize` parameter specifies the size of the cache of the temporary internal environment in bytes.

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.

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.

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 `DbEnv::log_verify()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL or DB_LOG_VERIFY_BAD.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::lsn_reset()

#include <db_cxx.h>

int DbEnv::lsn_reset(const char *file, u_int32_t flags);

The DbEnv::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 DbEnv::lsn_reset() on a database in a new environment when the database LSNs still reflect the old environment.

The DbEnv::lsn_reset() method modifies the physical file, in-place. Applications should not reset LSNs in files that are currently in use.

The DbEnv::lsn_reset() method is a non-atomic operation. If the operation fails, it is possible for the database to be in an inconsistent state.

The DbEnv::lsn_reset() method may be called at any time during the life of the application.

The DbEnv::lsn_reset() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::lsn_reset() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

An invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::msg()

```cpp
#include <db_cxx.h>

DbEnv::msg(const char *fmt, ...);
```

The DbEnv::msg() and Db::msg() (page 73) methods provide informational messaging functionality for applications written using the Berkeley DB library.

The Db::msg() (page 73) and DbEnv::msg() (page 284) methods constructs an informational message consisting of the following elements:

- An optional prefix string
  
  If no informational callback function has been set using the DbEnv::set_msgcall() (page 340) method, any prefix string specified using the DbEnv::set_msgpfx() (page 343) method.

- 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.

This constructed informational message is then handled as follows:

- If an informational callback function has been set (see Db::set_msgcall() (page 135) and DbEnv::set_msgcall() (page 340)), that function is called with two parameters: any prefix string specified (see Db::set_msgpfx() (page 138) and DbEnv::set_msgpfx() (page 343)) and the informational message.

- If a C library FILE * has been set (see Db::set_msgfile() (page 137) and DbEnv::set_msgfile() (page 342)), the informational message is written to that output stream.

- If a C++ ostream has been set (see DbEnv::set_message_stream() (page 339) and Db::set_message_stream() (page 134)), the informational message is written to that stream.

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

**Note**

The total length of the message written by this method, including the prefix string, can be no longer than 4096.

**Parameters**

`fmt`

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

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::open()

```c
#include <db_cxx.h>

int
DbEnv::open(const char *db_home, u_int32_t flags, int mode);
```

The DbEnv::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 DbEnv::open() method method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success. If DbEnv::open() fails, the DbEnv::close() (page 235) method must be called to discard the DbEnv handle.

**Warning**

Using environments with some journaling filesystems might result in log file corruption. This can occur if the operating system experiences an unclean shutdown when a log file is being created. Please see Using Recovery on Journaling Filesystems in the Berkeley DB Programmer's Reference Guide for more information.

**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 `DbEnv::lock_detect()` (page 396) 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.

  You can also specify this flag in the DB_CONFIG configuration file. The syntax is a single line with the string "set_open_flags", one or more whitespace characters, the string "DB_INIT_REP", 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_open_flags DB_INIT_REP on". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

- **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 DbEnv::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 DbEnv::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. If the operating systems does not support the `mlock()` system call, then this flag has no effect.

- **DB_FAILCHK**

Internally call the `DbEnv::failchk()` method as part of opening the environment. When DB_FAILCHK is specified, a check is made to ensure all `DbEnv::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 `DbEnv::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 `DbEnv::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 `DbEnv::failchk()` as the last step of opening the environment. If `DbEnv::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.

**Note**

Use of this flag means that the environment can only be accessed by one environment handle. The environment cannot be accessed by multiple processes. This is true even if one of those processes is one of the the Berkeley DB utilities. (For example, `db_archive`, `db_checkpoint` or `db_stat`.) Nor can a single process open multiple handles to the environment.

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.

See Shared Memory Regions for more information.

You can also specify this flag in the `DB_CONFIG` configuration file. The syntax is a single line with the string "set_open_flags", one or more whitespace characters, the string "DB_PRIVATE", 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_open_flags DB_PRIVATE”
or "set_open_flags DB_PRIVATE on". Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

- **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 **DbEnv** handle returned by **DbEnv::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 **DbEnv** handle will be concurrently used by more than one thread in the process, or if any **Db** handles opened in the scope of the **DbEnv** handle will be concurrently used by more than one thread in the process.

  If this flag is specified, then any database opened using this environment handle will also be free-threaded.

  Be aware that enabling this flag will serialize calls to DB when using the handle across threads. If concurrent scaling is important to your application we recommend opening separate handles for each thread (and not specifying this flag), rather than sharing handles between threads.

  This flag is required when using the Replication Manager.

  You can also specify this flag in the **DB_CONFIG** configuration file. The syntax is a single line with the string "set_open_flags", one or more whitespace characters, the string "DB_THREAD", 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_open_flags DB_THREAD" or "set_open_flags DB_THREAD on". Because the **DB_CONFIG** file is read when the database environment is opened, it will silently overrule configuration done before that time.

**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 `DbEnv::open()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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.

**ENOENT**

The file or directory does not exist.

Class

**DbEnv**

See Also

Database Environments and Related Methods (page 228)
DbEnv::remove()

```cpp
#include <db_cxx.h>

int
DbEnv::remove(const char *db_home, u_int32_t flags);
```

The DbEnv::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 DbEnv::open() (page 286) without calling DbEnv::close() (page 235) (that is, there are processes currently using the environment), DbEnv::remove() will fail without further action unless the DB_FORCE flag is set, in which case DbEnv::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 DbEnv::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 DbEnv::remove() as part of application shut down to free up system resources. For example, if the DB_SYSTEM_MEM flag was specified to DbEnv::open() (page 286), it may be useful to call DbEnv::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 DbEnv::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 DbEnv::remove() method.

A DbEnv handle that has already been used to open an environment should not be used to call the DbEnv::remove() method; a new DbEnv handle should be created for that purpose.

After DbEnv::remove() has been called, regardless of its return, the Berkeley DB environment handle may not be accessed again.

The DbEnv::remove() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::remove()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EBUSY**
  
  The shared memory region was in use and the force flag was not set.

Class

`DbEnv`

See Also

*Database Environments and Related Methods (page 228)*


**DbEnv::set_alloc()**

```c
#include <db_cxx.h>

extern "C" {
    typedef void *(*db_malloc_fcn_type)(size_t);
    typedef void *(*db_realloc_fcn_type)(void *, size_t);
    typedef void *(*db_free_fcn_type)(void *);
};

int DbEnv::set_alloc(db_malloc_fcn_type app_malloc,
    db_realloc_fcn_type app_realloc,
    db_free_fcn_type app_free);
```

Set the allocation functions used by the DbEnv 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_BT_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 157), DbEnv::lock_stat() (page 404), DbEnv::log_archive() (page 423), DbEnv::log_stat() (page 437), DbEnv::memp_stat() (page 471), and DbEnv::txn_stat() (page 680). 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 DbEnv::set_alloc() and Db::set_alloc() (page 90) 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 DbEnv::set_alloc() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_alloc() method may not be called after the DbEnv::open() (page 286) method is called.

The DbEnv::set_alloc() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_alloc() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_app_dispatch()

```cpp
#include <db_cxx.h>

int
DbEnv::set_app_dispatch(int (*tx_recover)(DbEnv *dbenv,
             Dbt *log_rec, DbLsn *lsn, db_recops op));
```

Declare a function to be called during transaction abort and recovery to process application-specific log records.

The DbEnv::set_app_dispatch() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_app_dispatch() method may not be called after the DbEnv::open() method is called. If the database environment already exists when DbEnv::open() is called, the information specified to DbEnv::set_app_dispatch() must be consistent with the existing environment or corruption can occur.

The DbEnv::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 DbEnv::set_app_dispatch() method may fail and throw a **DbException** exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv, DbTxn

**See Also**

Transaction Subsystem and Related Methods (page 664)
The `DbEnv::set_backup_callbacks()` method configures three callback functions which can be used by the `DbEnv::backup()` (page 232) or `DbEnv::dbbackup()` (page 239) methods to override their default behavior. If one callback is configured, then all three callbacks must be configured. These callbacks are required if the `target` parameter is set to `NULL` for the `DbEnv::backup()` (page 232) or `DbEnv::dbbackup()` (page 239) methods.

The `DbEnv::set_backup_callbacks()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_backup_callbacks()` method may be called at any time during the life of the application.

The `DbEnv::set_backup_callbacks()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*open_func*

The `open_func` parameter is the function used when a target location is opened during a backup. This function should do whatever is necessary to prepare the backup destination for writing the data.

This function takes four parameters:

- `dbenv`
  
  The `dbenv` parameter is the enclosing database environment handle.

- `dbname`
  
  The `dbname` parameter is the name of the database being backed up.

- `target`
  
  The `target` parameter is the backup's directory destination.

- `handle`
The handle parameter references the handle (usually a file handle) to which the backup will be written.

write_func

The write_func parameter is the function used to write data during a backup. The function takes six parameters:

• dbenv
  The dbenv parameter is the enclosing database environment handle.

• offset_gbytes
  The offset_gbytes parameter specifies the number of gigabytes into the output handle where the data can be written. This value, plus the value specified on offset_bytes, indicates the offset within the output handle where the backup should begin.

• offset_bytes
  The offset_bytes parameter specifies the number of bytes into the output handle where the data can be located. This value, plus the value specified on offset_gbytes, indicates the offset within the output handle where the backup should begin.

• size
  The size parameter specifies the number of bytes to back up from the buffer.

• buf
  The buf parameter is the buffer which contains the data to be backed up.

• handle
  The handle parameter references the handle (usually a file handle) to which the backup will be written.

close_func

The close_func parameter is the function used when ending a backup and closing a backup target. The function takes three parameters:

• dbenv
  The dbenv parameter is the enclosing database environment handle.

• dbname
  The dbname parameter is the name of the database that has now been backed up.

• handle
The handle parameter references the handle (usually a file handle) to which the backup was written, and which now must be closed or otherwise discarded.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228), DbEnv::get_backup_callbacks() (page 259), DbEnv::backup() (page 232), and DbEnv::dbbackup() (page 239).
DbEnv::set_backup_config()

```c
#include <db_cxx.h>

DB_ENV->set_backup_config(DB_BACKUP_CONFIG option, u_int32_t value);
```

The `DbEnv::set_backup_config()` method configures tuning parameters for the hot backup APIs. See the `DbEnv::backup()` (page 232) and `DbEnv::dbbackup()` (page 239) methods for a description of the hot backup APIs.

The `DbEnv::set_backup_config()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_backup_config()` method may be called at any time during the life of the application.

The `DbEnv::set_backup_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **option**

  The `option` parameter identifies the backup parameter to be modified. It must be one of the following:

  - **DB_BACKUP_WRITE_DIRECT**
    
    Turning this on causes direct I/O to be used when writing pages to the disk. For some environments, direct I/O can provide faster write throughput, but usually it is slower because the OS buffer pool offers asynchronous activity.
    
    By default, this option is turned off.

  - **DB_BACKUP_READ_COUNT**
    
    Configures the number of pages to read before pausing. Increasing this value increases the amount of I/O the backup process performs for any given time interval. If your application is already heavily I/O bound, setting this value to a lower number may help to improve your overall data throughput by reducing the I/O demands placed on your system.
    
    By default, all pages are read without a pause.

  - **DB_BACKUP_READ_SLEEP**
    
    Configures the number of microseconds to sleep between batches of reads. Increasing this value decreases the amount of I/O the backup process performs for any given time interval. If your application is already heavily I/O bound, setting this value to a higher number may help to improve your overall data throughput by reducing the I/O demands placed on your system.
• **DB_BACKUP_SIZE**

  Configures the size of the buffer, in bytes, to read from the database. Default is 1 megabyte.

  **value**

  The **value** parameter sets the configuration value for the option identified by the **option** parameter. For those options which can only be turned on or off, this parameter should be set to 0 for off and 1 for on. Otherwise, set this parameter to an integer value that represents the number of units for which you are configuring the backup APIs.

**Class**

DbEnv,

**See Also**

[Database Environments and Related Methods](page 228), DbEnv::get_backup_config() (page 260), DbEnv::backup() (page 232), DbEnv::dbbackup() (page 239)
DbEnv::set_data_dir()

```cpp
#include <db_cxx.h>

int DbEnv::set_data_dir(const char *dir);
```

**Note**

This interface has been deprecated. You should use `DbEnv::add_data_dir()` (page 230) and `DbEnv::set_create_dir()` (page 305) 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 75) 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. Note that if you use this method for your application, and you also want to use the `db_recover` (page 748) or `db_archive` (page 722) utilities, then you should create a `DB_CONFIG` file and set the "set_data_dir" parameter in it.

The `DbEnv::set_data_dir()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_data_dir()` method may not be called after the `DbEnv::open()` (page 286) method is called. If the database environment already exists when `DbEnv::open()` (page 286) is called, the information specified to `DbEnv::set_data_dir()` must be consistent with the existing environment or corruption can occur.

The `DbEnv::set_data_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dir**

The `dir` parameter is a directory to be used as a location for database files. This directory must currently exist at environment open time.

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 DbEnv::set_data_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_create_dir()

```cpp
#include <db_cxx.h>

int DbEnv::set_create_dir(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 75) 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 DbEnv::set_create_dir() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_create_dir() method may be called at any time.

The DbEnv::set_create_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::add_data_dir() (page 230)

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 DbEnv::set_create_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv
See Also

Database Environments and Related Methods (page 228)
**DbEnv::set_encrypt()**

```cpp
#include <db_cxx.h>

int
DbEnv::set_encrypt(const char *passwd, u_int32_t flags);
```

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

The DbEnv::set_encrypt() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_encrypt() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_encrypt() must be consistent with the existing environment or an error will be returned.

The DbEnv::set_encrypt() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_encrypt() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**EOPNOTSUPP**

Cryptography is not available in this Berkeley DB release.

**Class**

DbEnv
See Also

Database Environments and Related Methods (page 228)
**DbEnv::set_event_notify()**

```cpp
#include <db_cxx.h>

int DbEnv::set_event_notify(
    void (*db_event_fcn)(DB_ENV *dbenv, u_int32_t event,
    void *event_info));
```

The DbEnv::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 DbEnv::set_event_notify() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_event_notify() method may be called at any time during the life of the application.

The DbEnv::set_event_notify() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_FAILCHK_PANIC`
    
The thread is about to return a `DB_RUNRECOVERY` error because a prior panic event has occurred and the thread has been marked by `DbEnv::failchk()` (page 248) as being held by a crashed process.

  The `event_info` parameter is a pointer to a `DB_FAILCHK_PANIC_INFO` structure, which contains these fields:

  ```c
  int error;
  ```
char symptom[DB_FAILURE_SYMPTOM_SIZE];

When this event is seen, the database environment has failed. All threads of control in the database environment should exit, and recovery should be run.

This event is generated only when failchk broadcasting is configured. You configured broadcasting by specifying `--enable-failchk_broadcast` when you compile your Berkeley DB library.

- **DB_EVENT_MUTEX_DIED**

  The thread is about to return a `DB_RUNRECOVERY` error because a mutex it requires has been marked by `DbEnv::failchk()` as being held by a crashed process.

  The `event_info` parameter is a pointer to a `DB_MUTEX_DIED_INFO` structure, which contains these fields:

  ```
  pid_t         mtxdied_pid;
  db_threadid_t mtxdied_tid;
  db_mutex_t    mtxdied_mtx;
  char          mtxdied_desc[DB_MUTEX_DESCRIBE_STRLEN];
  ```

  When this event is seen, the database environment has failed. All threads of control in the database environment should exit, and recovery should be run.

  This event is generated only when failchk broadcasting is configured. You configured broadcasting by specifying `--enable-failchk_broadcast` when you compile your Berkeley DB library.

- **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 `DbEnv::open()` 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 `DbEnv::open()` (page 286) 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_AUTOTAKEOVER_FAILED**

  The current subordinate process attempted to take over as the replication process, but the attempt failed.

  The replication process is the main Replication Manager process which is responsible for sending and processing most Replication Manager messages. Normally this is the first process started in a replication group, but when that process shuts down cleanly, a subordinate process will take over if one is available.

  This event means that this Replication Manager subordinate process attempted to take over as the replication process, but it failed. Replication Manager is not running locally but may be restarted by invoking `DbEnv::repmgr_start()` (page 629).

  The DB_EVENT_REP_AUTOTAKEOVER_FAILED event is provided only to applications configured for the Replication Manager.

- **DB_EVENT_REP_CLIENT**

  The local site is now a replication client.

  This event is generated when the replication role changes to client, either from master or from being unset. The role is unset when an environment is first created and after an environment is recovered. This event is not generated when restarting replication in an environment that was previously a client and was opened without recovery.

- **DB_EVENT_REP_CONNECT_BROKEN**

  A previously established Replication Manager message connection between the local site and a remote site has been broken. This event supplies the EID of the remote site, and an integer error code that identifies the reason the connection was broken.

  A non-zero error code indicates an unexpected condition such as a hardware failure or a protocol error. An application might respond by emitting an informational message or passing this information to other parts of the application using the app_private field. A zero error code indicates that the connection was cleanly closed by the other end. Replication Manager retries broken connections periodically until they are restored.

  The DB_EVENT_REP_CONNECT_BROKEN event is provided only to applications configured for the Replication Manager.
• **DB_EVENT_REP_CONNECT_ESTD**

A Replication Manager message connection has been established between the local site and a remote site. This event supplied the EID of the remote site.

The **DB_EVENT_REP_CONNECT_ESTD** event is provided only to applications configured for the Replication Manager.

• **DB_EVENT_REP_CONNECT_TRY_FAILED**

A Replication Manager attempt to establish a connection between the local site and a remote site has failed. This event supplies the EID of the remote site, and an integer error code that identifies the reason the connection attempt failed.

The **DB_EVENT_REP_CONNECT_TRY_FAILED** event is provided only to applications configured for the Replication Manager.

• **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 580) 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_ELECTED**

The local replication site has just won an election. A Base API application should call the **DbEnv::rep_start()** (page 599) 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 **DbEnv::rep_start()** (page 599) automatically on behalf of the application when appropriate (resulting in firing of the **DB_EVENT_REP_MASTER** event).

• **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_INIT_DONE**

The local client site has completed an internal initialization procedure.
• **DB_EVENT_REP_INQUEUE_FULL**

Incoming messages will be dropped because the Replication Manager incoming queue has reached its maximum threshold.

• **DB_EVENT_REP_JOIN_FAILURE**

The local client site is unable to synchronize with a new master, possibly because the client has turned off automatic internal initialization by setting the **DB_REP_CONF_AUTOINIT** flag to 0.

• **DB_EVENT_REP_LOCAL_SITE_REMOVED**

The local site has been removed from the replication group.

The **DB_EVENT_REP_LOCAL_SITE_REMOVED** event is provided only to applications configured for the Replication Manager.

• **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.

This event is generated when the replication role changes to master, either from client or from being unset. The role is unset when an environment is first created and after an environment is recovered. This event is not generated when restarting replication in an environment that was previously a master and was opened without recovery.

• **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 580) 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_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_PERM_FAILED**
The Replication Manager did not receive enough acknowledgements (based on the acknowledgement policy configured with `DbEnv::repmgr_set_ack_policy()` (page 618)) 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_SITE_ADDED**
  
  A new site has joined the replication group. The `event_info` parameter points to an integer containing the environment ID of the new site.

  The `DB_EVENT_REP_SITE_ADDED` event is provided only to applications configured for the Replication Manager.

- **DB_EVENT_REP_SITE_REMOVED**

  An existing remote site has been removed from the replication group. The `event_info` parameter points to an integer containing the environment ID of the site that was removed.

  The `DB_EVENT_REP_SITE_REMOVED` event is provided only to applications configured for the Replication Manager.

- **DB_EVENT_REP_STARTUPDONE**

  The replication 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**

`DbEnv`

**See Also**

Database Environments and Related Methods (page 228)
**DbEnv::set_errcall()**

```c
#include <db_cxx.h>

void DbEnv::set_errcall(void (*db_errcall_fcn)
    (const Dbenv *dbenv, const char *errpfx, const char *msg));
```

When an error occurs in the Berkeley DB library, an exception is thrown 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 `DbEnv::set_errcall()` and `DbEnv::set_errcall()` (page 315) 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 `DbEnv::set_error_stream()` (page 319) and `Db::set_error_stream()` (page 113) methods to display the additional information via an output stream, or the `Db::set_errfile()` (page 111) or `Db::set_errfile()` (page 317) methods to display the additional information via a C library `FILE *`. You should not mix these approaches.

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 `DbEnv::set_errcall()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::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 114) or `DbEnv::set_errpfx()` (page 320)).

• msg

   The msg parameter is the error message string.

Class

   DbEnv

See Also

   Database Environments and Related Methods (page 228)
DbEnv::set_errfile()

#include <db_cxx.h>

void
DbEnv::set_errfile(FILE *errfile);

When an error occurs in the Berkeley DB library, an exception is thrown 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 DbEnv::set_errfile() and Db::set_errfile() (page 111) 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 DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113) methods to display the additional messages via an output stream, or the DbEnv::set_errnocall() (page 315) or Db::set_errnocall() (page 109) methods to capture the additional error information in a way that does not use C library FILE *’s. You should not mix these approaches.

The error message will consist of the prefix string and a colon (":") (if a prefix string was previously specified using Db::set_errpfx() (page 114) or DbEnv::set_errpfx() (page 320) ), an error string, and a trailing <newline> character.

The default configuration when applications first create Db or DbEnv handles is as if the Db::set_errfile() (page 111) or DbEnv::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 111) or DbEnv::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:

- DbEnv::set_errfile()
- Db::set_errfile() (page 111)
- DbEnv::set_errnocall() (page 315)
- Db::set_errnocall() (page 109)
- DbEnv::set_error_stream() (page 319)
- Db::set_error_stream() (page 113)

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 DbEnv::set_errfile() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::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**

`DbEnv`

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set_error_stream()

```c
#include <db_cxx.h>

void DbEnv::set_error_stream(class ostream*);
```

When an error occurs in the Berkeley DB library, an exception is thrown or an `errno` 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 DbEnv::set_error_stream() and Db::set_error_stream() (page 113) methods are used to enhance the mechanism for reporting error messages to the application by setting the C++ ostream 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 stream.

The error message will consist of the prefix string and a colon (":") (if a prefix string was previously specified using Db::set_errpfx() (page 114), an error string, and a trailing <newline> character.

Setting stream to NULL unconfigures the interface.

Alternatively, you can use the DbEnv::set_errfile() (page 317) or Db::set_errfile() (page 111) methods to display the additional information via a C Library FILE *, or the DbEnv::set_errcall() (page 315) and Db::set_errcall() (page 109) methods to capture the additional error information in a way that does not use either output streams or C Library FILE *'s. You should not mix these approaches.

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 DbEnv::set_error_stream() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_error_stream() method may be called at any time during the life of the application.

**Parameters**

`stream`

The `stream` parameter is the application-specified output stream to be used for additional error information.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set_errpfx()

```cpp
#include <db_cxx.h>

void
DbEnv::set_errpfx(const char *errpfx);
```

Set the prefix string that appears before error messages issued by Berkeley DB. The length of this string can be no longer than 200.

The `Db::set_errpfx()` (page 114) and `DbEnv::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 `DbEnv::set_errpfx()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::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**

- `DbEnv`

**See Also**

- Database Environments and Related Methods (page 228)
DbEnv::set_feedback()  

```cpp
#include <db_cxx.h>

int
DbEnv::set_feedback(void (*db_feedback_fcn)(DbEnv *dbenv, int opcode,
    int percent));
```

Some operations performed by the Berkeley DB library can take non-trivial amounts of time. The DbEnv::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 DbEnv::set_feedback() method configures operations performed using the specified DbEnv handle.

The DbEnv::set_feedback() method may be called at any time during the life of the application.

The DbEnv::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**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set Flags()

```cpp
#include <db_cxx.h>

int
DbEnv::set_flags(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 DbEnv::set_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_flags() with this flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 DbEnv::set_flags() with the DB_CDB_ALLDB flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 DbEnv::open() (page 286) method is called.

• DB_DIRECT_DB

Turn off system buffering of Berkeley DB database files to avoid double caching.

Calling DbEnv::set_flags() with the DB_DIRECT_DB flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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_HOTBACKUP_IN_PROGRESS

Set this flag prior to creating a hot backup of a database environment. If a transaction with the bulk insert optimization enabled (with the DB_TXN_BULK (page 675) flag) is in progress, setting the DB_HOTBACKUP_IN_PROGRESS flag forces a checkpoint in the environment. After this flag is set in the environment, the bulk insert optimization is disabled, until the flag is reset. Using this protocol allows a hot backup procedure to make a consistent copy of the database even when bulk transactions are ongoing. For more information, see the section on Hot Backup in the Getting Started With Transaction Processing Guide and the description of the DB_TXN_BULK (page 675) flag in the DbEnv::txn_begin() (page 674) method.

The db_hotbackup (page 734) utility implements the protocol described above.

• 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 filesystems (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 DbEnv::set_flags() with the DB_DSYNC_DB flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 75) method. This flag will be ignored for queue databases for which DB_MULTIVERSION is not supported.

  Calling `DbEnv::set_flags()` with the DB_MULTIVERSION flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::set_flags()` with the DB_NOLOCKING flag only affects the specified `DbEnv` 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 `DbEnv::set_mp_mmapsize()` (page 487) method for further information).

  Calling `DbEnv::set_flags()` with the DB_NOMMAP flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::set_flags()` with the DB_NOPANIC flag only affects the specified `DbEnv` 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 `DbEnv::set_flags()` with the DB_OVERWRITE flag only affects the specified `DbEnv` 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 `DbEnv::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 `DbEnv::open()` (page 286) 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 `DbEnv::set_flags()` with the DB_REGION_INIT flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 DbEnv::set_flags() with the DB_TIME_NOTGRANTED flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 DbEnv::lock_get() (page 398) and DbEnv::lock_vec() (page 412) 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 DbEnv::set_flags() with the DB_TXN_NOSYNC flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 DbEnv::set_flags() with the DB_TXN_NOWAIT flag only affects the specified DbEnv handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all DbEnv 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 `DbEnv::txn_begin()` (page 674) method, and all non-transactional cursors will be opened as if DB_TXN_SNAPSHOT were passed to the `Db::cursor()` (page 179) method.

  Calling `DbEnv::set_flags()` with the DB_TXN_SNAPSHOT flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::set_flags()` with the DB_TXN_WRITE_NOSYNC flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::set_flags()` with the DB_YIELDCPU flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::set_flags()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

*Database Environments and Related Methods (page 228)*
DbEnv::set_intermediate_dir_mode()

```cpp
#include <db_cxx.h>

int DbEnv::set_intermediate_dir_mode(u_int32_t 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 `DbEnv::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 `DbEnv::set_intermediate_dir_mode()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_intermediate_dir_mode()` method may not be called after the `DbEnv::open()` (page 286) method is called.

The `DbEnv::set_intermediate_dir_mode()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `rwxrwxx---` 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 `DbEnv::set_intermediate_dir_mode()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

*Database Environments and Related Methods (page 228)*
DbEnv::set_isalive()

#include <db_cxx.h>

int DbEnv::set_isalive(int (*is_alive)(DbEnv *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 DbEnv::set_isalive() method supports the DbEnv::failchk() (page 248) 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 DbEnv::set_isalive() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_isalive() method may be called at any time during the life of the application.

The DbEnv::set_isalive() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_thread_id() (page 348) method.

• tid

  The tid parameter is a thread ID returned by the function specified to the DbEnv::set_thread_id() (page 348) 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 DbEnv::set_isalive() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_memory_init()

```c
#include <db_cxx.h>

int
DbEnv::set_memory_init(DB_MEM_CONFIG type, u_int32_t count);
```

This method sets the number of objects to allocate and initialize for a specified structure when an environment is created. Doing this helps avoid memory contention after startup. Using this method is optional; failure to use this method causes BDB to allocate a minimal number of structures that will grow dynamically. These structures are all allocated from the main environment region. The amount of memory in this region can be set via the DbEnv::set_memory_max() (page 336) method. If this method is not called then memory will be limited to the initial settings or by the (deprecated) set maximum interfaces.

The database environment's initialization 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_memory_init", one or more whitespace characters, followed by the struct specification, more white space and the count to be allocated. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DbEnv::set_memory_init() method must be called prior to opening the database environment. It may be called as often as needed to set the different configurations.

**Parameters**

**type**

The `type` parameter must be set to one of the following:

- **DB_MEM_LOCK**
  
  Initialize locks. A thread uses this structure to lock a page (or record for the QUEUE access method) and hold it to the end of a transactions.

- **DB_MEM_LOCKOBJECT**
  
  Initialize lock objects. For each page (or record) which is locked in the system, a lock object will be allocated.

- **DB_MEM_LOCKER**
  
  Initialize lockers. Each thread which is active in a transactional environment will use a locker structure either for each transaction which is active, or for each non-transactional cursor that is active.

- **DB_MEM_LOGID**
  
  Initialize the log fileid structures. For each database handle which is opened for writing in a transactional environment, a log fileid structure is used.

- **DB_MEM_TRANSACTION**
Initialize transaction structures. Each active transaction uses a transaction structure until it either commits or aborts.

**Note**

Currently transaction structures are not preallocated. This setting will be used to preallocate memory and objects related to transactions such as locker structures and mutexes.

- **DB_MEM_THREAD**

  Initialize thread identification structures. If thread tracking is enabled then each active thread will use a structure. Note that since a thread does not signal the BDB library that it will no longer be making calls, unused structures may accumulate until a cleanup is triggered either using a high water mark or by running `DbEnv::failchk()` (page 248).

**count**

The `count` parameter sets the number of specified objects to initialize.

The `count` specified for locks and lock objects should be at least 5 times the number of lock table partitions. You can examine the current number of lock table partitions configured for your environment using the `DbEnv::get_lk_partitions()` (page 378) method.

**Errors**

The `DbEnv::set_memory_init()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set_memory_max()

```cpp
#include <db_cxx.h>

int
DbEnv::set_memory_max(u_int32_t gbytes, u_int32_t bytes);
```

This method sets the maximum amount of memory to be used by shared structures in the main environment region. These are the structures used to coordinate access to the environment other than mutexes and those in the page cache (memory pool). If the region files are in memory mapped files, or if DB_PRIVATE is specified, the memory specified by this method is not allocated completely at startup. As memory is needed, the shared region will be extended or, in the case of DB_PRIVATE, more memory will be allocated using the system malloc call. For memory mapped files, a mapped region will be allocated to this size but the underlying file will only be allocated sufficient memory to hold the initial allocation of shared memory structures as set by DbEnv::set_memory_init() (page 334).

If no memory maximum is specified then it is calculated from defaults, initial settings or (deprecated) maximum settings of the various shared structures. In the case of environments created with DB_PRIVATE, no maximum need be set and the shared structure allocation will grow as needed until the process memory limit is exhausted.

The database environment’s maximum memory 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_memory_max", one or more whitespace characters, followed by the maximum to be allocated. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DbEnv::set_memory_max() method must be called prior to opening the database environment.

**Parameters**

- `gbytes`
  The maximum memory is set to `gbytes` gigabytes plus `bytes`.

- `bytes`
  The maximum memory is set to `gbytes` gigabytes plus `bytes`.

**Errors**

The DbEnv::set_memory_max() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.
Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_metadata_dir()

```cpp
#include <db_cxx.h>

int
DbEnv::set_metadata_dir(const char *dir);
```

The `DbEnv::set_metadata_dir()` method sets the directory where persistent metadata is stored. By default, persistent metadata is stored in the environment home directory.

When used in a replicated application, the metadata directory must be the same location for all sites within a replication group.

The `DbEnv::set_metadata_dir()` method may not be called after the `DbEnv::open()` (page 286) method is called. The directory identified by this method must already exist when the `DbEnv::open()` method is called. The directory identified by this method is added to the environment's list of data directories, if this directory is not already included on that list.

The database environment's metadata 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_metadata_dir", one or more whitespace characters, followed by the directory location. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The `DbEnv::set_metadata_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`dir`

The `dir` parameter identifies the directory used to store persistent metadata files.

**Class**

DbEnv

**See Also**

[Database Environments and Related Methods (page 228)](https://dblib.oracle.com), [DbEnv::get_metadata_dir() (page 267)](https://dblib.oracle.com)
DbEnv::set_message_stream()

```cpp
#include <db_cxx.h>

void DbEnv::set_message_stream(class ostream*);
```

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, the DbEnv::set_verbose() (page 357) and
DbEnv::stat_print() (page 360) methods.

The DbEnv::set_message_stream() and Db::set_message_stream() (page 134) methods are
used to display these messages for the application. In this case, the message will include a
trailing <newline> character.

Setting stream to NULL unconfigures the interface.

Alternatively, you can use the DbEnv::set_msgfile() (page 342) or Db::set_msgfile() (page
137) methods to display the additional information via a C Library FILE *, or the
DbEnv::set_msgcall() (page 340) and Db::set_msgcall() (page 135) methods to capture the
additional error information in a way that does not use either output streams or C Library
FILE *s. You should not mix these approaches.

The DbEnv::set_message_stream() method configures operations performed using
the specified DbEnv handle, not all operations performed on the underlying database
environment.

The DbEnv::set_message_stream() method may be called at any time during the life of the
application.

Parameters

**stream**

The `stream` parameter is the application-specified output stream to be used for additional
message information.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_msgcall()

```c
#include <db_cxx.h>

void DbEnv::set_msgcall(void (*db_msgcall_fcn)(const DbEnv *dbenv,
const char *msgpfx,
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, DbEnv::set_verbose() (page 357) and DbEnv::stat_print() (page 360).

The DbEnv::set_msgcall() and Db::set_msgcall() (page 135) 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 DbEnv::set_error_stream() (page 319) and Db::set_error_stream() (page 113) methods to display the messages via an output stream, or the Db::set_msgfile() (page 137) or Db::set_msgfile() (page 342) methods to display the messages via a C library FILE *. You should not mix these approaches.

The DbEnv::set_msgcall() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::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 three parameters:

- `dbenv`

  The `dbenv` parameter is the enclosing database environment.

- `msgpfx`

  The `msgpfx` parameter is the message prefix string that appears before informational messages. This value is the environmental default, and can be changed at any time using DbEnv::set_msgpfx() (page 343) or Db::set_msgpfx() (page 138).
• msg

The msg parameter is the message string.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_msgfile()

```cpp
#include <db_cxx.h>

void
DbEnv::set_msgfile(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, DbEnv::set_verbose() (page 357) and DbEnv::stat_print() (page 360).

The DbEnv::set_msgfile() and Db::set_msgfile() (page 137) 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 DbEnv::set_message_stream() (page 339) and Db::set_message_stream() (page 134) methods to display the messages via an output stream, or the DbEnv::set_msgcall() (page 340) or Db::set_msgcall() (page 135) methods to capture the additional error information in a way that does not use C library FILE *'s. You should not mix these approaches.

The DbEnv::set_msgfile() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::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**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set_msgpfx()

```c
#include <db_cxx.h>

void
DbEnv::set_msgpfx(const char *msgpfx);
```

Set the prefix string that appears before informational messages issued by Berkeley DB. The length of this string can be no longer than 200.

The `Db::set_msgpfx()` (page 138) and `DbEnv::set_msgpfx()` methods do not copy the memory to which the `msgpfx` parameter refers; rather, they maintain a reference to it. Although this allows applications to modify the informational message prefix at any time (without repeatedly calling the interfaces), it means the memory must be maintained until the handle is closed.

The `DbEnv::set_msgpfx()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_msgpfx()` method may be called at any time during the life of the application.

**Parameters**

`msgpfx`

The `msgpfx` parameter is the application-specified informational prefix for additional informational messages.

**Class**

`DbEnv`

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::set_shm_key()

```cpp
#include <db_cxx.h>

int DbEnv::set_shm_key(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 DbEnv::set_shm_key() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_shm_key() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_shm_key() must be consistent with the existing environment or corruption can occur.

The DbEnv::set_shm_key() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**shm_key**

The *shm_key* parameter is the base segment ID for the database environment.
Errors

The DbEnv::set_shm_key() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
**DbEnv::set_thread_count()**

```c
#include <db_cxx.h>

int DbEnv::set_thread_count(u_int32_t count);
```

Declare an approximate number of threads in the database environment. This method allocates resources in your environment for the threads your application will use. If you fail to properly estimate the number of threads your application will use, your application will run out of resources and errors will be returned when the application attempts to start one too many threads.

The `DbEnv::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.

The `DbEnv::set_thread_count()` method must be called prior to opening the database environment. In addition, this method must be used with the `DbEnv::failchk()` (page 248) method.

If a process invokes this method without the use of `DbEnv::failchk()` (page 248) the program may be unable to allocate a thread control block. This is true of the standalone Berkeley DB utility programs.

If a process has not configured an `is_alive` function from the `DbEnv::set_isalive()` (page 332) method, and then attempts to join a database environment configured for failure checking with the `DbEnv::failchk()` (page 248), `DbEnv::set_thread_id()` (page 348), `DbEnv::set_isalive()` (page 332) and `DbEnv::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 `DbEnv::failchk()` (page 248) 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 `DbEnv::set_thread_count()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_thread_count()` method may not be called after the `DbEnv::open()` (page 286) method is called.

The `DbEnv::set_thread_count()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.
Parameters

count

The count parameter is an approximate thread count for the database environment.

Errors

The DbEnv::set_thread_count() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_thread_id()

#include <db_cxx.h>

int DbEnv::set_thread_id(void (*thread_id)(DbEnv *dbenv,
       pid_t *pid, db_threadid_t *tid));

Declare a function that returns a unique identifier pair for the current thread of control. The DbEnv::set_thread_id() method supports the DbEnv::failchk() (page 248) 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 DbEnv::set_thread_id() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_thread_id() method may be called at any time during the life of the application.

The DbEnv::set_thread_id() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_thread_id() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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 DbEnv::failchk() (page 248) method is run, it may be possible for DbEnv::failchk() (page 248) 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 DbEnv::failchk() (page 248) 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 DbEnv::set_isalive() (page 332) method must be compatible with any thread_id function specified to DbEnv::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_threadid_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

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_thread_id_string()

```c
#include <db_cxx>

int DbEnv::set_thread_id_string(void (*thread_id)(DbEnv *dbenv, pid_t *pid, db_threadid_t *tid));
```

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 DbEnv::set_thread_id_string() method supports the DbEnv::set_thread_id() method.

The DbEnv::set_thread_id_string() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_thread_id_string() method may be called at any time during the life of the application.

The DbEnv::set_thread_id_string() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_thread_id_string() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_timeout()

```cpp
#include <db_cxx.h>

int
DbEnv::set_timeout(db_timeout_t timeout, u_int32_t flags);
```

The `DbEnv::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 `DbEnv::lock_vec()` (page 412) and `DbTxn::set_timeout()` (page 698) for more information.

The `DbEnv::set_timeout()` method may not be used in a database environment without a locking subsystem.

The `DbEnv::set_timeout()` method may be called at any time during the life of the application.

The `DbEnv::set_timeout()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv handle.

• **DB_SET_MUTEX_FAILCHK_TIMEOUT**

If failchk broadcasting has been configured, then set the timeout value on how long a thread will wait for a mutex lock before checking whether DbEnv::failchk() (page 248) has marked the mutex as failed. The default is to check once every second.

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_mutex_failchk_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 DbEnv 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 DbEnv::open() (page 286) 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 DbEnv 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 DbEnv handle.

**Errors**

The DbEnv::set_timeout() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

DB_SET_MUTEX_FAILCHK_TIMEOUT was specified even though Berkeley DB was not configured with --enable-failchk_broadcast; an invalid flag value or parameter was specified;

Class

DbEnv

See Also

Database Environments and Related Methods (page 228)
DbEnv::set_tmp_dir()

```
#include <db_cxx.h>

int
DbEnv::set_tmp_dir(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 DbEnv::set_tmp_dir() method configures operations performed using the specified
DbEnv handle, not all operations performed on the underlying database environment.
The `DbEnv::set_tmp_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dir**

The `dir` parameter is the directory to be used to store temporary files. This directory must currently exist at environment open time.

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 `DbEnv::set_tmp_dir()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

`Database Environments and Related Methods` (page 228)
DbEnv::set_verbose()

```c++
#include <db_cxx.h>

int DbEnv::set_verbose(u_int32_t which, int onoff);
```

The `DbEnv::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 `DbEnv::set_msgfile()` (page 342) 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 `DbEnv::set_verbose()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_verbose()` method may be called at any time during the life of the application.

The `DbEnv::set_verbose()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::open() (page 286) 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_REPLEASE
  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_SLICE
  Display detailed information about sliced databases.

• 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 `DbEnv::set_verbose()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

Database Environments and Related Methods (page 228)
DbEnv::stat_print()

```cpp
#include <db_cxx.h>

int
DbEnv::stat_print(u_int32_t flags);
```

The DbEnv::stat_print() method displays the default statistical information. The information is printed to a specified output channel (see the DbEnv::set_msgfile() (page 342) method for more information), or passed to an application callback function (see the DbEnv::set_msgcall() (page 340) method for more information).

The DbEnv::stat_print() method may not be called before the DbEnv::open() (page 286) method is called.

The DbEnv::stat_print() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

For Berkeley DB SQL environment statistics, see Command Line Features Unique to dbsql (page 760).

**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_ALLOC**
  Display allocation information. To display allocation information, both DB_STAT_ALLOC and DB_STAT_ALL need to be set.

- **DB_STAT_CLEAR**
  Reset statistics after displaying their values.

- **DB_STAT_SUBSYSTEM**
  Display information for all configured subsystems.

**Class**

DbEnv

**See Also**

Database Environments and Related Methods (page 228)
### DbEnv::strerror()

```cpp
#include <db_cxx.h>

static char * DbEnv::strerror(int error);
```

The DbEnv::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**

DbEnv

**See Also**

[Database Environments and Related Methods (page 228)]
DbEnv::version()

```cpp
#include <db_cxx.h>

static char *
DbEnv::version(int *major, int *minor, int *patch);
```

The DbEnv::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 DbEnv::full_version() (page 252).

**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**

DbEnv

**See Also**

- Database Environments and Related Methods (page 228)
# Chapter 6. The DbException Class

```cpp
#include <db_cxx.h>
class DbException {
public:
    int get_errno() const;
    virtual const char *what() const;
    DbEnv *get_env() const;
};
```

This information describes the DbException class and how it is used by the various Berkeley DB classes.

Most methods in the Berkeley DB classes return an `int`, but also throw an exception. This allows for two different error behaviors. By default, the Berkeley DB C++ API is configured to throw an exception whenever a serious error occurs. This generally allows for cleaner logic for transaction processing because a try block can surround a single transaction. Alternatively, Berkeley DB can be configured to not throw exceptions, and instead have the individual function return an error code, by setting the `DB_CXX_NO_EXCEPTIONS` for the `Db` (page 21) and `DbEnv` (page 237) constructors.

A DbException object contains an informational string, an `errno`, and a reference to the environment from which the exception was thrown. The `errno` can be obtained by using the `DbException::get_errno()` method, and can be used, in standard cases, to determine the type of the exception. The informational string can be obtained by using the `DbException::what()` method. And, the environment can be obtained using the `DbException::get_env()` method.

Where available, this class inherits from the standard class exception.

Some methods may return non-zero values without issuing an exception. This occurs in situations that are not normally considered an error, but when some informational status is returned. For example, the `Db::get()` (page 31) method returns `DB_NOTFOUND` when a requested key does not appear in the database.
## DB C++ Exceptions

<table>
<thead>
<tr>
<th>DB C++ Exceptions</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbDeadlockException</td>
<td>Exception class for deadlocks</td>
</tr>
<tr>
<td>DbLockNotGrantedException</td>
<td>Exception class for lock request failures</td>
</tr>
<tr>
<td>DbMemoryException</td>
<td>Exception class for insufficient memory</td>
</tr>
<tr>
<td>DbRepHandleDeadException</td>
<td>Exception class for database and cursor handles that are invalidated in a replicated application.</td>
</tr>
<tr>
<td>DbRunRecoveryException</td>
<td>Exception class for failures requiring recovery</td>
</tr>
</tbody>
</table>
DbDeadlockException

```cpp
#include <db_cxx.h>

class DbDeadlockException : public DbException { ... };
```

This information describes the DbDeadlockException class and how it is used by the various Berkeley DB classes.

A DbDeadlockException is thrown when multiple threads competing for a lock are deadlocked, when a lock request has timed out (and `DB_TIME_NOTGRANTED` has not been set in the environment), or when a lock request would need to block and the transaction has been configured to not wait for locks. One of the threads' transactions is selected for termination, and a DbDeadlockException is thrown to that thread.

The `DbException` errno value is set to `DB_LOCK_DEADLOCK`.
DbLockNotGrantedException

```cpp
#include <db_cxx.h>

class DbLockNotGrantedException : public DbException {
  public:
    db_lockop_t get_op() const;
    db_lockmode_t get_mode() const;
    const Dbt* get_obj() const;
    DbLock* get_lock() const;
    int get_index() const;
};
```

This information describes the DbLockNotGrantedException class and how it is used by the various Berkeley DB classes.

A DbLockNotGrantedException is thrown when lock or transaction timeouts have been configured, a database operation has timed out, and the DB_TIME_NOTGRANTED configuration flag has been specified.

Additionally DbLockNotGrantedException is thrown when a Berkeley DB Concurrent Data Store database environment configured for lock timeouts was unable to grant a lock in the allowed time.

Finally, DbLockNotGrantedException is thrown when a lock requested using the DbEnv::lock_get() (page 398) or DbEnv::lock_vec() (page 412) methods, where the DB_LOCK_NOWAIT flag or lock timers were configured, could not be granted before the wait-time expired.

The DbException errno value is set to DB_LOCK_NOTGRANTED.

The following getter methods are available on this class:

- **get_op()**
  Returns DB_LOCK_GET when DbEnv::lock_get() (page 398) was called, and returns the op for the failed DB_LOCKREQ when DbEnv::lock_vec() (page 412) was called. If this exception is raised due to a database operation, DB_LOCK_GET is returned.

- **get_mode()**
  Returns the mode parameter when DbEnv::lock_get() (page 398) was called, and returns the mode for the failed DB_LOCKREQ when DbEnv::lock_vec() (page 412) was called. If this exception is raised due to a database operation, DB_LOCK_NG is returned.

- **get_obj()**
  Returns the object parameter when DbEnv::lock_get() (page 398) was called, and returns the object for the failed DB_LOCKREQ when DbEnv::lock_vec() (page 412) was called. The Dbt pointer may or may not refer valid memory, depending on whether the Dbt used in the
call to the failed `DbEnv::lock_get()` (page 398) or `DbEnv::lock_vec()` (page 412) method is still in scope and has not been deleted.

- **get_lock()**
  
  Returns NULL when `DbEnv::lock_get()` (page 398) was called, and returns the lock in the failed `DB_LOCKREQ` when `DbEnv::lock_vec()` (page 412) was called. If this exception is raised due to a database operation, NULL is returned.

- **get_index()**
  
  Returns -1 when `DbEnv::lock_get()` (page 398) was called, and returns the index of the failed `DB_LOCKREQ` when `DbEnv::lock_vec()` (page 412) was called. If this exception is raised due to a database operation, 0 is returned.
DbMemoryException

```cpp
#include <db_cxx.h>

class DbMemoryException : public DbException {
public:
    Dbt *get_dbt() const;
};
```

This information describes the DbMemoryException class and how it is used by the various Berkeley DB classes.

A DbMemoryException is thrown when there is insufficient memory to complete an operation, and there is the possibility of recovering. An example is during a `Db::get()` (page 31) or `Dbc::get()` (page 193) operation with the `Dbt` flags set to `DB_DBT_USERMEM`.

The `DbException` errno value is set to `DB_BUFFER_SMALL` or `ENOMEM`.

The `get_dbt()` method returns the `Dbt` with insufficient memory to complete the operation, causing the DbMemoryException to be thrown. The `Dbt` pointer may or may not refer to valid memory, depending on whether the `Dbt` used in the call to the failed Berkeley DB method is still in scope and has not been deleted.
DbRepHandleDeadException

```cpp
#include <db_cxx.h>

class DbRepHandleDeadException : public DbException {
};
```

This information describes the DbRepHandleDead class and how it is used by the various Berkeley DB classes.

A DbRepHandleDeadException is seen only for replicated applications. 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.

This exception is therefore thrown when the application attempts to access a database or cursor handle that has been invalidated due to a transaction roll back.

When this exception is seen, the application must abandon the attempted operation, discard the handle, and then open a new one before proceeding with the abandoned operation.
DbRunRecoveryException

```cpp
#include <db_cxx.h>

class DbRunRecoveryException : public DbException { ... };
```

This information describes the DbRunRecoveryException class and how it is used by the various Berkeley DB classes.

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). When a fatal error occurs in Berkeley DB, methods will throw a DbRunRecoveryException, at which point all subsequent Berkeley DB calls will also fail in the same way. When this occurs, recovery should be performed.

The `DbException` errno value is set to `DB_RUNRECOVERY`. 
Chapter 7. The DbLock Handle

```cpp
#include <db_cxx.h>

class DbLock {
public:
    DbLock();
    DbLock(const DbLock &);
    DbLock &operator = (const DbLock &);
    ~DbLock();
};
```

The locking interfaces for the Berkeley DB database environment are methods of the `DbEnv` handle. The `DbLock` object is the handle for a single lock, and has no methods of its own.
## Locking Subsystem and Related Methods

<table>
<thead>
<tr>
<th>Locking Subsystem and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbDeadlockException</td>
<td>Exception class for deadlocks</td>
</tr>
<tr>
<td>DbLockNotGrantedException</td>
<td>Exception class for lock request failures</td>
</tr>
<tr>
<td>DbEnv::lock_detect()</td>
<td>Perform deadlock detection</td>
</tr>
<tr>
<td>DbEnv::lock_get()</td>
<td>Acquire a lock</td>
</tr>
<tr>
<td>DbEnv::lock_id()</td>
<td>Acquire a locker ID</td>
</tr>
<tr>
<td>DbEnv::lock_id_free()</td>
<td>Release a locker ID</td>
</tr>
<tr>
<td>DbEnv::lock_put()</td>
<td>Release a lock</td>
</tr>
<tr>
<td>DbEnv::lock_stat()</td>
<td>Return lock subsystem statistics</td>
</tr>
<tr>
<td>DbEnv::lock_stat_print()</td>
<td>Print lock subsystem statistics</td>
</tr>
<tr>
<td>DbEnv::lock_vec()</td>
<td>Acquire/release locks</td>
</tr>
<tr>
<td>DbEnv::cdsgroup_begin()</td>
<td>Get a locker ID in Berkeley DB Concurrent Data Store</td>
</tr>
</tbody>
</table>

## Locking Subsystem Configuration

<table>
<thead>
<tr>
<th>Configuration Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbEnv::set_timeout(), DbEnv::get_timeout()</td>
<td>Set/get lock and transaction timeout</td>
</tr>
<tr>
<td>DbEnv::set_lk_conflicts(), DbEnv::get_lk_conflicts()</td>
<td>Set/get lock conflicts matrix</td>
</tr>
<tr>
<td>DbEnv::set_lk_detect(), DbEnv::get_lk_detect()</td>
<td>Set/get automatic deadlock detection</td>
</tr>
<tr>
<td>DbEnv::set_lk_max_lockers(), DbEnv::get_lk_max_lockers()</td>
<td>Set/get maximum number of lockers</td>
</tr>
<tr>
<td>DbEnv::set_lk_max_locks(), DbEnv::get_lk_max_locks()</td>
<td>Set/get maximum number of locks</td>
</tr>
<tr>
<td>DbEnv::set_lk_max_objects(), DbEnv::get_lk_max_objects()</td>
<td>Set/get maximum number of lock objects</td>
</tr>
<tr>
<td>DbEnv::set_lk_partitions(), DbEnv::get_lk_partitions()</td>
<td>Set/get number of lock partitions</td>
</tr>
<tr>
<td>DbEnv::set_lk_priority(), DbEnv::get_lk_priority()</td>
<td>Set/get a locker's deadlock priority</td>
</tr>
<tr>
<td>DbEnv::set_lk_tablesize(), DbEnv::get_lk_tablesize()</td>
<td>Set/get size of the lock object hash table</td>
</tr>
</tbody>
</table>
DbEnv::get_lk_conflicts()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_conflicts(const u_int8_t **lk_conflictsp, int *lk_modesp);
```

The `DbEnv::get_lk_conflicts()` method returns the current conflicts array. You can specify a conflicts array using `DbEnv::set_lk_conflicts()` (page 381).

The `DbEnv::get_lk_conflicts()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_conflicts()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::get_lk_conflicts()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLock`

**See Also**

- Locking Subsystem and Related Methods (page 372), `DbEnv::set_lk_conflicts()` (page 381)
DbEnv::get_lk_detect()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_detect(u_int32_t *lk_detectp);
```

The `DbEnv::get_lk_detect()` method returns the deadlock detector configuration. You can manage this using the `DbEnv::set_lk_detect()` (page 383) method.

The `DbEnv::get_lk_detect()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_detect()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lk_detectp**

The `DbEnv::get_lk_detect()` method returns the deadlock detector configuration in `lk_detectp`.

**Errors**

The `DbEnv::get_lk_detect()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLock`

**See Also**

- Locking Subsystem and Related Methods (page 372), `DbEnv::set_lk_detect()` (page 383)
DbEnv::get_lk_max_lockers()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_max_lockers(u_int32_t *, lk_maxp);
```

The `DbEnv::get_lk_max_lockers()` method returns the maximum number of potential lockers. You can configure this using the `DbEnv::set_lk_max_lockers()` (page 385) method.

The `DbEnv::get_lk_max_lockers()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_max_lockers()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

lk_maxp

The `DbEnv::get_lk_max_lockers()` method returns the maximum number of lockers in `lk_maxp`.

**Errors**

The `DbEnv::get_lk_max_lockers()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372), DbEnv::set_lk_max_lockers() (page 385)
DbEnv::get_lk_max_locks()

```c
#include <db_cxx.h>

int DbEnv::get_lk_max_locks(u_int32_t *lk_maxp);
```

The `DbEnv::get_lk_max_locks()` method returns the maximum number of potential locks. You can configure this using the `DbEnv::set_lk_max_locks()` (page 387) method.

The `DbEnv::get_lk_max_locks()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_max_locks()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lk_maxp**

The `DbEnv::get_lk_max_locks()` method returns the maximum number of locks in `lk_maxp`.

**Errors**

The `DbEnv::get_lk_max_locks()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLock`

**See Also**

`Locking Subsystem and Related Methods (page 372), DbEnv::set_lk_max_locks()` (page 387)
DbEnv::get_lk_max_objects()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_max_objects(u_int32_t *lk_maxp);
```

The `DbEnv::get_lk_max_objects()` method returns the maximum number of locked objects. You can configure this using the `DbEnv::set_lk_max_objects()` method.

The `DbEnv::get_lk_max_objects()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_max_objects()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lk_maxp**

The `DbEnv::get_lk_max_objects()` method returns the maximum number of potentially locked objects in `lk_maxp`.

**Errors**

The `DbEnv::get_lk_max_objects()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLock`

**See Also**

`Locking Subsystem and Related Methods (page 372)`, `DbEnv::set_lk_max_objects()` (page 389)
DbEnv::get_lk_partitions()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_partitions(u_int32_t *lk_partitions);
```

The `DbEnv::get_lk_partitions()` method returns the number of lock table partitions used in the Berkeley DB environment. You can configure this using the `DbEnv::set_lk_partitions()` (page 391) method.

The `DbEnv::get_lk_partitions()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_partitions()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lk_partitions**

The `DbEnv::get_lk_partitions()` method returns the number of partitions in `lk_partitions`.

**Errors**

The `DbEnv::get_lk_partitions()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLock`

**See Also**

- Locking Subsystem and Related Methods (page 372), `DbEnv::set_lk_partitions()` (page 391)
DbEnv::get lk_priority()

```c++
#include <db_cxx.h>

int
DbEnv::get lk_priority(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 DbEnv::get lk_priority() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372), DbEnv::set lk_priority() (page 393)
DbEnv::get_lk_tablesize()

```c
#include <db_cxx.h>

int
DbEnv::get_lk_tablesize(u_int32_t *tablesizep);
```

The `DbEnv::get_lk_tablesize()` method returns the size of the lock object hash table in
the Berkeley DB environment. This value is set using the `DbEnv::set_lk_tablesize()` method.

The `DbEnv::get_lk_tablesize()` method may be called at any time during the life of the application.

The `DbEnv::get_lk_tablesize()` method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`tablesizep`

The `tablesizep` parameter references memory into which is copied the size of the lock object
hash table configured for the Berkeley DB environment.

**Class**

`DbEnv`, `DbLock`

**See Also**

`Locking Subsystem and Related Methods (page 372)`
DbEnv::set_lk_conflicts()

#include <db_cxx.h>

int
DbEnv::set_lk_conflicts(u_int8_t *conflicts, int nmodes);

Set the locking conflicts matrix.

If DbEnv::set_lk_conflicts() is never called, a standard conflicts array is used; see Standard Lock Modes for more information.

The DbEnv::set_lk_conflicts() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lk_conflicts() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_lk_conflicts() will be ignored.

The DbEnv::set_lk_conflicts() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_lk_conflicts() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

ENOMEM

The conflicts array could not be copied.
Class

DbEnv, DbLock

See Also

Locking Subsystem and Related Methods (page 372)
### DbEnv::set_lk_detect()

```cpp
#include <db_cxx.h>

int DbEnv::set_lk_detect(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 `DbEnv::set_lk_detect()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::set_lk_detect()` method may be called either before or after environment open, but once it is set it may not be changed again during the environment's lifetime.

The `DbEnv::set_lk_detect()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_lk_detect()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbLock`

**See Also**

[Locking Subsystem and Related Methods (page 372)]
DbEnv::set_lk_max_lockers()

```c
#include <db_cxx.h>

int
DbEnv::set_lk_max_lockers(u_int32_t max);
```

This method is deprecated. Instead, use `DbEnv::set_memory_init()` (page 334), `DbEnv::set_memory_max()` (page 336), and `DbEnv::set_lk_tablesize()` (page 394).

Sets the maximum number of locking entities supported by the Berkeley DB environment. This value is used by `DbEnv::open()` (page 286) 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 `DbEnv::set_lk_max_lockers()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::set_lk_max_lockers()` method may not be called after the `DbEnv::open()` (page 286) method is called. If the database environment already exists when `DbEnv::open()` (page 286) is called, the information specified to `DbEnv::set_lk_max_lockers()` will be ignored.

The `DbEnv::set_lk_max_lockers()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **max**

  The `max` parameter is the maximum number simultaneous locking entities supported by the Berkeley DB environment.

**Errors**

The `DbEnv::set_lk_max_lockers()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbLock`
See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::set_lk_max_locks()

#include <db_cxx.h>

int
DbEnv::set_lk_max_locks(u_int32_t max);

This method is deprecated. Instead, use DbEnv::set_memory_init() (page 334),
DbEnv::set_memory_max() (page 336), and DbEnv::set_lk_tablesize() (page 394).

Set the maximum number of locks supported by the Berkeley DB environment. This value is
used by DbEnv::open() (page 286) to estimate how much space to allocate for various lock-
table data structures. The default value is 1000 locks. The final value specified for the locks
should be more than or equal to the number of lock table partitions. 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 DbEnv::set_lk_max_locks() method configures a database environment, not only
operations performed using the specified DbEnv handle.

The DbEnv::set_lk_max_locks() method may not be called after the DbEnv::open() (page
286) method is called. If the database environment already exists when DbEnv::open() (page
286) is called, the information specified to DbEnv::set_lk_max_locks() will be ignored.

The DbEnv::set_lk_max_locks() method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

max

The max parameter is the maximum number of locks supported by the Berkeley DB
environment.

Errors

The DbEnv::set_lk_max_locks() method may fail and throw a DbException exception,
encapsulating one of the following non-zero errors, or return one of the following non-zero
errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value
or parameter was specified.

Class

DbEnv, DbLock
See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::set_lk_max_objects()

```c
#include <db_cxx.h>

int DbEnv::set_lk_max_objects(u_int32_t max);
```

This method is deprecated. Instead, use `DbEnv::set_memory_init()` (page 334), `DbEnv::set_memory_max()` (page 336), and `DbEnv::set_lk_tablesize()` (page 394).

Set the maximum number of locked objects supported by the Berkeley DB environment. This value is used by `DbEnv::open()` (page 286) to estimate how much space to allocate for various lock-table data structures. The default value is 1000 objects. The final value specified for the lock objects should be more than or equal to the number of lock table partitions. 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 `DbEnv::set_lk_max_objects()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::set_lk_max_objects()` method may not be called after the `DbEnv::open()` (page 286) method is called. If the database environment already exists when `DbEnv::open()` (page 286) is called, the information specified to `DbEnv::set_lk_max_objects()` will be ignored.

The `DbEnv::set_lk_max_objects()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `max`

  The `max` parameter is the maximum number of locked objects supported by the Berkeley DB environment.

**Errors**

The `DbEnv::set_lk_max_objects()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.
Class

DbEnv, DbLock

See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::set_lk_partitions()

```c
#include <db_cxx.h>

int DbEnv::set_lk_partitions(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 DbEnv::set_lk_partitions() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lk_partitions() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_lk_partitions() will be ignored.

The DbEnv::set_lk_partitions() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **partitions**
  
  The `partitions` parameter is the number of partitions to be configured in the Berkeley DB environment.

**Errors**

The DbEnv::set_lk_partitions() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv, DbLock
See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::set_lk_priority()

```c
#include <db_cxx.h>

int
DbEnv::set_lk_priority(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 DbEnv::set_lk_priority() method may be called at any time during the life of the application.

The DbEnv::set_lk_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **lockerid**

The lockerid parameter represents a locker returned by DbEnv::lock_id().

- **priority**

The priority parameter must be a value between 0 and 2^32-1.

**Errors**

The DbEnv::set_lk_priority() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372)
DbEnv::set_lk_tablesize()

#include <db_cxx.h>

int DbEnv::set_lk_tablesize(u_int32_t tablesize);

Sets the number of buckets in the lock object hash table in the Berkeley DB environment. The default value is estimated based on defaults, initial and (deprecated) maximum settings of the number of lock objects allocated. The maximum memory allocation is also considered. The table is generally set to be close to the number of lock objects in the system to avoid collisions and delay in processing lock operations.

The database environment’s tablesize 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_tablesize”, one or more whitespace characters, and the size of the table. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DbEnv::set_lk_tablesize() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lk_tablesize() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_lk_tablesize() will be ignored.

The DbEnv::set_lk_tablesize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

tablesize

The tablesize parameter provides the size of the lock object hash table to be configured in the Berkeley DB environment.

Errors

The DbEnv::set_lk_tablesize() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv, DbLock
See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::lock_detect()

```c++
#include <db_cxx.h>

int
DbEnv::lock_detect(u_int32_t flags, u_int32_t atype, int *rejected);
```

The `DbEnv::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 `DbEnv::lock_detect()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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.
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 `DbEnv::lock_detect()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372)
DbEnv::lock_get()

```c
#include <db_cxx.h>

int DbEnv::lock_get(u_int32_t locker, u_int32_t flags,
                    const Dbt *object, const db_lockmode_t lock_mode, DbLock *lock);
```

The `DbEnv::lock_get()` method acquires a lock from the lock table, returning information about it in the `lock` parameter.

The `DbEnv::lock_get()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_lk_conflicts() (page 381) and Standard Lock Modes for more information on the lock conflict matrix.

**lock**

The DbEnv::lock_get() method returns the lock information in `lock`.

**Errors**

The DbEnv::lock_get() method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

`DbLockNotGrantedException` (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**DbLockNotGrantedException or 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.

`DbLockNotGrantedException` (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**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
DbEnv, DbLock

See Also
Locking Subsystem and Related Methods (page 372)
**DbEnv::lock_id()**

```c
#include <db_cxx.h>

int
DbEnv::lock_id(u_int32_t *idp);
```

The `DbEnv::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.

Note that lockers are not free-threaded; lockers can not be used by more than one thread at the same time.

The `DbEnv::lock_id_free()` method should be called to return the locker ID to the Berkeley DB library when it is no longer needed.

The `DbEnv::lock_id()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*idp*

The `idp` parameter references memory into which the allocated locker ID is copied.

**Class**

`DbEnv, DbLock`

**See Also**

`Locking Subsystem and Related Methods (page 372)`
DbEnv::lock_id_free()

```cpp
#include <db_cxx.h>

int
DbEnv::lock_id_free(u_int32_t id);
```

The DbEnv::lock_id_free() method frees a locker ID allocated by the DbEnv::lock_id() (page 401) method.

The DbEnv::lock_id_free() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

id

The id parameter is the locker id to be freed.

**Errors**

The DbEnv::lock_id_free() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372)
The `DbEnv::lock_put()` method releases the `lock`.

The `DbEnv::lock_put()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **lock**

  The `lock` parameter is the lock to be released.

**Errors**

The `DbEnv::lock_put()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  An invalid flag value or parameter was specified.

**Class**

- `DbEnv`, `DbLock`

**See Also**

- "Locking Subsystem and Related Methods (page 372)"
**DbEnv::lock_stat()**

```c
#include <db_cxx.h>

int
DbEnv::lock_stat(DB_LOCK_STAT **statp, u_int32_t flags);
```

The `DbEnv::lock_stat()` method returns the locking subsystem statistics.

The `DbEnv::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 `DbEnv::set_alloc()` (page 294) for more information), they are used to allocate the memory; otherwise, the standard C library `malloc` 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_cur_maxid;
  The current maximum unused locker ID.
- `u_int32_t st_hash_len;
  Maximum length of a lock hash bucket.
- `u_int32_t st_id;
  The last allocated locker ID.
- `u_int32_t st_initlocks;
  The initial number of locks allocated in the lock table.
- `u_int32_t st_initlockers;
  The initial number of lockers allocated in the lock table.
- `u_int32_t st_initobjects;
  The initial number of lock objects allocated in the lock table.
- `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_lock_wait;
  The number of lock requests not immediately available due to conflicts, 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.

• uintmax_t st_lockers_wait;
The number of requests to allocate or deallocate a locker for which the thread of control
waited.

• u_int32_t st_lockers;
The current number of lockers allocated in the lock table.

• u_int32_t st_locks;
The current number of locks allocated in the lock table.

• uintmax_t st_locksteals;
The maximum number of locks stolen by an empty partition.

• db_timeout_t st_locktimeout;
Lock timeout value.

• u_int32_t st_maxhlocks;
The maximum number of locks in any hash bucket at any one time.

• u_int32_t st_maxhobjects;
The maximum number of objects in any hash bucket at any one time.

• u_int32_t st_maxlockers;
The maximum number of lockers possible.

• u_int32_t st_maxlocks;
The maximum number of locks possible.

• uintmax_t st_maxlsteals;
The maximum number of lock steals for any one partition.

• u_int32_t st_maxnlockers;
The maximum number of lockers at any one time.

• 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_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_maxobjects;`
  The maximum number of lock objects possible.

- `uintmax_t st_maxsteals;`
  The maximum number of object steals for any one partition.

- `uintmax_t st_ndeadlocks;`
  The number of deadlocks.

- `uintmax_t st_ndowngrade;`
  The total number of locks downgraded.

- `u_int32_t st_nlockers;`
  The number of current lockers.

- `uintmax_t st_nlockers_hit;`
  The number of hits in the thread locker cache.

- `uintmax_t st_nlockers_reused;`
  Total number of lockers reused.

- `u_int32_t st_nlocks;`
  The number of current locks.

- `uintmax_t st_nlocktimeouts;`
  The number of lock requests that have timed out.

- `int st_nmodes;`
  The number of lock modes.

- `u_int32_t st_objects;`
  The number of current lock objects.

- `uintmax_t st_nreleases;`
  The total number of locks released.

- `uintmax_t st_nrequests;`
The total number of locks requested.

- `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_nupgrade;`
  The total number of locks upgraded.

- `u_int32_t st_objects;`
  The current number of lock objects allocated in the lock table.

- `uintmax_t st_objectsteals;`
  The maximum number of objects stolen by an empty partition.

- `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_objs_wait;`
  The number of requests to allocate or deallocate an object for which the thread of control waited.

- `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_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_nowait;`
  The number of times that a thread of control was able to obtain the lock partition mutex without waiting.

- `uintmax_t st_part_wait;`
  The number of times that a thread of control was forced to wait before obtaining the lock partition mutex.

- `u_int32_t st_partitions;`
  The number of lock table partitions.
• uintmax_t st_region_nowait;
  The number of times that a thread of control was able to obtain the lock region 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.

• roff_t st_regsize;
  The region size, in bytes.

• u_int32_t st_tablesize;
  The size of the object hash table.

• db_timeout_t st_txntimeout;
  Transaction timeout value.

The `DbEnv::lock_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::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 `DbEnv::lock_stat()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLock
See Also

Locking Subsystem and Related Methods (page 372)
DbEnv::lock_stat_print()

```cpp
#include <db_cxx.h>

int DbEnv::lock_stat_print(u_int32_t flags);
```

The `DbEnv::lock_stat_print()` method displays the locking subsystem statistical information, as described for the `DbEnv::lock_stat()` method. The information is printed to a specified output channel (see the `DbEnv::set_msgfile()` method for more information), or passed to an application callback function (see the `DbEnv::set_msgcall()` method for more information).

The `DbEnv::lock_stat_print()` method may not be called before the `DbEnv::open()` method is called.

The `DbEnv::lock_stat_print()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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. For each object, the amount of data displayed is limited to 100 bytes, unless some other limit is set using the `DB_CONFIG “set_data_len”` parameter.

- **DB_STAT_ALLOC**
  
  Display allocation information. To display allocation information, both `DB_STAT_ALLOC` and `DB_STAT_ALL` need to be set.

- **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. For each object, the amount of data displayed is limited to 100 bytes, unless some other limit is set using the `DB_CONFIG “set_data_len”` parameter.
• **DB_STAT_LOCK_PARAMS**
  Display the locking subsystem parameters.

**Class**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372)
The `DbEnv::lock_vec()` method atomically obtains and releases one or more locks from the lock table. The `DbEnv::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 `DbEnv::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 `DbEnv::lock_vec()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::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 DbEnv::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 DbEnv::lock_vec() or DbEnv::lock_put() (page 403) 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 DbEnv::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 DbEnv::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 [DbEnv::set_lk_conflicts()](page 381) 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 `DbEnv::lock_vec()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
**DbDeadlockException or DB_LOCK_DEADLOCK**

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

**DbDeadlockException (page 365)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

**DbLockNotGrantedException (page 366)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**DbLockNotGrantedException or 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.

**DbLockNotGrantedException (page 366)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**EINVAL**

An invalid flag value or parameter was specified.

**ENOMEM**

The maximum number of locks has been reached.

**Class**

DbEnv, DbLock

**See Also**

Locking Subsystem and Related Methods (page 372)
Chapter 8. The DbLsn Handle

#include <db_cxx.h>

class DbLsn : public DB_LSN { ... };  

The DbLsn object is a log sequence number which specifies a unique location in a log file. A DbLsn 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

<table>
<thead>
<tr>
<th>Logging Subsystem and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::log_archive()</code></td>
<td>List log and database files</td>
</tr>
<tr>
<td><code>DbEnv::log_file()</code></td>
<td>Map Log Sequence Numbers to log files</td>
</tr>
<tr>
<td><code>DbEnv::log_flush()</code></td>
<td>Flush log records</td>
</tr>
<tr>
<td><code>DbEnv::log_printf()</code></td>
<td>Append informational message to the log</td>
</tr>
<tr>
<td><code>DbEnv::log_put()</code></td>
<td>Write a log record</td>
</tr>
<tr>
<td><code>DbEnv::log_stat()</code></td>
<td>Return log subsystem statistics</td>
</tr>
<tr>
<td><code>DbEnv::log_stat_print()</code></td>
<td>Print log subsystem statistics</td>
</tr>
<tr>
<td><code>DbEnv::log_compare()</code></td>
<td>Compare two Log Sequence Numbers</td>
</tr>
</tbody>
</table>

## Logging Subsystem Cursors

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::log_cursor()</code></td>
<td>Create a log cursor handle</td>
</tr>
</tbody>
</table>

## The DbLogc Handle

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbLogc::close()</code></td>
<td>Close a log cursor</td>
</tr>
<tr>
<td><code>DbLogc::get()</code></td>
<td>Retrieve a log record</td>
</tr>
</tbody>
</table>

## Logging Subsystem Configuration

<table>
<thead>
<tr>
<th>method</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::log_set_config()</code>, <code>DbEnv::log_get_config()</code></td>
<td>Configure the logging subsystem</td>
</tr>
<tr>
<td><code>DbEnv::set_lg_bsize()</code>, <code>DbEnv::get_lg_bsize()</code></td>
<td>Set/get log buffer size</td>
</tr>
<tr>
<td><code>DbEnv::set_lg_dir()</code>, <code>DbEnv::get_lg_dir()</code></td>
<td>Set/get the environment logging directory</td>
</tr>
<tr>
<td><code>DbEnv::set_lg_filemode()</code>, <code>DbEnv::get_lg_filemode()</code></td>
<td>Set/get log file mode</td>
</tr>
<tr>
<td><code>DbEnv::set_lg_max()</code>, <code>DbEnv::get_lg_max()</code></td>
<td>Set/get log file size</td>
</tr>
<tr>
<td><code>DbEnv::set_lg_regionmax()</code>, <code>DbEnv::get_lg_regionmax()</code></td>
<td>Set/get logging region size</td>
</tr>
</tbody>
</table>
DbEnv::get_lg_bsize()

```cpp
#include <db_cxx.h>

int DbEnv::get_lg_bsize(u_int32_t *lg_bsizep);
```

The `DbEnv::get_lg_bsize()` method returns the size of the log buffer, in bytes. You can manage this value using the `DbEnv::set_lg_bsize()` (page 442) method.

The `DbEnv::get_lg_bsize()` method may be called at any time during the life of the application.

The `DbEnv::get_lg_bsize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*lg_bsizep*

The `DbEnv::get_lg_bsize()` method returns the size of the log buffer, in bytes in `lg_bsizep`.

**Class**

`DbEnv`, `DbLogc`, `DbLsn`

**See Also**

`Logging Subsystem and Related Methods (page 417)`, `DbEnv::set_lg_bsize() (page 442)`
DbEnv::get_lg_dir()  

```cpp
#include <db_cxx.h>

int DbEnv::get_lg_dir(const char **dirp);
```

The DbEnv::get_lg_dir() method returns the log directory, which is the location for logging files. You can manage this value using the DbEnv::set_lg_dir() (page 444) method.

The DbEnv::get_lg_dir() method may be called at any time during the life of the application.

The DbEnv::get_lg_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*dirp*  

The DbEnv::get_lg_dir() method returns a reference to the log directory in *dirp*.

**Class**

DbEnv, DbLogc, DbLsn

**See Also**

Logging Subsystem and Related Methods (page 417), DbEnv::set_lg_dir() (page 444)
### DbEnv::get_lg_filemode()

```cpp
#include <db_cxx.h>

int DbEnv::get_lg_filemode(int *lg_modep);
```

The DbEnv::set_lg_filemode() method returns the log file mode. You can manage this value using the DbEnv::set_lg_filemode() (page 446) method.

The DbEnv::set_lg_filemode() method may be called at any time during the life of the application.

The DbEnv::set_lg_filemode() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **lg_modep**

  The DbEnv::set_lg_filemode() method returns the log file mode in `lg_modep`.

**Class**

- DbEnv, DbLogc, DbLsn

**See Also**

Logging Subsystem and Related Methods (page 417), DbEnv::set_lg_filemode() (page 446)
DbEnv::get_lg_max()

```c
#include <db_cxx.h>

int
DbEnv::get_lg_max(u_int32_t *lg_maxp);
```

The `DbEnv::get_lg_max()` method returns the maximum log file size. You can manage this value using the `DbEnv::set_lg_max()` (page 447) method.

The `DbEnv::get_lg_max()` method may be called at any time during the life of the application.

The `DbEnv::get_lg_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **lg_maxp**
  
  The `DbEnv::get_lg_max()` method returns the maximum log file size in `lg_maxp`.

**Class**

- `DbEnv`, `DbLogc`, `DbLsn`

**See Also**

 Logging Subsystem and Related Methods (page 417), `DbEnv::set_lg_max()` (page 447)
DbEnv::get_lg_regionmax()

```c
#include <db_cxx.h>

int DbEnv::get_lg_regionmax(u_int32_t *lg_regionmaxp);
```

The `DbEnv::get_lg_regionmax()` method returns the size of the underlying logging subsystem region. You can manage this value using the `DbEnv::set_lg_regionmax()` method.

The `DbEnv::get_lg_regionmax()` method may be called at any time during the life of the application.

The `DbEnv::get_lg_regionmax()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **lg_regionmaxp**

  The `DbEnv::get_lg_regionmax()` method returns the size of the underlying logging subsystem region in `lg_regionmaxp`.

**Class**

- `DbEnv`, `DbLogc`, `DbLsn`

**See Also**

- Logging Subsystem and Related Methods (page 417), `DbEnv::set_lg_regionmax()` (page 449)
DbEnv::log_archive()

#include <db_cxx.h>

int
DbEnv::log_archive(char *(*listp)[], u_int32_t flags);

The DbEnv::log_archive() method returns an array of log or databasefilenames.

By default, DbEnv::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 DbEnv::set_alloc() (page 294) 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 DbEnv::log_cursor() (page 425) 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, DbEnv::log_put() (page 431)
and DbTxn::abort() (page 685) ) 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 DbEnv::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 DbEnv::log_archive() method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 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, DbEnv::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 DbEnv::log_archive() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLogc, DbLsn

**See Also**

Logging Subsystem and Related Methods (page 417)
**DbEnv::log_cursor()**

```cpp
#include <db_cxx.h>

int DbEnv::log_cursor(DbLogc **cursorp, u_int32_t flags);
```

The `DbEnv::log_cursor()` method returns a created log cursor.

The `DbEnv::log_cursor()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::log_cursor()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv, DbLogc, DbLsn`

**See Also**

[Logging Subsystem and Related Methods (page 417)]
DbEnv::log_file()

```cpp
#include <db_cxx.h>

int DbEnv::log_file(const DbLsn *lsn, char *namep, size_t len);
```

The `DbEnv::log_file()` method maps `DbLsn` structures to filenames, returning the name of the file containing the record named by `Isn`.

This mapping of `DbLsn` structures to files is needed for database administration. For example, a transaction manager typically records the earliest `DbLsn` needed for restart, and the database administrator may want to archive log files to tape when they contain only `DbLsn` entries before the earliest one needed for restart.

The `DbEnv::log_file()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `lsn`
  The `lsn` parameter is the `DbLsn` 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, `DbEnv::log_file()` will fail. (Log filenames are always 14 characters long.)

**Errors**

The `DbEnv::log_file()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  If the supplied buffer was too small to hold the log filename; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv, DbLogc, DbLsn`

**See Also**

`Logging Subsystem and Related Methods (page 417)`
DbEnv::log_flush()

#include <db_cxx.h>

int DbEnv::log_flush(const DbLsn *lsn);

The DbEnv::log_flush() method writes log records to disk.

The DbEnv::log_flush() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

lsn

All log records with DbLsn 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 DbEnv::log_flush() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DbEnv, DbLogc, DbLsn

See Also

Logging Subsystem and Related Methods (page 417)
```cpp
#include <db_cxx.h>

int DbEnv::log_get_config(u_int32_t which, int *onoffp)
```

The `DbEnv::log_get_config()` method returns whether the specified `which` parameter is currently set or not. You can manage this value using the `DbEnv::log_set_config()` method.

The `DbEnv::log_get_config()` method may be called at any time during the life of the application.

The `DbEnv::log_get_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_EXT_FILE**
  
  Enables full logging of external file data.

- **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_NOSYNC**
  
  The transaction log is not flushed from the operating system cache to stable storage when the logging system switches log files or a durable transaction commits. This means that transactions exhibit the ACI (atomicity, consistency, and isolation) properties, but...
only partially support D (durability); that is, database integrity will be maintained if the application fails and the environment is recovered, but not if the system fails. All database files must be verified and/or restored from a replication group master or archival backup after system failure.

- **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**

DbEnv

**See Also**

Logging Subsystem and Related Methods (page 417), DbEnv::log_set_config() (page 433)
DbEnv::log_printf()

```cpp
#include <db_cxx.h>

int DbEnv::log_printf(DB_TXN *txnid, const char *fmt, ...);
```

The `DbEnv::log_printf()` method appends an informational message to the Berkeley DB database environment log files.

The `DbEnv::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 `DbEnv::log_printf()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*txnid*

If the logged message refers to an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); 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 `DbEnv::log_printf()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbLogc`, `DbLsn`

**See Also**

`Logging Subsystem and Related Methods (page 417)`
DbEnv::log_put()

```cpp
#include <db_cxx.h>

int DbEnv::log_put(DbLsn *lsn, const Dbt *data, u_int32_t flags);
```

The `DbEnv::log_put()` method appends records to the log. The `DbLsn` of the put record is returned in the `lsn` parameter.

The `DbEnv::log_put()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lsn**

The `lsn` parameter references memory into which the `DbLsn` 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 `DbLsn` 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 `DbLsn` values less than or equal to the one being "put" are on disk before `DbEnv::log_put()` returns.

**Errors**

The `DbEnv::log_put()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLogc`, `DbLsn`
See Also

Logging Subsystem and Related Methods (page 417)
**DbEnv::log_set_config()**

```cpp
#include <db_cxx.h>

int DbEnv::log_set_config(u_int32_t flags, int onoff);
```

The `DbEnv::log_set_config()` method configures the Berkeley DB logging subsystem.

The `DbEnv::log_set_config()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::log_set_config()` method may be called at any time during the life of the application.

The `DbEnv::log_set_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_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 `DbEnv::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_EXT_FILE**

  Enables full logging of external file data. Only use this flag if using replication or running a hot backup; otherwise, it will impose a performance penalty on external file operations.

  This flag is off by default, unless Replication is enabled for the environment. In that case, it is on by default.
The DB_LOG_EXT_FILE flag may be used to configure Berkeley DB at any time during the life of the application.

- **DB_LOG_DIRECT**

Turn off system buffering of Berkeley DB log files to avoid double caching.

Calling `DbEnv::log_set_config()` with the DB_LOG_DIRECT flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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 `DbEnv::log_set_config()` with the DB_LOG_DSYNC flag only affects the specified `DbEnv` handle (and any other Berkeley DB handles opened within the scope of that handle). For consistent behavior across the environment, all `DbEnv` 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_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 `DbEnv::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 `DbEnv::open()` (page 286) method is called.

- **DB_LOG_NOSYNC**

  If set, the transaction log is not flushed from the operating system cache to stable storage when a transaction commits or the logging system switches log files. This can further increase performance when all of the applications in an environment use less-than-durable transactions (DB_TXN_NOSYNC and DB_TXN_WRITE_NOSYNC).

  Turning on `DB_LOG_NOSYNC` removes the durability guarantee of current or future `DB_TXN_SYNC` transactions, as well as any transactions which have been active since the last checkpoint. All database files must be verified and/or restored from a replication group master or archival backup after system failure.

  Turning off `DB_LOG_NOSYNC` may be done at any time. Current and future `DB_TXN_SYNC` transactions will return to full ACID characteristics once a checkpoint or log file switch occurs.

  Calling `DbEnv::log_set_config()` with the `DB_LOG_NOSYNC` flag affects the database environment, including all threads of control accessing the database environment.

  The `DB_LOG_NOSYNC` flag may be used to configure Berkeley DB at any time. `DbEnv::open()` (page 286) 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 `DbEnv::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 `DbEnv::log_set_config()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Logging Subsystem and Related Methods (page 417)
**DbEnv::log_stat()**

```cpp
#include <db_cxx.h>

int DbEnv::log_stat(DB_LOG_STAT **statp, u_int32_t flags);
```

The `DbEnv::log_stat()` method returns the logging subsystem statistics.

The `DbEnv::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 `DbEnv::set_alloc()` (page 294) 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_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_fileid_init;`
  - The initial allocated file logging identifiers.

- `u_int32_t st_lg_bsize;`
  - The in-memory log record cache size.

- `u_int32_t st_lg_size;`
  - The log file size.

- `u_int32_t st_magic;`
  - The magic number that identifies a file as a log file.

- `u_int32_t st_maxcommitperflush;`
The maximum number of commits contained in a single log flush.

- u_int32_t st_maxnfileid;
  The maximum number of file logging identifiers used.

- u_int32_t st_mincommitperflush;
  The minimum number of commits contained in a single log flush that contained a commit.

- int st_mode;
  The mode of any created log files.

- u_int32_t st_nfileid;
  The current number of file logging identifiers.

- uintmax_t st_rcount;
  The number of times the log has been read from disk.

- uintmax_t st_record;
  The number of records written to this log.

- roff_t st_regsize;
  The region size, 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.

- uintmax_t st_scount;
  The number of times the log has been flushed to disk.

- u_int32_t st_version;
  The version of the log file type.

- u_int32_t st_w_bytes;
  The number of bytes over and above st_w_mbytes written to this log.

- u_int32_t st_w_mbytes;
The number of megabytes written to this log.

- `u_int32_t st_wc_bytes;
  The number of bytes over and above `st_wc_mbytes` written to this log since the last checkpoint.

- `u_int32_t st_wc_mbytes;
  The number of megabytes written to this log since the last checkpoint.

- `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_wcount;
  The number of times the log has been written to disk.

The `DbEnv::log_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::log_stat()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::log_stat()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  An invalid flag value or parameter was specified.

**Class**

DbEnv, DbLogc, DbLsn
See Also

Logging Subsystem and Related Methods (page 417)
DbEnv::log_stat_print()

```cpp
#include <db_cxx.h>

int
DbEnv::log_stat_print(u_int32_t flags);
```

The `DbEnv::log_stat_print()` method displays the logging subsystem statistical information, as described for the `DbEnv::log_stat()` method. The information is printed to a specified output channel (see the `DbEnv::set_msgfile()` (page 342) method for more information), or passed to an application callback function (see the `DbEnv::set_msgcall()` (page 340) method for more information).

The `DbEnv::log_stat_print()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::log_stat_print()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_ALLOC`
  
  Display allocation information. To display allocation information, both `DB_STAT_ALLOC` and `DB_STAT_ALL` need to be set.

**Class**

DbEnv, DbLogc, DbLsn

**See Also**

Logging Subsystem and Related Methods (page 417)
DbEnv::set_lg_bsize()

```cpp
#include <db_cxx.h>

int
DbEnv::set_lg_bsize(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 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 DbEnv::set_lg_bsize() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lg_bsize() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_lg_bsize() will be ignored.

The DbEnv::set_lg_bsize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lg_bsize**

The **lg_bsize** parameter is the size of the in-memory log buffer, in bytes.

**Errors**

The DbEnv::set_lg_bsize() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

An invalid flag value or parameter was specified.

Class

DbEnv, DbLogc, DbLsn

See Also

Logging Subsystem and Related Methods (page 417)
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. Note that if you use this method for your application, and you also want to use the db_recover (page 748), db_printlog (page 746), db_archive (page 722), or db_log_verify (page 743) utilities, then you should set create a DB_CONFIG file and set the "set_lg_dir" parameter in it.

The `DbEnv::set_lg_dir()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_lg_dir()` method may not be called after the `DbEnv::open()` (page 286) method is called. If the database environment already exists when `DbEnv::open()` (page 286) is called, the information specified to `DbEnv::set_lg_dir()` must be consistent with the existing environment or corruption can occur.

The `DbEnv::set_lg_dir()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dir**

The `dir` parameter is the directory used to store the logging files. This directory must currently exist at environment open time.

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 `DbEnv::set_lg_dir()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL
If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

Class
DbEnv, DbLogc, DbLsn

See Also
Logging Subsystem and Related Methods (page 417)
DbEnv::set_lg_filemode()

```c
#include <db_cxx.h>

int
DbEnv::set_lg_filemode(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 DbEnv::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 DbEnv::set_lg_filemode() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lg_filemode() method may be called at any time during the life of the application.

The DbEnv::set_lg_filemode() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lg_filemode**

The *lg_filemode* parameter is the absolute mode of the created log file.

**Class**

DbEnv, DbLogc, DbLsn

**See Also**

Logging Subsystem and Related Methods (page 417)
DbEnv::set_lg_max()

```cpp
#include <db_cxx.h>

int
DbEnv::set_lg_max(u_int32_t lg_max);
```

Sets the maximum size of a single file in the log, in bytes. Because DbLsn 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 DbEnv::set_lg_max() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::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 DbEnv::set_lg_max() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**lg_max**

The \texttt{lg\_max} parameter is the size of a single log file, in bytes.
Errors

The `DbEnv::set_lg_max()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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

`DbEnv`, `DbLogc`, `DbLsn`

See Also

`Logging Subsystem and Related Methods (page 417)`
DbEnv::set_lg_regionmax()

```c
#include <db_cxx.h>

int
DbEnv::set_lg_regionmax(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 DbEnv::set_lg_regionmax() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_lg_regionmax() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::set_lg_regionmax() will be ignored.

The DbEnv::set_lg_regionmax() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_lg_regionmax() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv, DbLogc, DbLsn
See Also

Logging Subsystem and Related Methods (page 417)
The DbLogc Handle

```
#include <db_cxx.h>

int
DbEnv::log_cursor(DbLogc **cursorp, u_int32_t flags);
```

The DbLogc 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 DbLogc::close() (page 452) method is called, the handle may not be accessed again, regardless of that method's return.

For more information, see the DbLsn handle.
DbLogc::close()

```cpp
#include <db_cxx.h>

int DbLogc::close(u_int32_t flags);
```

The `DbLogc::close()` method discards the log cursor. After `DbLogc::close()` has been called, regardless of its return, the cursor handle may not be used again.

The `DbLogc::close()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

flags

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

**Errors**

The `DbLogc::close()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLogc`, `DbLsn`

**See Also**

`Logging Subsystem and Related Methods (page 417)`
DbLogc::get()

```c
#include <db_cxx.h>

int
DbLogc::get(DbLsn *lsn, Dbt *data, u_int32_t flags);
```

The `DbLogc::get()` method returns records from the log.

Unless otherwise specified, the `DbLogc::get()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbLsn` value of the record retrieved. When `flag` is set to `DB_SET`, the `lsn` parameter is the `DbLsn` 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 `DbLsn` of the record returned.

  The `DbLogc::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 `DbLsn` of the record returned.

  The `DbLogc::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 `DbLsn` of the record returned.

  If the cursor has not been initialized via `DB_FIRST`, `DB_LAST`, `DB_SET`, `DB_NEXT`, or `DB_PREV`, `DbLogc::get()` will return the first record in the log.

  The `DbLogc::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 `DbLsn` of the record returned.

  If the cursor has not been initialized via `DB_FIRST`, `DB_LAST`, `DB_SET`, `DB_NEXT`, or `DB_PREV`, `DbLogc::get()` will return the last record in the log.

  The `DbLogc::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 `DbLogc::get()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`, `DbLogc`, `DbLsn`

**See Also**

*Logging Subsystem and Related Methods (page 417)*
DbEnv::log_compare()

```cpp
#include <db_cxx.h>

static int
DbEnv::log_compare(const DbLsn *lsn0, const DbLsn *lsn1);
```

The `DbEnv::log_compare()` method allows the caller to compare two `DbLsn` 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 `DbLsn` structures to be compared.

- **lsn1**
  The `lsn1` parameter is one of the `DbLsn` structures to be compared.

**Class**

- `DbEnv`, `DbLogc`, `DbLsn`

**See Also**

- [Logging Subsystem and Related Methods (page 417)]
Chapter 9. The DbMpoolFile Handle

```cpp
#include <db_cxx.h>

class DbMpoolFile {
public:
    DB_MPOOLFILE *DbMpoolFile::get_DB_MPOOLFILE();
    const DB_MPOOLFILE *DbMpoolFile::get_const_DB_MPOOLFILE() const;
    ...
};
```

The memory pool interfaces for the Berkeley DB database environment are methods of the `DbEnv` handle. The `DbEnv` memory pool methods and the `DB_MPOOLFILE` class provide general-purpose, page-oriented buffer management of files. Although designed to work with the other `Db` classes, 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 `DbMpoolFile::get()` (page 495) are pinned in the cache until they are returned to the control of the cache using the `DbMpoolFile::put()` (page 500) method.

The `DbMpoolFile` object is the handle for a file in the cache. The handle is not free-threaded. Once the `DbMpoolFile::close()` (page 494) method is called, the handle may not be accessed again, regardless of that method's return.
## Memory Pools and Related Methods

<table>
<thead>
<tr>
<th>Memory Pools and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbMemoryException</td>
<td>Exception class for insufficient memory</td>
</tr>
<tr>
<td>Db::get_mpf()</td>
<td>Return the DbMpoolFile for a Db</td>
</tr>
<tr>
<td>DbEnv::memp_stat()</td>
<td>Return cache statistics</td>
</tr>
<tr>
<td>DbEnv::memp_stat_print()</td>
<td>Print cache statistics</td>
</tr>
<tr>
<td>DbEnv::memp_sync()</td>
<td>Flush all pages from the cache</td>
</tr>
<tr>
<td>DbEnv::memp_trickle()</td>
<td>Flush some pages from the cache</td>
</tr>
</tbody>
</table>

### Memory Pool Configuration

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbEnv::memp_register()</td>
<td>Register a custom file type</td>
</tr>
<tr>
<td>DbEnv::set_cache_max(), DbEnv::get_cache_max()</td>
<td>Set/get the maximum cache size</td>
</tr>
<tr>
<td>DbEnv::set_cachesize(), DbEnv::get_cachesize()</td>
<td>Set/get the environment cache size</td>
</tr>
<tr>
<td>DbEnv::set_mp_max_openfd(), DbEnv::get_mp_max_openfd()</td>
<td>Set/get the maximum number of open file descriptors</td>
</tr>
<tr>
<td>DbEnv::set_mp_max_write(), DbEnv::get_mp_max_write()</td>
<td>Set/get the maximum number of sequential disk writes</td>
</tr>
<tr>
<td>DbEnv::set_mp_mmapsize(), DbEnv::get_mp_mmapsize()</td>
<td>Set/get maximum file size to memory map when opened read-only</td>
</tr>
<tr>
<td>DbEnv::set_mp_mtxcount(), DbEnv::get_mp_mtxcount()</td>
<td>Set/get the number of mutexes allocated to the hash table</td>
</tr>
<tr>
<td>DbEnv::set_mp_pagesize(), DbEnv::get_mp_pagesize()</td>
<td>Set/get page size to configure the buffer pool</td>
</tr>
<tr>
<td>DbEnv::set_mp_tablesize(), DbEnv::get_mp_tablesize()</td>
<td>Set/get the hash table size</td>
</tr>
<tr>
<td>DbEnv::set_region_dir(), DbEnv::get_region_dir()</td>
<td>Set/get the environment region directory</td>
</tr>
</tbody>
</table>

### Memory Pool Files

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbMpoolFile::close()</td>
<td>Close a file in the cache</td>
</tr>
<tr>
<td>DbMpoolFile::get()</td>
<td>Get page from a file in the cache</td>
</tr>
<tr>
<td>DbMpoolFile::open()</td>
<td>Open a file in the cache</td>
</tr>
<tr>
<td>DbMpoolFile::put()</td>
<td>Return a page to the cache</td>
</tr>
<tr>
<td>DbMpoolFile::sync()</td>
<td>Flush pages from a file from the cache</td>
</tr>
</tbody>
</table>

### Memory Pool File Configuration
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbMpoolFile::set_clear_len(),</td>
<td>Set/get number of bytes to clear when</td>
</tr>
<tr>
<td>DbMpoolFile::get_clear_len()</td>
<td>creating a new page</td>
</tr>
<tr>
<td>DbMpoolFile::set_fileid(),</td>
<td>Set/get file unique identifier</td>
</tr>
<tr>
<td>DbMpoolFile::get_fileid()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_flags(),</td>
<td>Set/get file options</td>
</tr>
<tr>
<td>DbMpoolFile::get_flags()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_fype(),</td>
<td>Set/get file type</td>
</tr>
<tr>
<td>DbMpoolFile::get_fype()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_lsn_offset(),</td>
<td>Set/get file log-sequence-number offset</td>
</tr>
<tr>
<td>DbMpoolFile::get_lsn_offset()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_maxsize(),</td>
<td>Set/get maximum file size</td>
</tr>
<tr>
<td>DbMpoolFile::get_maxsize()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_pgcookie(),</td>
<td>Set/get file cookie for pgin/pgout</td>
</tr>
<tr>
<td>DbMpoolFile::get_pgcookie()</td>
<td></td>
</tr>
<tr>
<td>DbMpoolFile::set_priority(),</td>
<td>Set/get cache file priority</td>
</tr>
<tr>
<td>DbMpoolFile::get_priority()</td>
<td></td>
</tr>
</tbody>
</table>
Db::get_mpf()

```cpp
#include <db_cxx.h>
DbMpoolFile *
Db::get_mpf();
```

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 457)
DbEnv::get_cache_max()

```c
#include <db_cxx.h>

int DbEnv::get_cache_max(u_int32_t *gbytesp, u_int32_t *bytesp);
```

The DbEnv::get_cache_max() method returns the maximum size of the cache as set using the DbEnv::set_cache_max() method.

The DbEnv::get_cache_max() method may be called at any time during the life of the application.

The DbEnv::get_cache_max() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv

**See Also**

Database Environments and Related Methods (page 228), DbEnv::set_cache_max() (page 480)
DbEnv::get_cachesize()

#include <db_cxx.h>

int
DbEnv::get_cachesize(u_int32_t *gbytesp, u_int32_t *bytesp, int *ncachep);

The DbEnv::get_cachesize() method returns the current size and composition of the cache, as set using the DbEnv::set_cachesize() (page 482) method.

The DbEnv::get_cachesize() method may be called at any time during the life of the application.

The DbEnv::get_cachesize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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

DbEnv

See Also

Memory Pools and Related Methods (page 457), Database Environments and Related Methods (page 228), DbEnv::set_cachesize() (page 482)
DbEnv::get_mp_max_openfd()

```c
#include <db_cxx.h>

int
DbEnv::get_mp_max_openfd(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 DbEnv::set_mp_max_openfd() (page 484) method.

The DbEnv::get_mp_max_openfd() method may be called at any time during the life of the application.

The DbEnv::get_mp_max_openfd() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*maxopenfdp*

The DbEnv::get_mp_max_openfd() method returns the maximum number of file descriptors open in `maxopenfdp`.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbEnv::set_mp_max_openfd() (page 484)
DbEnv::get_mp_max_write()

```
#include <db_cxx.h>

int DbEnv::get_mp_max_write(int *maxwritep, db_timeout_t *maxwrite_sleepp);
```

The DbEnv::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 DbEnv::set_mp_max_write() (page 485) method.

The DbEnv::get_mp_max_write() method may be called at any time during the life of the application.

The DbEnv::get_mp_max_write() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbEnv::set_mp_max_write() (page 485)
DbEnv::get_mp_mmapsize()

```c
#include <db_cxx.h>

int DbEnv::get_mp_mmapsize(size_t *mp_mmapsizep);
```

The `DbEnv::get_mp_mmapsize()` method returns the maximum file size, in bytes, for a file to be mapped into the process address space. This value can be managed using the `DbEnv::set_mp_mmapsize()` method.

The `DbEnv::get_mp_mmapsize()` method may be called at any time during the life of the application.

The `DbEnv::get_mp_mmapsize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **mp_mmapsizep**

  The `DbEnv::get_mp_mmapsize()` method returns the maximum file map size in `mp_mmapsizep`.

**Class**

- `DbEnv`, `DbMpoolFile`

**See Also**

- Memory Pools and Related Methods (page 457), `DbEnv::set_mp_mmapsize()` (page 487)
DbEnv::get_mp_mtxcount()

```cpp
#include <db_cxx.h>

int DbEnv::get_mp_mtxcount(u_int32_t mtxcount);
```

The `DbEnv::get_mp_mtxcount()` method returns the number of mutexes allocated for the hash table in the buffer pool.

**Parameters**

- **mtxcount**
  
  This parameter specifies the number of mutexes allocated for the hash table in the buffer pool.

**Class**

- DbEnv, DbMpoolFile

**See Also**

- Memory Pools and Related Methods (page 457), DbEnv::set_mp_mtxcount() (page 489)
DbEnv::get_mp_pagesize()

```cpp
#include <db_cxx.h>

int DbEnv::get_mp_pagesize(u_int32_t *pagesizep);
```

The `DbEnv::get_mp_pagesize()` method returns the assumed page size used to configure the buffer pool.

The `DbEnv::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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbEnv::set_mp_pagesize() (page 490)
DbEnv::get_mp_tablesize()

```cpp
#include <db_cxx.h>

int DbEnv::get_mp_tablesize(u_int32_t tablesizet);
```

The `DbEnv::get_mp_tablesize()` method returns the hash table size in the buffer pool.

**Parameters**

- **tablesizet**
  
  This parameter specifies the hash table size in the buffer pool.

**Class**

- `DbEnv`, `DbMpoolFile`

**See Also**

- Memory Pools and Related Methods (page 457), `DbEnv::set_mp_tablesize()` (page 491)
DbEnv::memp_fcreate()

```c
#include <db_cxx.h>

int
DbEnv::memp_fcreate(DbMpoolFile **dbmfp, u_int32_t flags);
```

The `DbEnv::memp_fcreate()` method creates a `DbMpoolFile` 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 `DbMpoolFile::close()` (page 494) method will discard the returned handle.

The `DbEnv::memp_fcreate()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`dbmfp`

The `DbEnv::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**

`DbEnv, DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`
**DbEnv::memp_register()**

```c
#include <db_cxx.h>

extern "C" {
    typedef int (*pgin_fcn_type)(DB_ENV *dbenv,
        db_pgno_t pgno, void *pgaddr, DBT *pgcookie);
    typedef int (*pgout_fcn_type)(DB_ENV *dbenv,
        db_pgno_t pgno, void *pgaddr, DBT *pgcookie);
};

int DbEnv::memp_register(int ftype,
    pgin_fcn_type pgin_fcn, pgout_fcn_type pgout_fcn);
```

The DbEnv::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 DbMpoolFile::get() (page 495) 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 DbEnv::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 DbEnv::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 75) registers them separately.)

If a thread or process does not call DbEnv::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, DbEnv::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 DbEnv::memp_register() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbMpoolFile::set_pgcookie()` (page 520) 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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbEnv::memp_stat()

```c
#include <db_cxx.h>

int
DbEnv::memp_stat(DB_MPOOL_STAT **gsp,
    DB_MPOOL_FSTAT *(*fsp)[]), u_int32_t flags);
```

The `DbEnv::memp_stat()` method returns the memory pool (that is, the buffer cache) subsystem statistics.

The `DbEnv::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 `DbEnv::set_alloc()` (page 294) 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 `DbEnv::set_cache_max()` (page 480) method.

- `roff_t st_regsize;`
  
  Individual cache size, in bytes.

- `roff_t st_regmax;`
  
  The maximum size, in bytes, of the mutex region.

- `size_t st_mmapsize;`
Maximum memory-mapped file size.

- `int st_maxopenfd;`
  Maximum number of open file descriptors.

- `int st_maxwrite;`
  The maximum number of sequential write operations scheduled by the library when flushing dirty pages from the cache.

- `db_timeout_t st_maxwrite_sleep;`
  The number of microseconds the thread of control should pause before scheduling further write operations.

- `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 `DbEnv::memp_trickle()` method.
• u_int32_t st_pages;
  Pages in the cache.

• size_t st_pagesize;
  Page size in bytes.

• u_int32_t st_page_clean;
  Clean pages currently in the cache.

• u_int32_t st_page_dirty;
  Dirty pages currently in the cache.

• u_int32_t st_hash_buckets;
  Number of hash buckets in buffer hash table.

• uintmax_t st_hash_examined;
  Total number of hash elements traversed during hash table lookups.

• u_int32_t st_hash_longest;
  Longest chain ever encountered in buffer hash table lookups.

• u_int32_t st_hash_mutexes;
  The number of hash bucket mutexes in the buffer hash table.

• uintmax_t st_hash_nowait;
  Number of times that a thread of control was able to obtain a hash bucket lock without waiting.

• u_int32_t st_hash_searches;
  Total number of buffer hash table lookups.

• 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;

  The number of times that a thread of control was forced to wait before obtaining a cache region mutex.

- `uintmax_t st_region_nowait;

  The 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_reused;

  The number of outdated intermediate versions reused.

- `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.
- uintmax_t st_backup_spins;
  Spins while trying to back up the file.

The DbEnv::memp_stat() method may not be called before the DbEnv::open() (page 286) method is called.

The DbEnv::memp_stat() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::memp_stat() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

  EINVAL
  
  An invalid flag value or parameter was specified.

Class

  DbEnv, DbMpoolFile

See Also

  Memory Pools and Related Methods (page 457)
DbEnv::memp_stat_print()

```cpp
#include <db_cxx.h>

int
DbEnv::memp_stat_print(u_int32_t flags);
```

The DbEnv::memp_stat_print() method displays cache subsystem statistical information, as described for the DbEnv::memp_stat() (page 471) method. The information is printed to a specified output channel (see the DbEnv::set_msgfile() (page 342) method for more information), or passed to an application callback function (see the DbEnv::set_msgcall() (page 340) method for more information).

The DbEnv::memp_stat_print() method may not be called before the DbEnv::open() (page 286) method is called.

The DbEnv::memp_stat_print() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_ALLOC**
  
  Display allocation information. To display allocation information, both DB_STAT_ALLOC and DB_STAT_ALL need to be set.

- **DB_STAT_CLEAR**
  
  Reset statistics after displaying their values.

- **DB_STAT_MEMP_HASH**
  
  Display the buffers with hash chains.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbEnv::memp_sync()

#include <db_cxx.h>

int DbEnv::memp_sync(DbLsn *lsn);

The DbEnv::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 DbEnv::memp_sync() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 (DbLsn) less than the lsn parameter are written to disk. If lsn is NULL, all modified pages in the cache are written to disk.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbEnv::memp_trickle()

```cpp
#include <db_cxx.h>

int DbEnv::memp_trickle(int percent, int *nwrotep);
```

The `DbEnv::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 `DbEnv::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 `DbEnv::memp_trickle()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::memp_trickle()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  An invalid flag value or parameter was specified.

**Class**

- `DbEnv`, `DbMpoolFile`

**See Also**

- `Memory Pools and Related Methods (page 457)`
DbEnv::set_cache_max()

```cpp
#include <db_cxx.h>

int DbEnv::set_cache_max(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 DbEnv::set_cachesize() (page 482) 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 maximum cache size in bytes, specified in two parts: the gigabytes of cache and the additional bytes of cache. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The DbEnv::set_cache_max() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::set_cache_max() method must be called prior to opening the database environment.

The DbEnv::set_cache_max() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_cache_max() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv
See Also

Database Environments and Related Methods (page 228)
DbEnv::set_cachesize()

```cpp
#include <db_cxx.h>

int
DbEnv::set_cachesize(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 `DbEnv::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 `DbEnv::set_cache_max()` (page 480). 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 overwrite configuration done before that time.

The `DbEnv::set_cachesize()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

When used with sliced databases, the `DbEnv::set_cachesize()` method must be called for each slice environment. That is, for each slice, there is a sub-environment, and this method must be called on each such environment individually. Use `DbEnv::get_slices()` (page 272) to retrieve an array of slice environments.

The `DbEnv::set_cachesize()` method may be called at any time during the life of the application.

The `DbEnv::set_cachesize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_cachesize()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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

`DbEnv`

See Also

`Database Environments and Related Methods (page 228)`
DbEnv::set_mp_max_openfd()

```
#include <db_cxx.h>

int DbEnv::set_mp_max_openfd(int maxopenfd);
```

The `DbEnv::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_mp_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 `DbEnv::set_mp_max_openfd()` (page 484) method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::set_mp_max_openfd()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_mp_max_openfd()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`
DbEnv::set_mp_max_write()

```c
#include <db_cxx.h>

int DbEnv::set_mp_max_write(int maxwrite, db_timeout_t maxwrite_sleep);
```

The `DbEnv::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 `DbEnv::set_mp_max_write()` method configures a database environment, not only operations performed using the specified DbEnv handle.

The `DbEnv::set_mp_max_write()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_mp_max_write()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbMpoolFile
See Also

Memory Pools and Related Methods (page 457)
DbEnv::set_mp_mmapsize()

```cpp
#include <db_cxx.h>

int
DbEnv::set_mp_mmapsize(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 DbEnv::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 DbEnv::set_mp_mmapsize() method configures a database environment, not only
operations performed using the specified DbEnv handle.

The DbEnv::set_mp_mmapsize() method may be called at any time during the life of the
application.

The DbEnv::set_mp_mmapsize() method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::set_mp_mmapsize() method may fail and throw a DbException exception,
encapsulating one of the following non-zero errors, or return one of the following non-zero
effects:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv, DbMpoolFile
See Also

Memory Pools and Related Methods (page 457)
DbEnv::set_mp_mtxcount()

```c
#include <db_cxx.h>

int
DbEnv::set_mp_mtxcount(u_int32_t mtxcount);
```

The `DbEnv::set_mp_mtxcount()` method overrides the default number of mutexes for the hash table in each memory pool cache. The default is one mutex per hash bucket. Setting it to a lower number decreases the number of mutexes used and the amount of memory needed to store them at the expense of concurrency in the memory pool. This can also improve startup time. Setting a number greater than the number size of the hash table will waste mutexes and space.

You must call this method only before the environment is opened.

**Parameters**

**mtxcount**

Specifies the number of mutexes allocated to the buffer pool hash table.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

*Memory Pools and Related Methods (page 457)*, `DbEnv::get_mp_mtxcount()` (page 465)
DbEnv::set_mp_pagesize()

```cpp
#include <db_cxx.h>

int
DbEnv::set_mp_pagesize(u_int32_t pagesize);
```

The `DbEnv::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.

This method may be called 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**

`DbEnv`, `DbMpoolFile`

**See Also**

Memory Pools and Related Methods (page 457), `DbEnv::get_mp_pagesize()` (page 466)
DbEnv::set_mp_tablesize()

```cpp
#include <db_cxx.h>

int DbEnv::set_mp_tablesize(u_int32_t tablesiz);
```

The `DbEnv::set_mp_tablesize()` method overrides the calculated hash table size. This value is then internally adjusted to a nearby prime number in order to enhance the hashing algorithm.

This method may be called only before the environment is opened.

**Parameters**

`tablesiz`

The table size parameter specifies the size of the buffer pool hash table. It is adjusted to a near prime number to enhance the hashing algorithm.

**Class**

`DbEnv, DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457), DbEnv::get_mp_tablesize() (page 467)`
DbEnv::set_region_dir()

#include <db_cxx.h>

int
DbEnv::set_region_dir(const char *dir);

The path of a directory to be used as the location of disk-backed region files. Region files
created by the Memory Pool subsystem will be created in this directory.

If no region directory is specified, region files are created in the environment home directory.
See Berkeley DB File Naming for more information.

The database environment's region 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_region_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. Note that if you use this method for your application,
and you also want to use the various utilities like db_recover (page 748), then you should set
create a DB_CONFIG file and set the "set_region_dir" parameter in it.

The DbEnv::set_region_dir() method configures operations performed using the specified
DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::set_region_dir() method may not be called after the DbEnv::open() (page 286)
method is called. If the database environment already exists when DbEnv::open() (page 286) is
called, the information specified to DbEnv::set_region_dir() must be consistent with the
existing environment or corruption can occur.

The DbEnv::set_region_dir() method either returns a non-zero error value or throws an
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

dir

The dir parameter is the directory used to store the region files. This directory must currently
exist at environment open time.

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 DbEnv::set_region_dir() method may fail and throw a DbException exception,
encapsulating one of the following non-zero errors, or return one of the following non-zero
errors:

EINVAL

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value
or parameter was specified.
Class

DbEnv, DbMpoolFile

See Also

Memory Pools and Related Methods (page 457), DbEnv::get_region_dir() (page 511)
DbMpoolFile::close()

```c
#include <db_cxx.h>

int DbMpoolFile::close(u_int32_t flags);
```

The DbMpoolFile::close() method closes the source file indicated by the DbMpoolFile structure. Calling DbMpoolFile::close() does not imply a call to DbMpoolFile::sync() (page 502); that is, no pages are written to the source file as a result of calling DbMpoolFile::close().

If the DbMpoolFile was temporary, any underlying files created for this DbMpoolFile will be removed.

After DbMpoolFile::close() has been called, regardless of its return, the DbMpoolFile handle may not be accessed again.

The DbMpoolFile::close() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

flags

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

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbMpoolFile::get()

#include <db_cxx.h>

int DbMpoolFile::get(db_pgno_t *pgnoaddr,
                     DbTxn *txnid, u_int32_t flags, void **pagep);

The DbMpoolFile::get() method returns pages from the cache.

All pages returned by DbMpoolFile::get() will be retained (that is, latched) in the cache until a subsequent call to DbMpoolFile::put() (page 500). 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 exclusive 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 DbMpoolFile::set_clear_len() (page 512) method was called to specify other behavior before the file was opened.

The DbMpoolFile::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 DbMpoolFile::get() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbMpoolFile::put() (page 500).

- 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 DbMpoolFile::put() (page 500).

- **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 `DbEnv::memp_register()` (page 469), 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 `DbEnv::txn_begin()` (page 674); otherwise NULL. A transaction is required if the file is open for multiversion concurrency control by passing DB_MULTIVERSION to DbMpoolFile::open() (page 498) and the DB_MPOOL_DIRTY, DB_MPOOL_CREATE or DB_MPOOL_NEW flags were specified. Otherwise it is ignored.

**Errors**

The DbMpoolFile::get() method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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

DbEnv, DbMpoolFile

See Also

Memory Pools and Related Methods (page 457)
DbMpoolFile::open()

```cpp
#include <db_cxx.h>

int DbMpoolFile::open(const char *file, u_int32_t flags, int mode,
                      size_t pagesize);
```

The DbMpoolFile::open() method opens a file in the in-memory cache.

The DbMpoolFile::open() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbMpoolFile::get()` (page 495) 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 `DbEnv::set_mp_mmapsize()` (page 487) 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 DbMpoolFile::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 DbMpoolFile::open() are created with mode `mode`, unmodified by the process' umask value. If `mode` is 0, DbMpoolFile::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 DbMpoolFile::open() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbMpoolFile::put()

```cpp
#include <db_cxx.h>

int DbMpoolFile::put(void *pgaddr, DB_CACHE_PRIORITY priority,
                      u_int32_t flags);
```

The DbMpoolFile::put() method returns a reference to a page in the cache, setting the priority of the page as specified by the `priority` parameter.

The DbMpoolFile::put() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbMpoolFile::get() (page 495) method.

**priority**

Set the page's priority as follows:

- **DB_PRIORITY_UNCHANGED**
  
  The priority is unchanged.

- **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 DbMpoolFile::put() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv, DbMpoolFile

See Also

Memory Pools and Related Methods (page 457)
DbMpoolFile::sync()

#include <db_cxx.h>

int DbMpoolFile::sync();

The DbMpoolFile::sync() method writes all modified pages associated with the DbMpoolFile back to the source file. If any of the modified pages are pinned (that is, currently in use), DbMpoolFile::sync() will ignore them.

The DbMpoolFile::sync() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Class

DbEnv, DbMpoolFile

See Also

Memory Pools and Related Methods (page 457)
DbMpoolFile::get_clear_len()

```cpp
#include <db_cxx.h>

int DbMpoolFile::get_clear_len(u_int32_t *lenp);
```

The `DbMpoolFile::get_clear_len()` method returns the bytes to be cleared.

The `DbMpoolFile::get_clear_len()` method may be called at any time during the life of the application.

The `DbMpoolFile::get_clear_len()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `lenp`

  The `DbMpoolFile::get_clear_len()` method returns the bytes to be cleared in `lenp`.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`
DbMpoolFile::get_fileid()

```cpp
#include <db_cxx.h>
int DbMpoolFile::get_fileid(u_int8_t *fileid);
```

The DbMpoolFile::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 DbMpoolFile::get_fileid() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbMpoolFile::set_fileid() (page 513)
DbMpoolFile::get_flags()

```cpp
#include <db_cxx.h>

int DbMpoolFile::get_flags(u_int32_t *flagsp);
```

The DbMpoolFile::get_flags() method returns the flags used to configure a file in the cache.

The DbMpoolFile::get_flags() method may be called at any time during the life of the application.

The DbMpoolFile::get_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **flagsp**

  The DbMpoolFile::get_flags() method returns the flags in `flagsp`.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbMpoolFile::set_flags() (page 515)
DbMpoolFile::get_ftype()

```c
#include <db_cxx.h>

int
DbMpoolFile::get_ftype(int *ftypep);
```

The DbMpoolFile::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 DbEnv::memp_register() (page 469) method.

The DbMpoolFile::get_ftype() method may be called at any time during the life of the application.

The DbMpoolFile::get_ftype() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*ftypep*  
The DbMpoolFile::get_ftype() method returns the file type in *ftypep*.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbMpoolFile::set_ftype() (page 517)
DbMpoolFile::get_lsn_offset()

```cpp
#include <db_cxx.h>

int DbMpoolFile::get_lsn_offset(int32_t *lsn_offsetp);
```

The `DbMpoolFile::get_lsn_offset()` method returns the log sequence number byte offset configured for a file's pages using the `DbMpoolFile::set_lsn_offset()` method.

The `DbMpoolFile::get_lsn_offset()` method may be called at any time during the life of the application.

The `DbMpoolFile::get_lsn_offset()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*lsn_offsetp*

The `DbMpoolFile::get_lsn_offset()` method returns the log sequence number byte offset in `lsn_offsetp`.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`, `DbMpoolFile::set_lsn_offset()` (page 518)
DbMpoolFile::get_maxsize()

```
#include <db_cxx.h>

int DbMpoolFile::get_maxsize(u_int32_t *gbytesp, u_int32_t *bytesp);
```

Returns the maximum size configured for the file, as configured using the 
DbMpoolFile::set_maxsize() (page 519) method.

The DbMpoolFile::get_maxsize() method either returns a non-zero error value or throws an 
exception that encapsulates a non-zero error value on failure, and returns 0 on success.

The DbMpoolFile::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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457), DbMpoolFile::set_maxsize() (page 519)
DbMpoolFile::get_pgcookie()

```cpp
#include <db_cxx.h>

int
DbMpoolFile::get_pgcookie(DBT *dbt);
```

The `DbMpoolFile::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 `DbMpoolFile::set_pgcookie()` method. (page 520)

The `DbMpoolFile::get_pgcookie()` method may be called at any time during the life of the application.

The `DbMpoolFile::get_pgcookie()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

* dbt

The `DbMpoolFile::get_pgcookie()` method returns a reference to the byte string in `dbt`.

**Class**

`DbEnv, DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457), DbMpoolFile::set_pgcookie()` (page 520)
DbMpoolFile::get_priority()

```cpp
#include <db_cxx.h>

int
DbMpoolFile::get_priority(DB_CACHE_PRIORITY *priorityp);
```

The `DbMpoolFile::get_priority()` method returns the cache priority for the file referenced by the `DbMpoolFile` 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 `DbMpoolFile::set_priority()` (page 521) method.

The `DbMpoolFile::get_priority()` method may be called at any time during the life of the application.

The `DbMpoolFile::get_priority()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`priorityp`

The `DbMpoolFile::get_priority()` method returns a reference to the cache priority for the file referenced by the `DbMpoolFile` handle in `priorityp`.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`, `DbMpoolFile::set_priority()` (page 521)
DbEnv::get_region_dir()

```
#include <db_cxx.h>

int DbEnv::get_region_dir(const char **dirp);
```

The DbEnv::get_region_dir() method returns the region directory, which is the location for the disk-backed region files. You can manage this value using the DbEnv::set_region_dir() (page 492) method.

The DbEnv::get_region_dir() method may be called at any time during the life of the application.

The DbEnv::get_region_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **dirp**

  The DbEnv::get_region_dir() method returns a reference to the region directory in `dirp`.

**Class**

DbEnv, DbMpoolFile

**See Also**

*Memory Pools and Related Methods (page 457), DbEnv::set_region_dir() (page 492)*
DbMpoolFile::set_clear_len()

```cpp
#include <db_cxx.h>

int DbMpoolFile::set_clear_len(u_int32_t len);
```

The `DbMpoolFile::set_clear_len()` method sets the number of initial bytes in a page that should be set to null when the page is created as a result of the `DB_MPOOL_CREATE` or `DB_MPOOL_NEW` flags specified to `DbMpoolFile::get()` (page 495). If no clear length is specified, the entire page is cleared when it is created.

The `DbMpoolFile::set_clear_len()` method configures a file in the cache, not only operations performed using the specified `DbMpoolFile` handle.

The `DbMpoolFile::set_clear_len()` method may not be called after the `DbMpoolFile::open()` (page 498) method is called. If the file is already open in the cache when `DbMpoolFile::open()` (page 498) is called, the information specified to `DbMpoolFile::set_clear_len()` must be consistent with the existing file or an error will be returned.

The `DbMpoolFile::set_clear_len()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

_len_

The `len` parameter is the number of initial bytes in a page that should be set to null when the page is created. A value of 0 results in the entire page being set to null bytes.

**Class**

`DbEnv`, `DbMpoolFile`

**See Also**

`Memory Pools and Related Methods (page 457)`
DbMpoolFile::set_fileid()

```c
#include <db_cxx.h>

int DbMpoolFile::set_fileid(u_int8_t *fileid);
```

The DbMpoolFile::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 DbMpoolFile::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 DbMpoolFile::set_fileid() method configures a file in the cache, not only operations performed using the specified DbMpoolFile handle.

The DbMpoolFile::set_fileid() method may not be called after the DbMpoolFile::open() (page 498) method is called. If the mpool file already exists when DbMpoolFile::open() is called, the information specified to DbMpoolFile::set_fileid() must be the same as that historically used to create the mpool file or corruption can occur.

The DbMpoolFile::set_fileid() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv, DbMpoolFile
See Also

Memory Pools and Related Methods (page 457)
DbMpoolFile::set_flags()

```c
#include <db_cxx.h>

int DbMpoolFile::set_flags(u_int32_t flags, bool onoff);
```

Configure a file in the cache.

To set the flags for a particular database, call the DbMpoolFile::set_flags() method using the DbMpoolFile handle stored in the mpf field of the Db handle.

The DbMpoolFile::set_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**flags**

The flags must be set to one 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 DbMpoolFile 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_UNLINK flag configures a file in the cache, not only operations performed using the specified DbMpoolFile handle.

  The DB_MPOOL_UNLINK 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**

DbEnv, DbMpoolFile
See Also

Memory Pools and Related Methods (page 457)
DbMpoolFile::set_ftype()

```cpp
#include <db_cxx.h>

int DbMpoolFile::set_flags(int ftype);
```

The `DbMpoolFile::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 `DbMpoolFile::set_ftype()` method configures a file in the cache, not only operations performed using the specified `DbMpoolFile` handle.

The `DbMpoolFile::set_ftype()` method may not be called after the `DbMpoolFile::open()` (page 498) method is called. If the file is already open in the cache when `DbMpoolFile::open()` (page 498) is called, the information specified to `DbMpoolFile::set_ftype()` will replace the existing information.

The `DbMpoolFile::set_ftype()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::memp_register()` (page 469) method.

**Class**

- `DbEnv`, `DbMpoolFile`

**See Also**

- *Memory Pools and Related Methods* (page 457)
DbMpoolFile::set_lsn_offset()

```cpp
#include <db_cxx.h>

int DbMpoolFile::set_lsn_offset(int32_t lsn_offset);
```

The DbMpoolFile::set_lsn_offset() method specifies the zero-based byte offset of a log sequence number (DbLsn) on the file's pages, for the purposes of page-flushing as part of transaction checkpoint. (See the DbEnv::memp_sync() (page 478) documentation for more information.)

The DbMpoolFile::set_lsn_offset() method configures a file in the cache, not only operations performed using the specified DbMpoolFile handle.

The DbMpoolFile::set_lsn_offset() method may not be called after the DbMpoolFile::open() (page 498) method is called. If the file is already open in the cache when DbMpoolFile::open() (page 498) is called, the information specified to DbMpoolFile::set_lsn_offset() must be consistent with the existing file or an error will be returned.

The DbMpoolFile::set_lsn_offset() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbMpoolFile::set_maxsize()

```cpp
#include <db_cxx.h>

int
DbMpoolFile::set_maxsize(u_int32_t gbytes, u_int32_t bytes);
```

Set the maximum size for the file to be `gbytes` gigabytes plus `bytes`. Attempts to set the file size smaller than or equal to the page size removes the file size limit. 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 `DbMpoolFile::set_maxsize()` method using the `DbMpoolFile` 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 `DbMpoolFile::set_maxsize()` method configures a file in the cache, not only operations performed using the specified `DbMpoolFile` handle.

The `DbMpoolFile::set_maxsize()` method may be called at any time during the life of the application.

The `DbMpoolFile::set_maxsize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

- `DbEnv`, `DbMpoolFile`

**See Also**

- *Memory Pools and Related Methods (page 457)*
DbMpoolFile::set_pgcookie()

```cpp
#include <db_cxx.h>

int DbMpoolFile::set_pgcookie(DBT *pgcookie);
```

The DbMpoolFile::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 DbEnv::memp_register() (page 469) documentation for more information.)

The DbMpoolFile::set_pgcookie() method configures a file in the cache, not only operations performed using the specified DbMpoolFile handle.

The DbMpoolFile::set_pgcookie() method may not be called after the DbMpoolFile::open() (page 498) method is called. If the file is already open in the cache when DbMpoolFile::open() (page 498) is called, the information specified to DbMpoolFile::set_pgcookie() will replace the existing information.

The DbMpoolFile::set_pgcookie() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**pgcookie**

The pgcokie parameter is a byte string provided to the functions registered to do input or output processing of the file's pages.

**Class**

DbEnv, DbMpoolFile

**See Also**

Memory Pools and Related Methods (page 457)
DbMpoolFile::set_priority()

```
#include <db_cxx.h>

int DbMpoolFile::set_priority(DB_CACHE_PRIORITY priority);
```

Set the cache priority for pages referenced by the DbMpoolFile 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 DbMpoolFile::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 DbMpoolFile::set_priority() method using the DbMpoolFile handle returned by the Db::get_mpf() (page 459) method.

The DbMpoolFile::set_priority() method configures a file in the cache, not only operations performed using the specified DbMpoolFile handle.

The DbMpoolFile::set_priority() method may be called at any time during the life of the application.

The DbMpoolFile::set_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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

DbEnv, DbMpoolFile

See Also

Memory Pools and Related Methods (page 457)
Chapter 10. 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

<table>
<thead>
<tr>
<th>Mutexes and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::mutex_alloc()</code></td>
<td>Allocate a mutex</td>
</tr>
<tr>
<td><code>DbEnv::mutex_free()</code></td>
<td>Free a mutex</td>
</tr>
<tr>
<td><code>DbEnv::mutex_lock()</code></td>
<td>Lock a mutex</td>
</tr>
<tr>
<td><code>DbEnv::mutex_stat()</code></td>
<td>Mutex statistics</td>
</tr>
<tr>
<td><code>DbEnv::mutex_stat_print()</code></td>
<td>Print mutex statistics</td>
</tr>
<tr>
<td><code>DbEnv::mutex_unlock()</code></td>
<td>Unlock a mutex</td>
</tr>
</tbody>
</table>

### Mutex Configuration

<table>
<thead>
<tr>
<th>Mutex Configuration</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::mutex_set_align()</code></td>
<td>Configure mutex alignment</td>
</tr>
<tr>
<td><code>DbEnv::mutex_get_align()</code></td>
<td></td>
</tr>
<tr>
<td><code>DbEnv::mutex_set_increment()</code></td>
<td>Configure number of additional mutexes</td>
</tr>
<tr>
<td><code>DbEnv::mutex_get_increment()</code></td>
<td></td>
</tr>
<tr>
<td><code>DbEnv::mutex_set_init()</code></td>
<td>Configure initial number of mutexes</td>
</tr>
<tr>
<td><code>DbEnv::mutex_get_init()</code></td>
<td></td>
</tr>
<tr>
<td><code>DbEnv::mutex_set_max()</code></td>
<td>Configure total number of mutexes</td>
</tr>
<tr>
<td><code>DbEnv::mutex_get_max()</code></td>
<td></td>
</tr>
<tr>
<td><code>DbEnv::mutex_set_tas_spins()</code></td>
<td>Configure test-and-set mutex spin count</td>
</tr>
<tr>
<td><code>DbEnv::mutex_get_tas_spins()</code></td>
<td></td>
</tr>
</tbody>
</table>
DbEnv::mutex_alloc()

```c
#include <db_cxx.h>

int DbEnv::mutex_alloc(u_int32_t flags, db_mutex_t *mutexp);
```

The `DbEnv::mutex_alloc()` method allocates a mutex and returns a reference to it into the memory specified by `mutexp`.

The `DbEnv::mutex_alloc()` method may not be called before the `DbEnv::open()` method is called.

The `DbEnv::mutex_alloc()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::failchk()` 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 `DbEnv::mutex_alloc()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  An invalid flag value or parameter was specified.

**Class**

`DbEnv`
See Also

 Mutex Methods (page 524)
DbEnv::mutex_free()

#include <db_cxx.h>

int DbEnv::mutex_free(db_mutex_t mutex);

The DbEnv::mutex_free() method discards a mutex allocated by DbEnv::mutex_alloc() (page 525).

The DbEnv::mutex_free() method may not be called before the DbEnv::open() (page 286) method is called.

The DbEnv::mutex_free() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

mutex

The mutex parameter is a mutex previously allocated by DbEnv::mutex_alloc() (page 525).

Errors

The DbEnv::mutex_free() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

An invalid flag value or parameter was specified.

Class

DbEnv

See Also

Mutex Methods (page 524)
DbEnv::mutex_get_align()

```c
#include <db_cxx.h>

int
DbEnv::mutex_get_align(u_int32_t *alignp);
```

The `DbEnv::mutex_get_align()` method returns the mutex alignment, in bytes.

The `DbEnv::mutex_get_align()` method may be called at any time during the life of the application.

The `DbEnv::mutex_get_align()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

alignp

The `DbEnv::mutex_get_align()` method returns the mutex alignment, in bytes in `alignp`.

**Class**

`DbEnv`

**See Also**

`Mutex Methods (page 524)`
DbEnv::mutex_get_increment()

```c
#include <db_cxx.h>

int
DbEnv::mutex_get_increment(u_int32_t *incrementp);
```

The `DbEnv::mutex_get_increment()` method returns the number of additional mutexes to allocate.

The `DbEnv::mutex_get_increment()` method may be called at any time during the life of the application.

The `DbEnv::mutex_get_increment()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**incrementp**

The `DbEnv::mutex_get_increment()` method returns the number of additional mutexes to allocate in `incrementp`.

**Class**

`DbEnv`

**See Also**

`Mutex Methods (page 524)`
DbEnv::mutex_get_init()

```cpp
#include <db_cxx.h>

int DbEnv::mutex_get_init(u_int32_t *init);
```

The `DbEnv::mutex_get_init()` method returns the initial number of mutexes allocated. This value can be set using the `DbEnv::mutex_set_init()` (page 538) method.

The `DbEnv::mutex_get_init()` method may be called at any time during the life of the application.

The `DbEnv::mutex_get_init()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `init`:

The `DbEnv::mutex_get_init()` method returns the initial number of mutexes allocated in `init`.

**Class**

`DbEnv`

**See Also**

`Mutex Methods` (page 524)
**DbEnv::mutex_get_max()**

```cpp
#include <db_cxx.h>

int DbEnv::mutex_get_max(u_int32_t *maxp);
```

The `DbEnv::mutex_get_max()` method returns the total number of mutexes allocated. This method is deprecated.

The `DbEnv::mutex_get_max()` method may be called at any time during the life of the application.

The `DbEnv::mutex_get_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`maxp`

The `DbEnv::mutex_get_max()` method returns the total number of mutexes allocated in `maxp`.

**Class**

`DbEnv`

**See Also**

`Mutex Methods (page 524)`
DbEnv::mutex_get_tas_spins()

```cpp
#include <db_cxx.h>

int DbEnv::mutex_get_tas_spins(u_int32_t *, tas_spinsp);
```

The `DbEnv::mutex_get_tas_spins()` method returns the test-and-set spin count. This value may be configured using the `DbEnv::mutex_set_tas_spins()` method.

The `DbEnv::mutex_get_tas_spins()` method may be called at any time during the life of the application.

The `DbEnv::mutex_get_tas_spins()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**tas_spinsp**

The `DbEnv::mutex_get_tas_spins()` method returns the test-and-set spin count in `tas_spinsp`.

**Class**

`DbEnv`

**See Also**

`Mutex Methods (page 524)`
DbEnv::mutex_lock()

```cpp
#include <db_cxx.h>

int DbEnv::mutex_lock(db_mutex_t mutex);
```

The DbEnv::mutex_lock() method locks the mutex allocated by DbEnv::mutex_alloc() (page 525). The thread of control calling DbEnv::mutex_lock() will block until the lock is available.

The DbEnv::mutex_lock() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **mutex**

  The mutex parameter is a mutex previously allocated by DbEnv::mutex_alloc() (page 525).

**Errors**

The DbEnv::mutex_lock() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  An invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Mutex Methods (page 524)
DbEnv::mutex_set_align()

```cpp
#include <db_cxx.h>

int DbEnv::mutex_set_align(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 DbEnv::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 DbEnv::mutex_set_align() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::mutex_set_align() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::mutex_set_align() will be ignored.

The DbEnv::mutex_set_align() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::mutex_set_align() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv
See Also

Mutex Methods (page 524)
DbEnv::mutex_set_increment()

```c++
#include <db_cxx.h>

int
DbEnv::mutex_set_increment(u_int32_t increment);
```

Configure the number of additional mutexes to allocate.

If an application will allocate mutexes for its own use, the DbEnv::mutex_set_increment() method is used to add a number of mutexes to the default allocation.

Calling the DbEnv::mutex_set_increment() method discards any value previously set using the DbEnv::mutex_set_max() (page 539) 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 DbEnv::mutex_set_increment() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::mutex_set_increment() method may not be called after the DbEnv::open() (page 286) method is called. If the database environment already exists when DbEnv::open() (page 286) is called, the information specified to DbEnv::mutex_set_increment() will be ignored.

The DbEnv::mutex_set_increment() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

increment

The increment parameter is the number of additional mutexes to allocate.

**Errors**

The DbEnv::mutex_set_increment() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called after DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv
See Also

Mutex Methods (page 524)
DbEnv::mutex_set_init()

```c
#include <db_cxx.h>

int
DbEnv::mutex_set_init(u_int32_t init);
```

Configure the initial number of mutexes to allocate.

Berkeley DB allocates a default number of mutexes based on the initial configuration of the database environment. The `DbEnv::mutex_set_init()` method is used to override this default number of mutexes to allocate. This may be done to either speed up startup, or to force more work to be done at startup to avoid later contention due to allocation.

The database environment's initial 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_init", one or more whitespace characters, and the initial 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 `DbEnv::mutex_set_init()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::mutex_set_init()` method may not be called after the `DbEnv::open()` method is called. If the database environment already exists when `DbEnv::open()` is called, the information specified to `DbEnv::mutex_set_init()` will be ignored.

The `DbEnv::mutex_set_init()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*init*

The `init` parameter is the absolute number of mutexes to allocate.

**Errors**

The `DbEnv::mutex_set_init()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

Mutex Methods (page 524)
DbEnv::mutex_set_max()

```c
#include <db_cxx.h>

int DbEnv::mutex_set_max(u_int32_t max);
```

Configure the total number of mutexes to allocate. This method is deprecated. The maximum size of the mutex region is now inferred by the sizes of the other memory structures, and so this method is no longer needed. For example, the cache requires one mutex per page in the cache. When you specify the cache size, DB assumes a page size of 4K and allocates mutexes accordingly. If your page size is different than 4K, you indicate this using `DbEnv::set_mp_pagesize()` (page 490). DB will then allocate the proper number of mutexes based on this new page size.

You can use this method to override DB's mutex calculation, but it is not recommended to do so.

Calling the `DbEnv::mutex_set_max()` method discards any value previously set using the `DbEnv::mutex_set_increment()` (page 536) 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 `DbEnv::mutex_set_max()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::mutex_set_max()` method may not be called after the `DbEnv::open()` (page 286) method is called. If the database environment already exists when `DbEnv::open()` (page 286) is called, the information specified to `DbEnv::mutex_set_max()` will be ignored.

The `DbEnv::mutex_set_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`max`

The `max` parameter is the absolute number of mutexes to allocate.

**Errors**

The `DbEnv::mutex_set_max()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.
Class

DbEnv

See Also

Mutex Methods (page 524)
DbEnv::mutex_set_tas_spins()

```c
#include <db_cxx.h>

int
DbEnv::mutex_set_tas_spins(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, up to a maximum of 200.

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 `DbEnv::mutex_set_tas_spins()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::mutex_set_tas_spins()` method may be called at any time during the life of the application.

The `DbEnv::mutex_set_tas_spins()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::mutex_set_tas_spins()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  An invalid flag value or parameter was specified.

### Class

- **DbEnv**

### See Also

- Mutex Methods (page 524)
DbEnv::mutex_stat()

```c
#include <db_cxx.h>

int
DbEnv::mutex_stat(DB_MUTEX_STAT **statp, u_int32_t flags);
```

The DbEnv::mutex_stat() method returns the mutex subsystem statistics. The DbEnv::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 DbEnv::set_alloc() (page 294) 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.

- **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_init;**
  The initial number of mutexes configured.

- **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.

- **u_int32_t st_mutex_max;**
  The maximum number of mutexes.

- **u_int32_t st_mutex_tas_spins;**
  The number of times test-and-set mutexes will spin without blocking.

- **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.

- `roff_t st_regmax;`
  The max size of the mutex region size.

- `roff_t st_regsize;`
  The size of the mutex region, in bytes.

The `DbEnv::mutex_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::mutex_stat()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::mutex_stat()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  An invalid flag value or parameter was specified.

**Class**

`DbEnv`
See Also

Mutex Methods (page 524)
DbEnv::mutex_stat_print()

```cpp
#include <db_cxx.h>

int DbEnv::mutex_stat_print(u_int32_t flags);
```

The `DbEnv::mutex_stat_print()` method displays the mutex subsystem statistical information, as described for the `DbEnv::mutex_stat()` method. The information is printed to a specified output channel (see the `DbEnv::set_msgfile()` (page 342) method for more information), or passed to an application callback function (see the `DbEnv::set_msgcall()` (page 340) method for more information).

The `DbEnv::mutex_stat_print()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::mutex_stat_print()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_ALLOC**
  
  Display allocation information. To display allocation information, both `DB_STAT_ALLOC` and `DB_STAT_ALL` need to be set.

- **DB_STAT_CLEAR**
  
  Reset statistics after displaying their values.

**Class**

`DbEnv`

**See Also**

`Mutex Methods (page 524)`
DbEnv::mutex_unlock()

```c
#include <db_cxx.h>

int
DbEnv::mutex_unlock(db_mutex_t mutex);
```

The DbEnv::mutex_unlock() method unlocks the mutex locked by DbEnv::mutex_lock() (page 533).

The DbEnv::mutex_unlock() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**mutex**

The `mutex` parameter is a mutex previously locked by DbEnv::mutex_lock() (page 533).

**Errors**

The DbEnv::mutex_unlock() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Mutex Methods (page 524)
Chapter 11. 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.

Finally, Replication Manager applications use the DbSite class to manage and configure replication sites. The DbChannel class can be used to transmit custom messages between sites in the replication group. These classes are not used in any way by Base API applications.
## Replication and Related Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbChannel::close()</code></td>
<td>Closes a DB_CHANNEL handle</td>
</tr>
<tr>
<td><code>DbChannel::send_msg()</code></td>
<td>Sends an asynchronous message on a DB_CHANNEL</td>
</tr>
<tr>
<td><code>DbChannel::send_request()</code></td>
<td>Sends a synchronous message on a DB_CHANNEL</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_channel()</code></td>
<td>Creates a DB_CHANNEL handle</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_local_site()</code></td>
<td>Returns a DB_SITE handle for the local site</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_msg_dispatch()</code></td>
<td>Configures a DB_CHANNEL's message dispatch function</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_set_ack_policy()</code>, <code>DbEnv::repmgr_get_ack_policy()</code></td>
<td>Specify the Replication Manager's client acknowledgement policy</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_set_incoming_queue_max()</code>, <code>DbEnv::repmgr_get_incoming_queue_max()</code></td>
<td>Configure the Replication Manager incoming queue size limit.</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_set_socket()</code></td>
<td>Configure the Replication Manager's socket callback</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_site()</code></td>
<td>Creates a DB_SITE handle</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_site_by_eid()</code></td>
<td>Creates a DB_SITE handle given an EID value</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_site_list()</code></td>
<td>List the sites and their status</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_start()</code></td>
<td>Start the Replication Manager</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_stat()</code></td>
<td>Replication Manager statistics</td>
</tr>
<tr>
<td><code>DbEnv::repmgr_stat_print()</code></td>
<td>Print Replication Manager statistics</td>
</tr>
<tr>
<td><code>DbSite::close()</code></td>
<td>Closes the DB_SITE handle</td>
</tr>
</tbody>
</table>

### Base API Methods

- `DbEnv::rep_elect()` - Hold a replication election
- `DbEnv::rep_process_message()` - Process a replication message
- `DbEnv::rep_set_transport()` - Configure replication transport callback
- `DbEnv::rep_start()` - Start replication

### Additional Replication Methods

- `DbEnv::rep_stat()` - Replication statistics
- `DbEnv::rep_stat_print()` - Print replication statistics
- `DbEnv::rep_sync()` - Replication synchronization

### Replication Configuration

- `DbChannel::set_timeout()` - Sets the default timeout for a DB_CHANNEL
- `DbSite::get_address()` - Returns a site's network address
- `DbSite::get_eid()` - Returns a site's environment ID
<table>
<thead>
<tr>
<th>Replication Manager Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbSite::remove()</td>
<td>Removes the site from the replication group</td>
</tr>
<tr>
<td>DbSite::set_config(), DbSite::get_config()</td>
<td>Configure a DB_SITE handle</td>
</tr>
<tr>
<td>DbEnv::rep_set_clockskew(), DbEnv::rep_get_clockskew()</td>
<td>Configure master lease clock adjustment</td>
</tr>
<tr>
<td>DbEnv::rep_set_config(), DbEnv::rep_get_config()</td>
<td>Configure the replication subsystem</td>
</tr>
<tr>
<td>DbEnv::rep_set_limit(), DbEnv::rep_get_limit()</td>
<td>Limit data sent in response to a single message</td>
</tr>
<tr>
<td>DbEnv::rep_set_nsites(), DbEnv::rep_get_nsites()</td>
<td>Configure replication group site count</td>
</tr>
<tr>
<td>DbEnv::rep_set_priority(), DbEnv::rep_get_priority()</td>
<td>Configure replication site priority</td>
</tr>
<tr>
<td>DbEnv::rep_set_request(), DbEnv::rep_get_request()</td>
<td>Configure replication client retransmission requests</td>
</tr>
<tr>
<td>DbEnv::rep_set_timeout(), DbEnv::rep_get_timeout()</td>
<td>Configure replication timeouts</td>
</tr>
<tr>
<td>DbEnv::rep_set_view()</td>
<td>Configure the replication view callback</td>
</tr>
<tr>
<td><strong>Transaction Operations</strong></td>
<td></td>
</tr>
<tr>
<td>DbEnv::txn_applied()</td>
<td>Check if a transaction has been replicated</td>
</tr>
<tr>
<td>DbTxn::set_commit_token()</td>
<td>Set a commit token</td>
</tr>
</tbody>
</table>
The DbSite Handle

The DbSite handle is used by Replication Manager applications to manage and configure replication sites. You create a DbSite handle using the `DbEnv::repmgr_site()` (page 624), `DbEnv::repmgr_site_by_eid()` (page 626), or `DbEnv::repmgr_local_site()` (page 613), methods. All DbSite handles must be closed before closing DbEnv handles. Use the `DbSite::close()` (page 636) method to close a DbSite handle.
The DbChannel Handle

The DbChannel handle is used by Replication Manager applications to manage and configure message channels that carry custom message traffic between the sites in the replication group. You create a DbChannel handle using the `DbEnv::repmgr_channel()` (page 611) method. All DbChannel handles must be closed before closing DbEnv handles. Use the `DbChannel::close()` (page 552) method to close a DbChannel handle.
DbChannel::close()

```c
#include <db_cxx.h>

int DbChannel::close(u_int32_t flags);
```

The DbChannel::close() method closes the DbChannel handle, freeing any resources allocated to the handle. All DbChannel handles must be closed before the encompassing environment handle is closed. Also, all on-going messaging operations on the channel should be allowed to complete before attempting to close the channel handle.

After DbChannel::close() has been called, regardless of its return, the DbChannel handle may not be accessed again.

The DbChannel::close() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**flags**

This parameter is currently unused, and must be set to 0.

**Errors**

The DbChannel::close() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

**Class**

DbEnv, DbChannel

**See Also**

Replication and Related Methods (page 548)
DbChannel::send_msg()

```cpp
#include <db_cxx.h>

int DbChannel::send_msg(Dbt *msg, u_int32_t nmsg, u_int32_t flags);
```

The DbChannel::send_msg() method sends a message on the message channel. The message is sent asynchronously; the method does not wait for a response before returning. This method usually completes quickly because it only waits for the local TCP implementation to accept the bytes into its network data buffer. However, this message could block briefly for longer messages, and/or if the network data buffer is nearly full. This method could even block indefinitely if the remote site is slow to read.

If you want to block while waiting for a response from a remote site, use the DbChannel::send_request() (page 555) method instead of this method.

The message sent by this method is received and handled at remote sites using a message dispatch callback, which is configured using the DbEnv::repmgr_msg_dispatch() (page 616) method. Note that the DbChannel::send_msg() method may be used within the message dispatch callback on the remote site to send a response or acknowledgement for messages that it receives and is handling.

This method may be used on channels opened to any destination (see the DbEnv::repmgr_channel() (page 611) method for a list of potential destinations).

The DbChannel::send_msg() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**msg**

Refers to an array of Dbt handles. For more information, see The Dbt Handle (page 207).

Any flags provided to the Dbt handles used in this array are ignored.

**nmsg**

Indicates how many elements are contained in the msg array.

**flags**

This parameter is currently unused, and must be set to 0.

**Errors**

The DbChannel::send_msg() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
DB_NOSERVER

A message was sent to a remote site that has not configured a message dispatch callback function. Use the DbEnv::repmgr_msg_dispatch() (page 616) method at every site belonging to the replication group to configure a message dispatch callback function.

EINVAL

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

Class

DbEnv, DbChannel

See Also

Replication and Related Methods (page 548)
DbChannel::send_request()

```cpp
#include <db_cxx.h>

int
DbChannel::send_request(Dbt *request, u_int32_t nrequest,
                          Dbt *response, db_timeout_t timeout,
                          u_int32_t flags);
```

The `DbChannel::send_request()` method sends a message on the message channel. The message is sent synchronously; the method blocks waiting for a response before returning. If a response is not received within the timeout value configured for this request, this method returns with an error condition.

If you do not want to block while waiting for a response from a remote site, use the `DbChannel::send_msg()` (page 553) method.

The message sent by this method is received and handled at remote sites using a message dispatch callback, which is configured using the `DbEnv::repmgr_msg_dispatch()` (page 616) method.

The `DbChannel::send_request()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**request**

Refers to an array of Dbt handles. For more information, see [The Dbt Handle (page 207)](#).

Any flags provided to the Dbt handles used in this array are ignored.

**nrequest**

Indicates how many elements are contained in the `msg` array.

**response**

Points to a single Dbt handle, which is used to receive the response from the remote site. By default, the response is expected to be a single-part message. If there is a possibility that the response could be a multi-part message, specify DB_MULTIPLE to this method's `flags` parameter.

The response Dbt should specify one of the following flags: DB_DBT_MALLOC, DB_DBT_REALLOC, or DB_DBT_USERMEM.

For more information on configuring and using Dbts, see [The Dbt Handle (page 207)](#).

Note that the response Dbt can be empty. In this way an application can send an acknowledgement even if there is no other information that needs to be sent.
timeout

Configures the amount of time that may elapse while this method waits for a response from the remote site. If this timeout period elapses without a response, this method returns with an error condition.

The timeout value must be specified as an unsigned 32-bit number of microseconds, limiting the maximum timeout to roughly 71 minutes.

A timeout value of 0 indicates that the channel's default timeout value should be used. This default is configured using the `DbChannel::set_timeout()` (page 557) method.

flags

This parameter must be set to either DB_MULTIPLE or 0.

If there is a possibility that the response can consist of multiple Dbt handles, specify DB_MULTIPLE to this parameter. In that case, the response buffer is formatted for bulk operations.

Errors

The DbChannel::send_request() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DB_BUFFER_SMALL**

DB_MULTIPLE was not specified for the response Dbt, but the remote site sent a response consisting of more than one Dbt; or a buffer supplied using DB_DBT_USERMEM was not large enough to contain the message response.

**DB_NOSERVER**

A message was sent to a remote site that has not configured a message dispatch callback function. Use the `DbEnv::repmgr_msg_dispatch()` (page 616) method at every site belonging to the replication group to configure a message dispatch callback function.

**EINVAL**

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

Class

DbEnv, DbChannel

See Also

Replication and Related Methods (page 548)
DbChannel::set_timeout()

```c
#include <db_cxx.h>

int
DbChannel::set_timeout(db_timeout_t timeout);
```

The `DbChannel::set_timeout()` method sets the default timeout value for the DbChannel handle. This timeout is used by the `DbChannel::send_request()` (page 555) method.

The `DbChannel::set_timeout()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**timeout**

Configures the amount of time that may elapse while the `DbChannel::send_request()` (page 555) method waits for a message response. The timeout value must be specified as an unsigned 32-bit number of microseconds, limiting the maximum timeout to roughly 71 minutes.

**Errors**

The `DbChannel::set_timeout()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

**Class**

DbEnv, DbChannel

**See Also**

Replication and Related Methods (page 548)
DbSite::get_config()

```cpp
#include <db_cxx.h>

int DbSite::get_config(u_int32_t which, u_int32_t *valuep);
```

The `DbSite::get_config()` method returns whether the specified `which` parameter is currently set. See the `DbSite::set_config()` (page 562) method for the configuration flags that can be set for a DbSite handle.

The `DbSite::get_config()` method may be called at any time during the life of the application.

The `DbSite::get_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **which**
  
The `which` parameter is the configuration flag to check. See the `DbSite::set_config()` (page 562) method for a list of configuration flags that you can provide to this parameter.

- **valuep**
  
The `valuep` parameter references memory into which the configuration of the specified `which` parameter is copied.

  If the returned value is zero, the configuration flag is off; otherwise it is on.

**Class**

- `DbSite`

**See Also**

- Replication and Related Methods (page 548), `DbSite::set_config()` (page 562)
DbSite::get_address()

```c
#include <db_cxx.h>

int DbSite::get_address(const char **hostp, u_int *portp);
```

The DbSite::get_address() method returns a replication site's network address. That is, this method returns the site's host name and port.

The DbSite::get_address() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**hostp**

References memory into which is copied a pointer to the internal storage of the host name.

**portp**

References memory into which the port number will be copied.

**Class**

DbSite

**See Also**

Replication and Related Methods (page 548)
DbSite::get_eid()

```cpp
#include <db_cxx.h>

int DbSite::get_eid(int *eidp);
```

The `DbSite::get_eid()` method returns a replication site's environment ID (EID). This method may not be called before opening the database environment.

The `DbSite::get_eid()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

* eidp
  References memory into which the EID will be copied.

**Errors**

The `DbSite::get_eid()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

DbSite

**See Also**

Replication and Related Methods (page 548)
**DbSite::remove()**

```c++
#include <db_cxx.h>

int DbSite::remove();
```

The `DbSite::remove()` method removes the site from the replication group. If called at the master site, Replication Manager updates the group membership database directly. If called from a client, this method causes a request to be sent to the master to perform the operation. The method then awaits confirmation.

The `DbSite` handle must not be accessed again after this method is called, regardless of the return value. This method may not be called before starting Replication Manager.

The `DbSite::remove()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Errors**

The `DbSite::remove()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DB_REP_UNAVAIL**

The master updated the group membership database but did not receive enough acknowledgements from clients to meet the current acknowledgement policy or there was an attempt to remove the current master site from the replication group.

**EINVAL**

If Replication Manager has not been started.

**Class**

`DbSite`

**See Also**

`Replication and Related Methods (page 548)`
**DbSite::set_config()**

```cpp
#include <db_cxx.h>

int DbSite::set_config(u_int32_t which, u_int32_t value);
```

The `DbSite::set_config()` method configures a Replication Manager site.

The `DbSite::set_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

The Replication Manager site may also be configured using the environment's `DB_CONFIG` file. The syntax of the entry in that file is described in `repmgr_site` (page 789).

**Parameters**

**which**

This parameter must be set to one of the following values:

- **DB_BOOTSTRAP_HELPER**

  Specifies that a remote site may be used as a "helper" when the local site is first joining the replication group. Once the local site has been established as a member of the group, this setting is ignored.

- **DB_GROUP_CREATOR**

  Specifies that this site should create the initial group membership database contents, defining a "group" of just the one site, rather than trying to join an existing group when it starts for the first time.

  This setting can only be used on the local site. It is ignored after the local site's initial startup and when configured for a remote site.

- **DB_LEGACY**

  Specifies that the site is already part of an existing group. This setting causes the site to be upgraded from a previous version of Berkeley DB. All sites in the legacy group must specify this setting for themselves (the local site) and for all other sites currently existing in the group. Once the upgrade has been completed, this setting is no longer required.

- **DB_LOCAL_SITE**

  Specifies that this site is the local site within the replication group. The application must identify exactly one site as the local site in this way, before calling the `DbEnv::repmgr_start()` (page 629) method.

- **DB_REPMGR_PEER**
Specifies that the site may be used as a target for "client-to-client" synchronization messages. A peer can be either a client or a view. This setting is ignored if it is specified for the local site.

**value**

If 0, the parameter identified by the *which* is turned off. Otherwise, it is turned on.

**Errors**

The `DbSite::set_config()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If an invalid flag value or parameter was specified.

**Class**

`DbSite`

**See Also**

Replication and Related Methods (page 548)
**DbEnv::rep_elect()**

```c
#include <db_cxx.h>

int
DbEnv::rep_elect(u_int32_t nsites, u_int32_t nvotes, u_int32_t flags);
```

The `DbEnv::rep_elect()` method holds an election for the master of a replication group.

The `DbEnv::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 `DbEnv::set_event_notify()` (page 309) 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 `DbEnv::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 `DbEnv::open()` (page 286) method.
- configure the database environment to send replication messages by calling the `DbEnv::rep_set_transport()` (page 594) method.
- configure the database environment as a client or a master by calling the `DbEnv::rep_start()` (page 599) 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.

Replication view sites never participate in elections. Values chosen for `nsites` and `nvotes` must not include any replication view sites.

We recommend `nvotes` be set to at least:

\[(\text{sites participating in the election} / 2) + 1\]
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 \texttt{nvotes} votes.

We recommend \texttt{nsites} be set to:

\begin{center}
\texttt{number of sites in the replication group - 1}
\end{center}

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 or for the failed master to restart.

When choosing a master from among a group of client sites all restarting at the same time, it makes more sense to set \texttt{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 \texttt{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 \textit{Berkeley DB Programmer's Reference Guide} for more information.)

Setting \texttt{nsites} to lower values can increase the speed of an election, but can also result in election failure, and is usually not recommended.

**Parameters**

\textbf{nsites}

The \texttt{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 \texttt{nsites} parameter must be no less than \texttt{nvotes}, or 0 if the election should use the value previously set using the \texttt{DbEnv::rep_set_nsites()} (page 585) method. If an application is using master leases, then the value must be 0 and the value from \texttt{DbEnv::rep_set_nsites()} (page 585) method must be used. The value should exclude any replication views.

\textbf{nvotes}

The \texttt{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 \texttt{nvotes} parameter must be no greater than \texttt{nsites}, or 0 if the election should use the value \((\texttt{nsites} / 2) + 1\) as the \texttt{nvotes} argument.

\textbf{flags}

The \texttt{flags} parameter is currently unused, and must be set to 0.

**Errors**

The \texttt{DbEnv::rep_elect()} method may fail and throw a \texttt{DbException} exception, encapsulating one of the following non-zero errors, or 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 `DbEnv::rep_set_timeout()` (page 591) method for more information).

**EINVAL**

If the database environment was not already configured to communicate with a replication group by a call to `DbEnv::rep_set_transport()` (page 594); 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**

DbEnv

**See Also**

Replication and Related Methods (page 548)
DbEnv::rep_get_clockskew()

#include <db_cxx.h>

DbEnv::rep_get_clockskew(u_int32_t *fast_clockp, u_int32_t *slow_clockp);

The DbEnv::rep_get_clockskew() method returns the current clock skew ratio values, as set by the DbEnv::rep_set_clockskew() (page 577) method.

The DbEnv::rep_get_clockskew() method may be called at any time during the life of the application.

The DbEnv::rep_get_clockskew() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

fast_clockp

The fast_clockp parameter references memory into which the value for the fastest clock in the replication group is copied.

slow_clockp

The slow_clockp parameter references memory into which the value for the slowest clock in the replication group is copied.

Class

DbEnv

See Also

Replication and Related Methods (page 548), DbEnv::rep_set_clockskew() (page 577)
`DbEnv::rep_get_config()`

```cpp
#include <db_cxx.h>

int DbEnv::rep_get_config(u_int32_t which, int *onoffp);
```

The `DbEnv::rep_get_config()` method returns whether the specified `which` parameter is currently set or not. See the `DbEnv::rep_set_config()` (page 579) method for the configuration flags that can be set for replication.

The `DbEnv::rep_get_config()` method may be called at any time during the life of the application.

The `DbEnv::rep_get_config()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**which**

The `which` parameter is the configuration flag which is being checked. See the `DbEnv::rep_set_config()` (page 579) 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**

`DbEnv`

**See Also**

`Replication and Related Methods (page 548), DbEnv::rep_set_config()` (page 579)
DbEnv::rep_get_limit()

```c
#include <db_cxx.h>

int
DbEnv::rep_get_limit(u_int32_t *gbytesp, u_int32_t *bytesp);
```

The DbEnv::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 DbEnv::rep_process_message() (page 574) method. This value is configurable using the DbEnv::rep_set_limit() (page 583) method.

The DbEnv::rep_get_limit() method may be called at any time during the life of the application.

The DbEnv::rep_get_limit() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv

**See Also**

Replication and Related Methods (page 548), DbEnv::rep_set_limit() (page 583)
DbEnv::rep_get_nsites()

```c
#include <db_cxx.h>

int DbEnv::rep_get_nsites(u_int32_t *nsitesp);
```

The `DbEnv::rep_get_nsites()` method returns the total number of participant sites in the replication group. For Base API applications, this value is configurable using the `DbEnv::rep_set_nsites()` (page 585) method. For Replication Manager applications, this value is determined dynamically.

For Base API applications, this method may be called at any time during the life of the application. For Replication Manager applications, this method may be called only after a successful call to the `DbEnv::repmgr_start()` (page 629) method.

The `DbEnv::rep_get_nsites()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*nsitesp*

The `DbEnv::rep_get_nsites()` method returns the total number of participant sites in the replication group in `nsitesp`.

**Class**

`DbEnv`

**See Also**

`Replication and Related Methods (page 548), DbEnv::rep_set_nsites() (page 585)`
DbEnv::rep_get_priority()

```c
#include <db_cxx.h>

int DbEnv::rep_get_priority(u_int32_t *priorityp);
```

The DbEnv::rep_get_priority() method returns the database environment priority as configured using the DbEnv::rep_set_priority() (page 587) method.

The DbEnv::rep_get_priority() method may be called at any time during the life of the application.

The DbEnv::rep_get_priority() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*priorityp*

The DbEnv::rep_get_priority() method returns the database environment priority in *priorityp*.

**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548), DbEnv::rep_set_priority() (page 587)
DbEnv::rep_get_request()

```c
#include <db_cxx.h>

int
DbEnv::rep_get_request(u_int32_t *minp, u_int32_t *maxp);
```

The DbEnv::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 DbEnv::rep_set_request() (page 589) method.

The DbEnv::rep_get_request() method may be called at any time during the life of the application.

The DbEnv::rep_get_request() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

DbEnv

**See Also**

Replication and Related Methods (page 548), DbEnv::rep_set_request() (page 589)
DbEnv::rep_get_timeout()

```cpp
#include <db_cxx.h>

int DbEnv::rep_get_timeout(int which, u_int32_t *timeoutp);
```

The DbEnv::rep_get_timeout() method returns the timeout value for the specified `which` parameter. Timeout values can be managed using the DbEnv::rep_set_timeout() (page 591) method.

The DbEnv::rep_get_timeout() method may be called at any time during the life of the application.

The DbEnv::rep_get_timeout() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`which`

The `which` parameter is the timeout for which the value is being returned. See the DbEnv::rep_set_timeout() (page 591) 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 DbEnv::rep_get_timeout() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548), DbEnv::rep_set_timeout() (page 591)
DbEnv::rep_process_message()

```cpp
#include <db_cxx.h>

int
DbEnv::rep_process_message(Dbt *control, Dbt *rec, int envid,
    DbLsn *ret_lsnp)
```

The `DbEnv::rep_process_message()` method processes an incoming replication message sent by a member of the replication group to the local database environment.

The `DbEnv::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 `DbEnv` 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 `DbEnv::open()` method and must already have been configured to send replication messages by calling the `DbEnv::rep_set_transport()` method.

The `DbEnv::rep_process_message()` method has additional return values:

- **DB_REP_DUPMASTER**
  
  The `DbEnv::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 `DbEnv::rep_start()` method, and then call for an election by calling `DbEnv::rep_elect()`.

- **DB_REP_HOLDELECTION**
  
  The `DbEnv::rep_process_message()` method will return `DB_REP_HOLDELECTION` if an election is needed. The application should call for an election by calling `DbEnv::rep_elect()`.

- **DB_REP_IGNORE**
  
  The `DbEnv::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 master arrives and is processed late).

- **DB_REP_ISPERM**
  
  The `DbEnv::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 `DbEnv::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. This is possibly because the client has turned off automatic internal initialization by setting the `DB_REP_CONF_AUTOINIT` flag to 0. Another possible reason is that an unencrypted client is attempting to join a replication group where encryption is use, or an encrypted client is attempting to join a replication group where encryption is not in use.

• **DB_REP_NEWSITE**

  The `DbEnv::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 `DbEnv::rep_start()` (page 599) `cdata` parameter. The application should take whatever action is needed to establish a communication channel with this new environment.

• **DB_REP_NOTPERM**

  The `DbEnv::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 `DbEnv::rep_process_message()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::rep_set_transport()` (page 594) 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 `DbEnv::rep_set_transport()` (page 594) 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_lsn**

If `DbEnv::rep_process_message()` method returns `DB_REP_NOTPERM` then the `ret_lsn` parameter will contain the log sequence number of this permanent log message that could not
be written to disk. If DbEnv::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 DbEnv::rep_process_message() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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 DbEnv::rep_set_transport() (page 594); 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

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_set_clockskew()

```cpp
#include <db_cxx.h>

int DbEnv::rep_set_clockskew(u_int32_t fast_clock, u_int32_t slow_clock);
```

The DbEnv::rep_set_clockskew() method sets the clock skew ratio among replication group members based on the fastest and slowest clock measurements among the group for use with master leases. Calling this method is optional; the default values for clock skew assume no skew. The application must also configure leases via the DbEnv::rep_set_config() method and set the master lease timeout via the DbEnv::rep_set_timeout() method. Base API applications must also set the number of sites in the replication group via the DbEnv::rep_set_nsites() 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 DbEnv::rep_set_clockskew() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::rep_set_clockskew() method may not be called after the DbEnv::repmgr_start() or DbEnv::rep_start() methods are called.

The DbEnv::rep_set_clockskew() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::rep_set_clockskew() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called after replication is started with a call to the DbEnv::repmgr_start() (page 629) or the DbEnv::rep_start() (page 599) method; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_set_config()

#include <db_cxx.h>

int
DbEnv::rep_set_config(u_int32_t which, int onoff);

The DbEnv::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 DbEnv::rep_set_config() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::rep_set_config() method may not be called to set in-memory replication after the environment is opened using the DbEnv::open() (page 286) method. This method should not be called to set preferred master mode or master leases after the DbEnv::rep_start() (page 599) or DbEnv::repmgr_start() (page 629) methods are called. For all other which parameters, this method may be called at any time during the life of the application.

The DbEnv::rep_set_config() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::rep_sync() (page 609) 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.

- Replication Manager applications do not maintain group membership information persistently on-disk. For more information, see Managing Replication Files in the *Berkeley DB Programmer's Reference Guide*.

This configuration flag can only be turned on before the environment is opened with the `DbEnv::open()` (page 286) method. Its value cannot be changed while the environment is open. All sites in the replication group should have the same value for this configuration flag.

- **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 31) and `Dbc::get()` (page 193) 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 `DbEnv::repmgr_start()` (page 629) method or the `DbEnv::rep_start()` (page 599) 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 fails or becomes disconnected. This option is turned on by default. In preferred master mode, this option cannot be turned off.
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 replication process may change this configuration setting.

- **DB_REPMGR_CONF_FORWARD_WRITES**

  By default, write operations cannot be performed on a replication client site. This option enables forwarding of simple client put and delete operations to the master site for processing. These operations must use an implicit NULL transaction ID to be forwarded. Any other write operation that specifies a non-NULL transaction or uses a cursor throws a DatabaseException. This option is turned off by default.

  The master must have an open database handle for the database on which a forwarded write operation is being performed. All sites in the replication group should have the same value for this configuration option. (See the Configuring for Write Forwarding section in the *Berkeley DB Programmer's Reference Guide* for more information.)

- **DB_REPMGR_CONF_PREFMAS_CLIENT**

  This is the client site in a two-site replication group running in preferred master mode. This site automatically takes over as temporary master when the preferred master site is unavailable. Transactions committed on this site when it is operating as the temporary master may be rolled back if they conflict with preferred master transactions. (See the Preferred master mode section in the *Berkeley DB Programmer's Reference Guide* for more information.) This configuration flag may not be set after the `DbEnv::repmgr_start()` (page 629) method is called.

  The other site in the replication group should be specified as the preferred master site using the `DB_REPMGR_CONF_PREFMAS_MASTER` configuration flag.

- **DB_REPMGR_CONF_PREFMAS_MASTER**

  This is the preferred master site in a two-site replication group running in preferred master mode. This site functions as the master whenever its availability permits. When this site returns to the replication group after having been unavailable, it synchronizes with the temporary master and then automatically takes over as master. Transactions committed on this site will not be rolled back. (See the Preferred master mode section in the *Berkeley DB Programmer's Reference Guide* for more information.) This configuration flag may not be set after the `DbEnv::repmgr_start()` (page 629) method is called.

  The other site in the replication group should be specified as the preferred master client site using the `DB_REPMGR_CONF_PREFMAS_CLIENT` configuration flag.
• **DB_REPMGR_CONF_2SITE STRICT**

Replication Manager observes the strict “majority” rule in managing elections, even in a replication group with only two sites. This means the client in a two-site replication group will be unable to take over as master if the original master fails or becomes disconnected. (See the Special considerations for two-site replication groups 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. This option is turned on by default. In preferred master mode, this option cannot be turned off.

**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 DbEnv::rep_set_config() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If setting in-memory replication after the database environment is already opened; if setting preferred master or master leases after replication is started; if setting a Replication Manager configuration flag for a Base API application; or if an invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548)
DbEnv::rep_set_limit()

```c
#include <db_cxx.h>

int DbEnv::rep_set_limit(u_int32_t gbytes, u_int32_t bytes);
```

The `DbEnv::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 `DbEnv::rep_process_message()` 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 `DbEnv::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 `DbEnv::rep_set_limit()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::rep_set_limit()` method may be called at any time during the life of the application.

The `DbEnv::rep_set_limit()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::rep_process_message()` 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 `DbEnv::rep_process_message()` method.

**Class**

`DbEnv`
See Also

Replication and Related Methods (page 548)
DbEnv::rep_set_nsites()

```c
#include <db_cxx.h>

int
DbEnv::rep_set_nsites(u_int32_t nsites);
```

The `DbEnv::rep_set_nsites()` method specifies the total number of participant sites in a replication group. This method should not be used by Replication Manager applications; the number of sites in use by a Replication Manager application is determined dynamically.

The `DbEnv::rep_set_nsites()` method is typically called by Base API applications. (However, see also the `DbEnv::rep_elect()` (page 564) 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 `DbEnv::rep_set_nsites()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

If master leases are in use, the `DbEnv::rep_set_nsites()` method should not be called after the `DbEnv::rep_start()` (page 599) method is called as this could cause you to lose data previously thought to be durable. If master leases are not in use, the `DbEnv::rep_set_nsites()` method may be called at any time during the life of the application.

The `DbEnv::rep_set_nsites()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `nsites`

  An integer specifying the total number of participant sites in the replication group. This number should exclude any replication views.

**Errors**

The `DbEnv::rep_set_nsites()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_set_priority()

```cpp
#include <db_cxx.h>

int
DbEnv::rep_set_priority(u_int32_t priority);
```

The `DbEnv::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.

**Note**

The `DbEnv::repmgr_set_ack_policy()` (page 618) method describes *electable peers*, which are replication sites with a non-zero priority. For some acknowledgement policies, Replication Manager’s computation of the durability result for each new update transaction is sensitive to whether each site in the group is a peer. Therefore, if you change a site’s priority from a non-zero value to 0, or from 0 to a non-zero value, this can invalidate the durability result of previously committed transactions.

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. In preferred master mode, priority values for each site are automatically set and any attempt to change them results in an error.

The `DbEnv::rep_set_priority()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::rep_set_priority()` method may be called at any time during the life of the application.

The `DbEnv::rep_set_priority()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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).

**Errors**

The `DbEnv::rep_set_priority()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

If changing the automatically set priority value in Replication Manager preferred master mode.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_set_request()

```cpp
#include <db_cxx.h>

int DbEnv::rep_set_request(u_int32_t min, u_int32_t max);
```

The `DbEnv::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 `DbEnv::rep_set_request()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::rep_set_request()` method may be called at any time during the life of the application.

The `DbEnv::rep_set_request()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::rep_set_request() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
**DbEnv::rep_set_timeout()**

```
#include <db_cxx.h>

int
DbEnv::rep_set_timeout(int which, u_int32_t timeout);
```

The `DbEnv::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. 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 `DbEnv::rep_set_timeout()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::rep_set_timeout()` method may not be called to set the master lease timeout after the `DbEnv::repmgr_start()` (page 629) method or the `DbEnv::rep_start()` (page 599) method is called. For all other timeouts, the `DbEnv::rep_set_timeout()` method may be called at any time during the life of the application.

The `DbEnv::rep_set_timeout()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**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. Automatic takeover of a subordinate process is most reliable when this value is the same on all sites in the replication group.

- **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. In preferred master mode, a shorter wait time is recommended to facilitate automatic takeovers, so the default wait time is reduced to 1 second.

- **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. In preferred master mode the default is 2 seconds and heartbeat monitoring cannot be turned off because heartbeats are required for automatic takeovers.

- **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. Heartbeat messages are used at client sites to monitor the connection to the master and to help request missing master changes in the absence of master activity. When 0 (the default), no heartbeat messages will be sent. In preferred master mode the default is 0.75 second and heartbeats cannot be turned off because they are required for automatic takeovers.

- **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 the `DbEnv::repmgr_start()` (page 629) method or the `DbEnv::rep_start()` (page 599) method. See also `DbEnv::rep_set_clockskew()` (page 577) method, `DbEnv::rep_set_config()` (page 579) method and Master leases.

- **DB_REP_WRITE_FORWARD_TIMEOUT**
Configure the amount of time a Replication Manager client waits for a response from a forwarded write operation before returning a failure indication. The default value is 5 seconds.

For more information on write forwarding, see the *Berkeley DB Programmer’s Reference Guide*.

**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.

**Errors**

The `DbEnv::rep_set_timeout()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If setting the lease timeout and replication has already been started; if turning off a heartbeat timeout in Replication Manager preferred master mode; if setting a Replication Manager timeout for a Base API application; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

*Replication and Related Methods (page 548)*
DbEnv::rep_set_transport()

#include <db_cxx.h>

int
DbEnv::rep_set_transport(int envid,
    int (*send)(DB_ENV *dbenv,
        const Dbt *control, const Dbt *rec, const DbLsn *lsnp,
        int envid, u_int32_t flags));

The DbEnv::rep_set_transport() method initializes the communication infrastructure for a
database environment participating in a replicated application.

The DbEnv::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 DbEnv::rep_set_transport() method configures operations performed using 
the specified DbEnv handle, not all operations performed on the underlying database 
environment.

The DbEnv::rep_set_transport() method may be called at any time during the life of the 
application.

The DbEnv::rep_set_transport() method either returns a non-zero error value or throws an 
exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::rep_set_transport()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv`

**See Also**

Replication and Related Methods (page 548)


DbEnv::rep_set_view()

```c
#include <db_cxx.h>

int
DbEnv::rep_set_view(int (*partial_func)(DB_ENV *dbenv,
    const char *name, int *result, u_int32_t flags));
```

The `DbEnv::rep_set_view()` method specifies that this environment is a replication view. A replication view is a special type of client that can contain a full or partial copy of the replicated data. A partial view uses a callback to determine the subset of database files to replicate. A replication view does not vote in elections, cannot become master, and cannot contribute to transactional durability.

The `DbEnv::rep_set_view()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::rep_set_view()` method must be called prior to opening the environment. Also the method must be called every time the environment is used after that point. Once an environment is configured as a view, it stays that way for the lifetime of the environment.

The `DbEnv::rep_set_view()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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**

**partial_func**

The `partial_func` callback function determines whether a particular database file should be replicated to the local site. If a NULL callback is specified, all database files will be replicated. The parameters to `partial_func` are as follows:

- `dbenv`
  
The `dbenv` parameter is the enclosing database environment handle.

- `name`
  
The `name` parameter is the physical on-disk file name of the database. In-memory databases are always replicated and do not invoke this callback.

- `result`
  
The `result` parameter is an output parameter indicating whether the file should be replicated. Set it to 0 to reject this file or to a non-zero value to accept this file.
• flags

The *flags* parameter is currently unused.

The *partial* function must return 0 on success and non-zero on failure. If the partial function fails, the environment will panic.

**Errors**

The `DbEnv::rep_set_view()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

The method was called after the environment was opened.

**Class**

`DbEnv`

**See Also**

*Replication and Related Methods (page 548)*
DbEnv::rep_start()

```cpp
#include <db_cxx.h>

int DbEnv::rep_start(Dbt *cdata, u_int32_t flags);
```

The `DbEnv::rep_start()` method configures the database environment as a client or master in a group of replicated database environments.

The `DbEnv::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. Replication view environments are always read-only and can never become master environments. 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 `DbEnv::open()` (page 286) method and must already have been configured to send replication messages by calling the `DbEnv::rep_set_transport()` (page 594) method. If you are starting a view, you must have called the `DbEnv::rep_set_view()` (page 597) method before opening the enclosing database environment.

The `DbEnv::rep_start()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 or view.
  
  - `DB_REP_MASTER`
    
    Configure the environment as a replication master.
Errors

The DbEnv::rep_start() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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 DbEnv::rep_set_transport() (page 594); 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; a view is being started without having called the DbEnv::rep_set_view() (page 597) method before opening the database environment; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_stat()

```c
#include <db_cxx.h>

int DbEnv::rep_stat(DB_REP_STAT **statp, u_int32_t flags);
```

The `DbEnv::rep_stat()` method returns the replication subsystem statistics.

The `DbEnv::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 `DbEnv::set_alloc()` (page 294) 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_datagen;
  The master data generation number 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 \texttt{st\_election\_sec} plus \texttt{st\_election\_usec}).

- \texttt{u\_int32\_t st\_election\_votes};
  The number of votes received during the current election.

- \texttt{uintmax\_t st\_elections};
  The number of elections held.

- \texttt{uintmax\_t st\_elections\_won};
  The number of elections won.

- \texttt{int st\_env\_id};
  The current environment ID.

- \texttt{u\_int32\_t st\_env\_priority};
  The current environment priority.

- \texttt{uintmax\_t st\_ext\_duplicated};
  The number of external file data messages received multiple times. This happens when the client assumes a external file data message was lost because it took too long to arrive, and so the client unnecessarily re-requests it.

- \texttt{uintmax\_t st\_ext\_records};
  The number of external file data messages received and written to disk.

- \texttt{uintmax\_t st\_ext\_rereq};
  The number of external file data messages re-requested because the original message was not received before the configured timeout was reached.

- \texttt{uintmax\_t st\_ext\_update\_rereq};
  The number of external file update messages re-requested. External file update messages are expensive to create and send, so it is best to set the re-request timer high enough to avoid re-requesting them.

- \texttt{u\_int32\_t st\_gen};
  The current master generation number.

- \texttt{uintmax\_t st\_lease\_chk};
  The number of lease validity checks.

- \texttt{uintmax\_t st\_lease\_chk\_misses};
The number of invalid lease validity checks.

- \texttt{uintmax_t st\_lease\_chk\_refresh};
  The number of lease refresh attempts during lease validity checks.

- \texttt{uintmax_t st\_lease\_sends};
  The number of live messages sent while using leases.

- \texttt{uintmax_t st\_log\_duplicated};
  The number of duplicate log records received.

- \texttt{uintmax_t st\_log\_queued};
  The number of log records currently queued.

- \texttt{uintmax_t st\_log\_queued\_max};
  The maximum number of log records ever queued at once.

- \texttt{uintmax_t st\_log\_queued\_total};
  The total number of log records queued.

- \texttt{uintmax_t st\_log\_records};
  The number of log records received and appended to the log.

- \texttt{uintmax_t st\_log\_requested};
  The number of times log records were missed and requested.

- \texttt{int st\_master};
  The current master environment ID.

- \texttt{uintmax_t st\_master\_changes};
  The number of times the master has changed.

- \texttt{u\_int32\_t st\_max\_lease\_sec};
  The number of seconds of the longest lease (the total lease time is \texttt{st\_max\_lease\_sec} plus \texttt{st\_max\_lease\_usec}).

- \texttt{u\_int32\_t st\_max\_lease\_usec};
  The number of microseconds of the longest lease (the total lease time is \texttt{st\_max\_lease\_sec} plus \texttt{st\_max\_lease\_usec}).

- \texttt{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;`
  The number of times that data transmission was stopped to limit the amount of data sent in response to a single call to `DbEnv::rep_process_message()` (page 574).

- `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 role. 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.

- `u_int32_t st_view;`
  The current site is a replication view.

- `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 `DbEnv::rep_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::rep_stat()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::rep_stat() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::rep_stat_print()

```cpp
#include <db_cxx.h>

int DbEnv::rep_stat_print(u_int32_t flags);
```

The DbEnv::rep_stat_print() method displays the replication subsystem statistical information, as described for the DbEnv::rep_stat() method. The information is printed to a specified output channel (see the DbEnv::set_msgfile() (page 342) method for more information), or passed to an application callback function (see the DbEnv::set_msgcall() (page 340) method for more information).

The DbEnv::rep_stat_print() method may not be called before the DbEnv::open() (page 286) method is called.

The DbEnv::rep_stat_print() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::rep_stat_print() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If the method was called before DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548)
**DbEnv::rep_sync()**

```c
#include <db_cxx.h>

int DbEnv::rep_sync(u_int32_t flags);
```

The `DbEnv::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 `DbEnv::rep_set_config()` (page 579) 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 `DbEnv::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 `DbEnv::open()` (page 286) method and must already have been configured to send replication messages by calling the `DbEnv::rep_set_transport()` (page 594) method.

The `DbEnv::rep_sync()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**flags**

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

**Errors**

The `DbEnv::rep_sync()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **DB_REP_JOIN_FAILURE**
  
  If master synchronization requires an internal initialization but automatic internal initializations have been disabled by setting the `DB_REP_CONF_AUTOINIT` flag to 0.

- **EINVAL**
  
  If the database environment was not already configured to communicate with a replication group by a call to `DbEnv::rep_set_transport()` (page 594); the database environment was not already opened; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`
See Also

Replication and Related Methods (page 548)
DbEnv::repmgr_channel()

```c
#include <db_cxx.h>

int
DbEnv::repmgr_channel(int eid, DB_CHANNEL **channelp, u_int32_t flags);
```

The `DbEnv::repmgr_channel()` method returns a DbChannel handle. This is used to create and manage custom message traffic between the sites in the replication group.

This method allocates memory for the handle, returning a pointer to the structure in the memory to which `channelp` refers. To release the allocated memory and discard the handle, call the `DbChannel::close()` method.

The `DbEnv::repmgr_channel()` method may be called at any time after `DbEnv::repmgr_start()` has been called with a 0 return code.

The `DbEnv::repmgr_channel()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **eid**
  
  This parameter must be set to one of the following:
  
  - The numerical environment ID of a remote site in the replication group.
  - `DB_EID_MASTER`

  Messages sent on this channel are sent only to the master site. Note that messages are always sent to the current master, even if the master has changed since the channel was opened. If the current master is disconnected or unknown, the operation fails and Replication Manager returns an error code.

  If the local site is the master, then sending messages on this channel will result in the local site receiving those messages echoed back to itself.

- **channelp**

  References memory into which a pointer to the allocated handle is copied.

- **flags**

  This parameter is currently unused, and must be set to 0.

**Errors**

The `DbEnv::repmgr_channel()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
**DB_REP_UNAVAIL**

If the `eid` parameter is `DB_EID_MASTER` but the current master is disconnected or unknown.

**EINVAL**

If this method is called from a Base API application; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

[Replication and Related Methods](#), [The DbChannel Handle](#)
DbEnv::repmgr_local_site()

```cpp
#include <db_cxx.h>

int DbEnv::repmgr_local_site(DB_SITE **sitep);
```

The `DbEnv::repmgr_local_site()` method returns a `DbSite` handle that defines the local site’s host/port network address. You use the `DbSite` handle to configure and manage replication sites.

This method allocates memory for the handle, returning a pointer to the structure in the memory to which `sitep` refers. To release the allocated memory and discard the handle, call the `DbSite::close()` (page 636) method.

The `DbEnv::repmgr_local_site()` method may be called at any time after the environment handle has been created.

The `DbEnv::repmgr_local_site()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `sitep`

  References memory into which a pointer to the allocated handle is copied.

**Errors**

The `DbEnv::repmgr_local_site()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- `EINVAL`

  If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

- Replication and Related Methods (page 548)
DbEnv::repmgr_get_ack_policy()

```cpp
#include <db_cxx.h>

int DbEnv::repmgr_get_ack_policy(int *ack_policyp);
```

The `DbEnv::repmgr_get_ack_policy()` method returns the Replication Manager's client acknowledgment policy. This is configured using the `DbEnv::repmgr_set_ack_policy()` (page 618) method.

The `DbEnv::repmgr_get_ack_policy()` method may be called at any time during the life of the application.

The `DbEnv::repmgr_get_ack_policy()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`ack_policyp`

The `ack_policyp` parameter references memory into which the Replication Manager's client acknowledgement policy is copied.

**Class**

`DbEnv`

**See Also**

`Replication and Related Methods (page 548), DbEnv::repmgr_set_ack_policy()` (page 618)
DbEnv::repmgr_get_incoming_queue_max()

```c
#include <db_cxx.h>

int DbEnv::repmgr_get_incoming_queue_max(u_int32_t *gbytesp,
                                         u_int32_t *bytesp);
```

The `DbEnv::repmgr_get_incoming_queue_max()` method returns the byte-count limit on the amount of dynamic memory used by the Replication Manager incoming queue. This value is configurable using the `DbEnv::repmgr_set_incoming_queue_max()` method.

The `DbEnv::repmgr_get_incoming_queue_max()` method may be called at any time during the life of the application.

The `DbEnv::repmgr_get_incoming_queue_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**gbytesp**

The `gbytesp` parameter references memory into which the gigabytes component of the Replication Manager incoming queue limit is copied.

**bytesp**

The `bytesp` parameter references memory into which the bytes component of the Replication Manager incoming queue is copied.

**Class**

`DbEnv`

**See Also**

`Replication and Related Methods (page 548)`, `DbEnv::repmgr_set_incoming_queue_max()` (page 620)
# DbEnv::repmgr_msg_dispatch()

```c
#include <db_cxx.h>

int DbEnv::repmgr_msg_dispatch(
    void (*msg_dispatch_fcn) (DbEnv *env, DbChannel *channel,
     Dbt *request, u_int32_t nrequest,
     u_int32_t cb_flags),
    u_int32_t flags);
```

Sets the message dispatch function. This function is responsible for receiving messages sent from remote sites using either the `DbChannel::send_msg()` (page 553) or `DbChannel::send_request()` (page 555) methods. If the message received by this function was sent using the `DbChannel::send_request()` (page 555) method then no response is required. If the message was sent using the `DbChannel::send_request()` (page 555) method, then this function must send a response using the `DbChannel::send_msg()` (page 553) method.

For best results, the `DbEnv::repmgr_msg_dispatch()` method should be called before the Replication Manager has been started.

The `DbEnv::repmgr_msg_dispatch()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **msg_dispatch_fcn**
  
  This parameter is the application-specific function used to handle messages sent over Replication Manager message channels. It takes four parameters:

  - **channel**
    
    Provides the DbChannel to be used to send a response back to the originator of the message. If the message was sent by the remote site using `DbChannel::send_request()` (page 555) then this function should send a response back to the originator using the channel provided on this parameter. The message should be sent by calling `DbChannel::send_msg()` (page 553) exactly once.
    
    This channel is valid only during the current invocation of the dispatch function; it is destroyed when the dispatch function returns. The application may not save a copy of the pointer and use it later elsewhere. Methods that do not make sense in the context of a message dispatch function (such as `DbChannel::send_request()` (page 555) and `DbChannel::close()` (page 552)) will be rejected with EINVAL.

  - **request**
    
    Array of Dbts containing the message received from the remote site.

  - **nrequest**
Specifies the number of elements in the request array.

- cb_flags
  
  This flag is DB_REPMGR_NEED_RESPONSE if the message requires a response. Otherwise, it is 0.

This function does not return a value. If the function encounters an error, you can reflect the error back to the originator of the message by formatting an error message of your own design into the response.

flags

This parameter is currently unused, and must be set to 0.

Errors

The DbEnv::repmgr_msg_dispatch() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::repmgr_set_ack_policy()

```c
#include <db_cxx.h>

int DbEnv::repmgr_set_ack_policy(int ack_policy);
```

The DbEnv::repmgr_set_ack_policy() method specifies how master and client sites will handle acknowledgment of replication messages which are necessary for "permanent" records. View sites never send these acknowledgements and are not counted by any acknowledgement policy. The current implementation requires all sites in a replication group to 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 DbEnv::rep_set_timeout() method. If an insufficient number of client acknowledgements have been received, then the master will invoke the event callback function, if set, with the DB_EVENT_REP_PERM_FAILED value. (See the Choosing a Replication Manager Ack Policy section in the Berkeley DB Programmer's Reference Guide for more information.)

The DbEnv::repmgr_set_ack_policy() method configures a database environment, not only operations performed using the specified DbEnv handle.

The DbEnv::repmgr_set_ack_policy() method may be called at any time during the life of the application.

The DbEnv::repmgr_set_ack_policy() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_AVAILABLE**
The master should wait until all currently connected replication clients have acknowledged each permanent replication message. This policy will then invoke the DB_EVENT_REP_PERM_FAILED event if fewer than a quorum of clients acknowledged during that time.

- **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 DbEnv::repmgr_set_ack_policy() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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**

DbEnv

**See Also**

Replication and Related Methods (page 548)
The `DbEnv::repmgr_set_incoming_queue_max()` method sets a byte-count limit on the amount of dynamic memory used by the Replication Manager incoming queue. When the incoming queue reaches this limit, incoming messages are dropped until the Replication Manager processes some of the messages already in the queue. Any dropped messages are automatically rerequested at a later time. This limit is not a hard limit, and the message that exceeds this limit is the last one to be enqueued.

The Replication Manager incoming queue has a default size limit of 100MB. We recommend a minimum size limit of 32MB.

If the values passed to the `DbEnv::repmgr_set_incoming_queue_max()` method are both zero, then the incoming queue size limit is turned off.

We recommend increasing the incoming queue size limit in the following cases:

- Master leases are enabled, particularly if there are many `DB_REP_LEASE_EXPIRED` errors.
- Clients are far behind the master.
- The master is using bulk transfer to send groups of records to the clients in a single network transfer.
- The master has blob databases and is performing many blob operations on them.
- The master is performing intensive write operations.

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_incoming_queue_max", one or more whitespace characters, and the limit specified in two parts: the gigabytes and the bytes values. For example, "repmgr_set_incoming_queue_max 0 104857600" sets a 100 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 `DbEnv::repmgr_set_incoming_queue_max()` method configures a database environment, not only operations performed using the specified `DbEnv` handle.

The `DbEnv::repmgr_set_incoming_queue_max()` method may be called at any time during the life of the application. If the limit is reduced, messages already in the queue are not removed, but further incoming messages are not added to the queue until its size drops below the new limit.

The `DbEnv::repmgr_set_incoming_queue_max()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.
Parameters

**gbytes**

The `gbytes` parameter specifies the number of gigabytes which, when added to the `bytes` parameter, specifies the maximum size limit of the incoming queue.

**bytes**

The `bytes` parameter specifies the number of bytes which, when added to the `gbytes` parameter, specifies the maximum size limit of the incoming queue.

Class

DbEnv

See Also

[Replication and Related Methods (page 548)]
DbEnv::repmgr_set_socket() method specifies a socket callback. The socket callback enables an application to exert additional control over the connections used by Replication Manager.

Replication Manager creates IPv6 connections whenever possible but also supports IPv4. In cases where there are multiple addresses defined for a site, Replication Manager attempts connections first on any IPv6 addresses and then on any IPv4 addresses until one succeeds. Replication Manager relies on platform configuration settings and defaults to govern the use of IPv4-mapped IPv6 addresses in cases where one site is using IPv6 and the other site is using IPv4.

If your application requires more precise control over connections, you can use this method to provide a socket callback. The socket callback provides access to each socket before it is used in a connection attempt and provides the ability to accept or reject the socket. Within the socket callback's approval_func, you can test socket properties such as address characteristics (e.g. whether it is IPv6 or IPv4) and socket options. You can optionally change the IPV6_V6ONLY socket option to enable or disable use of IPv4-mapped IPv6 addresses. If the socket callback rejects a socket or if a connection attempt does not succeed, Replication Manager tries any other addresses defined for the site.

The only supported change to the socket in this callback is the IPV6_V6ONLY socket option. Any other changes may adversely affect the operation of Replication Manager's TCP/IP communications infrastructure.

The DbEnv::repmgr_set_socket() method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

The DbEnv::repmgr_set_socket() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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

approval_func

The approval_func callback function determines whether a particular socket should be used in a connection attempt to a site. The parameters to approval_func are as follows:
• dbenv

The dbenv parameter is the enclosing database environment handle.

• socket

The socket parameter is the socket that is about to be used in a connection attempt to a site.

• result

The result parameter is an output parameter indicating whether to use this socket. Set it to 0 to reject this socket or to a non-zero value to accept this socket.

• flags

The flags parameter is currently unused.

The approval function must return 0 on success and non-zero on failure. If the approval function fails, the environment will panic.

Errors

The DbEnv::repmgr_set_socket() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or 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

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::repmgr_site()

```cpp
#include <db_cxx.h>

int
DbEnv::repmgr_site(const char * host, u_int16_t port,
         DB_SITE **sitep, u_int32_t flags);
```

The DbEnv::repmgr_site() method returns a DbSite handle that defines a site's network address. You use the DbSite handle to configure and manage a Replication Manager site.

This method allocates memory for the handle, returning a pointer to the structure in the memory to which sitep refers. To release the allocated memory and discard the handle, call the DbSite::close() (page 636) method.

You must use the exact same host identification string and port number to refer to a given site throughout your application and on each of its sites.

The DbEnv::repmgr_site() method may be called at any time after the environment handle has been created.

The DbEnv::repmgr_site() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**host**

The site's host identification string, generally a TCP/IP host name. IPv6 and IPv4 addresses in numeric format are also accepted. If multiple addresses are defined for a host name, an IPv6 address is used whenever possible.

**port**

The port number on which the site is listening.

**sitep**

References memory into which a pointer to the allocated handle is copied.

**flags**

This parameter is currently unused, and must be set to 0.

**Errors**

The DbEnv::repmgr_site() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If this method is called from a Base API application, or if an invalid flag value or parameter was specified.
**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548), The DbSite Handle (page 550)
DbEnv::repmgr_site_by_eid()

```c
#include <db_cxx.h>

int DbEnv::repmgr_site_by_eid(int eid, DB_SITE **sitep)
```

The DbEnv::repmgr_site_by_eid() method returns a DbSite handle based on the site's environment ID value. You use the DbSite handle to configure and manage replication sites.

This method allocates memory for the handle, returning a pointer to the structure in the memory to which sitep refers. To release the allocated memory and discard the handle, call the DbSite::close() (page 636) method.

The DbEnv::repmgr_site_by_eid() method may be called at any time after opening the environment.

The DbEnv::repmgr_site_by_eid() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **eid**
  The environment ID of the site for which you want to create the DbSite handle. You can obtain a site's EID by using the DbSite::get_eid() (page 560) method.

- **sitep**
  References memory into which a pointer to the allocated handle is copied.

**Errors**

The DbEnv::repmgr_site() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **DB_NOTFOUND**
  Returned if there is no site corresponding to the supplied eid value.

- **EINVAL**
  If this method is called from a Base API application, or if an invalid flag value or parameter was specified.

**Class**

DbEnv

**See Also**

Replication and Related Methods (page 548)
DbEnv::repmgr_site_list()

#include <db_cxx.h>

int DbEnv::repmgr_site_list(u_int *countp, DB_REPMGR_SITE **listp);

The `DbEnv::repmgr_site_list()` method returns the status of the sites currently known by the Replication Manager.

The `DbEnv::repmgr_site_list()` method creates an array of statistical structures 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 DbEnv::set_alloc() (page 294) 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 and the DB_REPMGR_ISVIEW constants. The DB_REPMGR_ISPEER value means that the site is a possible client-to-client peer. The DB_REPMGR_ISVIEW value means that the site is a view.

The `DbEnv::repmgr_site_list()` method may be called only after the `DbEnv::repmgr_start()` (page 629) method has been called with a 0 return code.

The `DbEnv::repmgr_site_list()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::repmgr_start()

```c
#include <db_cxx.h>

int
DbEnv::repmgr_start(int nthreads, u_int32_t flags);
```

The `DbEnv::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. Use `DbEnv::repmgr_site()` (page 624) to obtain a DbSite handle, then use that handle to configure the sites in the replication group.
   a. Use `DbSite::set_config()` (page 562) to configure sites in the replication group.
   b. Use `DbSite::remove()` (page 561) to remove a site from the replication group.
2. Call `DbEnv::repmgr_set_ack_policy()` (page 618) to configure the message acknowledgment policy which best supports the replication group's transactional needs.
3. Call `DbEnv::rep_set_priority()` (page 587) to configure the local site's election priority.
4. Call `DbEnv::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 `DbEnv::repmgr_start()` method automatically opens the socket in the first process to call it, and skips this step in the later calls from other subordinate processes.

The `DbEnv::repmgr_start()` method may not be called before the `DbEnv::open()` (page 286) method is called to open the environment. In addition, this method may not be called before at least one replication site has been configured using the `DbSite` class. In addition, the local environment must be opened with the `DB_THREAD` flag set. If you are starting a view, you must call the `DbEnv::rep_set_view()` (page 597) method before opening the local environment.

The `DbEnv::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 `DbEnv::repmgr_start()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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. The TCP/IP listener process can change this value after the Replication Manager is started with a subsequent call to the `DbEnv::repmgr_start()` method.

**flags**

The `flags` parameter must be set to one of the following values when first starting the Replication Manager:

- **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, view, or preferred master 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.

If the Replication Manager is already started, a `flags` value of 0 can be used when making a subsequent call to change the value of `nthreads` or when starting a subordinate process.

**Errors**

The `DbEnv::repmgr_start()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DB_REP_UNAVAIL**

The local site tried to join the group, but was unable to do so for some reason (because a master site is not available, or because insufficient clients are running to acknowledge the new site). When that happens the application should pause and retry adding the site until it completes successfully.

**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; a view is being started without having called the `DbEnv::rep_set_view()` (page 597) method before opening the database environment; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

Replication and Related Methods (page 548)
DbEnv::repmgr_stat()

#include <db_cxx.h>

int
DbEnv::repmgr_stat(DB_REPMGR_STAT **statp, u_int32_t flags);

The DbEnv::repmgr_stat() method returns the Replication Manager statistics.

The DbEnv::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 DbEnv::set_alloc() (page 294) 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.

- u_int32_t st_elect_threads;
  The number of currently active election threads.

- uintmax_t st_incoming_msgs_dropped;
  The number of incoming messages that were dropped because the incoming queue was full.
  (Berkeley DB replication is tolerant of dropped messages, and will automatically request
  retransmission of any missing messages as needed.)

- u_int32_t st_incoming_queue_bytes;
  Bytes component of the memory consumption for the messages currently in the incoming
  queue.

- u_int32_t st_incoming_queue_gbytes;
  Gigabytes component of the memory consumption for the messages currently in the
  incoming queue.

- u_int32_t st_max_elect_threads;
  The number of election threads for which space is reserved.

- 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.

- **u_int32_t st_site_participants;**
  The number of participant sites in the replication group.

- **u_int32_t st_site_total;**
  The total number of sites in the replication group.

- **u_int32_t st_site_views;**
  The number of view sites in the replication group.

- **uintmax_t st_takeovers;**
  The number of times a subordinate process took over as the replication process after a previous replication process has finished successfully.

- **uintmax_t st_write_ops_forwarded;**
  The total number of outgoing write operations forwarded by this client.

- **uintmax_t st_write_ops_received;**
  The total number of incoming forwarded write operations received by this master.

The `DbEnv::repmgr_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::repmgr_stat()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::repmgr_stat() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

EINVAL

If the method was called before DbEnv::open() (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv

See Also

Replication and Related Methods (page 548)
DbEnv::repmgr_stat_print()

```cpp
#include <db_cxx.h>

int DbEnv::repmgr_stat_print(u_int32_t flags);
```

The `DbEnv::repmgr_stat_print()` method displays the Replication Manager statistical information, as described for the `DbEnv::repmgr_stat()` method. The information is printed to a specified output channel (see the `DbEnv::set_msgfile()` method for more information), or passed to an application callback function (see the `DbEnv::set_msgcall()` method for more information).

The `DbEnv::repmgr_stat_print()` method may not be called before the `DbEnv::open()` method is called.

The `DbEnv::repmgr_stat_print()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::repmgr_stat_print()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called before `DbEnv::open()` was called; or if an invalid flag value or parameter was specified.

**Class**

`DbEnv`

**See Also**

- Replication and Related Methods (page 548)
DbSite::close()

```cpp
#include <db_cxx.h>

int DbSite::close();
```

The DbSite::close() method deallocates the DbSite handle. The handle must not be accessed again after this method is called, regardless of the return value.

Use of this method does not in any way affect the configuration of the site to which the handle refers, or of the replication group in general.

All DbSite handles must be closed before the owning DbEnv handle is closed.

The DbSite::close() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Class

DbSite

See Also

Replication and Related Methods (page 548)
DbEnv::txn_applied()

```c
#include <db_cxx.h>

int DB_ENV->txn_applied(DB_ENV *env, DB_TXN_TOKEN *token,
                         db_timeout_t timeout, u_int32_t flags);
```

The DbEnv::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 DbEnv::txn_applied() method may not be called before the DbEnv::open() (page 286) method.

The DbEnv::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 change of master.

**Parameters**

- **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 DbTxn::set_commit_token() (page 639) 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.

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

The `DbEnv::txn_applied()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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).

**DbDeadlockException** or **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.

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**EINVAL**

If the database environment was not already opened; or if the specified token was generated from a non-replicated database environment.

Class

**DbEnv**

See Also

Transaction Subsystem and Related Methods (page 664), Replication and Related Methods (page 548)
DbTxn::set_commit_token()

```cpp
#include <db_cxx.h>

int
DbTxn::set_commit_token(DB_TXN_TOKEN *buffer);
```

The `DbTxn::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 `DbTxn::set_commit_token()` method may be called at any time after the `DbEnv::txn_begin()` (page 674) method has been called, and before `DbTxn::commit()` (page 686) has been called.

The `DbTxn::set_commit_token()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbTxn::set_commit_token()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbTxn`

**See Also**

`Transaction Subsystem and Related Methods (page 664)`, `Replication and Related Methods (page 548)`
Chapter 12. The DbSequence 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 DbSequence (page 642) method.

For more information on sequences, see the Berkeley DB Programmer’s Reference Guide guide.
# Sequences and Related Methods

<table>
<thead>
<tr>
<th>Sequences and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DbSequence</td>
<td>Create a sequence handle</td>
</tr>
<tr>
<td>DbSequence::close()</td>
<td>Close a sequence</td>
</tr>
<tr>
<td>DbSequence::get()</td>
<td>Get the next sequence element(s)</td>
</tr>
<tr>
<td>DbSequence::get_dbp()</td>
<td>Return a handle for the underlying sequence database</td>
</tr>
<tr>
<td>DbSequence::get_key()</td>
<td>Return the key for a sequence</td>
</tr>
<tr>
<td>DbSequence::initial_value()</td>
<td>Set the initial value of a sequence</td>
</tr>
<tr>
<td>DbSequence::open()</td>
<td>Open a sequence</td>
</tr>
<tr>
<td>DbSequence::remove()</td>
<td>Remove a sequence</td>
</tr>
<tr>
<td>DbSequence::stat()</td>
<td>Return sequence statistics</td>
</tr>
<tr>
<td>DbSequence::stat_print()</td>
<td>Print sequence statistics</td>
</tr>
<tr>
<td><strong>Sequences Configuration</strong></td>
<td></td>
</tr>
<tr>
<td>DbSequence::set_cachesize(),</td>
<td>Set/get the cache size of a sequence</td>
</tr>
<tr>
<td>DbSequence::get_cachesize()</td>
<td></td>
</tr>
<tr>
<td>DbSequence::set_flags(),</td>
<td>Set/get the flags for a sequence</td>
</tr>
<tr>
<td>DbSequence::get_flags()</td>
<td></td>
</tr>
<tr>
<td>DbSequence::set_range(),</td>
<td>Set/get the range for a sequence</td>
</tr>
<tr>
<td>DbSequence::get_range()</td>
<td></td>
</tr>
</tbody>
</table>
The DbSequence Handle

DbSequence

```cpp
#include <db_cxx.h>

class DbSequence {
public:
    DbSequence(Db *db, u_int32_t flags);
    ~DbSequence();

    DB_SEQUENCE *DbSequence::get_DB();
    const DB *DbSequence::get_const_DB() const;
    static DbSequence *DbSequence::get_DbSequence(DB *db);
    static const DbSequence *
        DbSequence::get_const_DbSequence(const DB *db);

    ...  
};
```

The DbSequence handle is the handle used to manipulate a sequence object. A sequence object is stored in a record in a database.

DbSequence handles are free-threaded if the DB_THREAD flag is specified to the DbSequence::open() (page 653) method when the sequence is opened. Once the DbSequence::close() (page 644) or DbSequence::remove() (page 655) 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 DbSequence::get() (page 645) method either singly or in groups depending on its delta parameter.

Calling the DbSequence::close() (page 644) or DbSequence::remove() (page 655) methods discards this handle.

**Parameters**

- **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 throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.
Class

DbSequence

See Also

Sequences and Related Methods (page 641)
DbSequence::close()

```cpp
#include <db_cxx.h>

int DbSequence::close(u_int32_t flags);
```

The `DbSequence::close()` method closes the sequence handle. Any unused cached values are lost.

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

The `DbSequence::close()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`flags`

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

**Errors**

The `DbSequence::close()` method method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbSequence`

**See Also**

`Sequences and Related Methods (page 641) `
DbSequence::get()

#include <db_cxx.h>

int DbSequence::get(DbTxn *txnid, u_int32_t delta, db_seq_t *retp, 
    u_int32_t flags);

The DbSequence::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 DbSequence::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 DbSequence::set_flags() (page 658) method.

The DbSequence::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 DbEnv::txn_begin() (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the txnid parameter is a handle returned from DbEnv::cdsgroup_begin() (page 666); 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 DbSequence::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 DbSequence::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 DbSequence::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

DbSequence

See Also

Sequences and Related Methods (page 641)
DbSequence::get_cachesize()

```cpp
#include <db_cxx.h>

int DbSequence::get_cachesize(u_int32_t *sizep);
```

The DbSequence::get_cachesize() method returns the current cache size.

The DbSequence::get_cachesize() method may be called at any time during the life of the application.

The DbSequence::get_cachesize() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`sizep`

The DbSequence::get_cachesize() method returns the current cache size in `sizep`.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
DbSequence::get_dbp()

```cpp
#include <db_cxx.h>

int DbSequence::get_dbp(Db **dbp);
```

The `DbSequence::get_dbp()` method returns the database handle used by the sequence.

The `DbSequence::get_dbp()` method may be called at any time during the life of the application.

The `DbSequence::get_dbp()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **dbp**

  The `dbp` parameter references memory into which a pointer to the database handle is copied.

**Class**

- DbSequence

**See Also**

- Sequences and Related Methods (page 641)
DbSequence::get_flags()

```cpp
#include <db_cxx.h>

int DbSequence::get_flags(u_int32_t *flagsp);
```

The `DbSequence::get_flags()` method returns the current flags.

The `DbSequence::get_flags()` method may be called at any time during the life of the application.

The `DbSequence::get_flags()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **flagsp**

  The `DbSequence::get_flags()` method returns the current flags in `flagsp`.

**Class**

`DbSequence`

**See Also**

- [Sequences and Related Methods (page 641)](#)
The `DbSequence::get_key()` method returns the key for the sequence.

The `DbSequence::get_key()` method may be called at any time during the life of the application.

The `DbSequence::get_key()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

key

The `key` parameter references memory into which a pointer to the key data is copied.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
DbSequence::get_range()

#include <db_cxx.h>

int DbSequence::get_range(u_int32_t, db_seq_t *minp, db_seq_t *maxp);

The DbSequence::get_range() method returns the range of values in the sequence.

The DbSequence::get_range() method may be called at any time during the life of the application.

The DbSequence::get_range() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

minp
The DbSequence::get_range() method returns the minimum value in minp.

maxp
The DbSequence::get_range() method returns the maximum value in maxp.

Class

DbSequence

See Also

Sequences and Related Methods (page 641)
DbSequence::initial_value()

```c
#include <db_cxx.h>

int DbSequence::initial_value(db_seq_t value);
```

Set the initial value for a sequence. This call is only effective when the sequence is being created.

The DbSequence::initial_value() method may not be called after the DbSequence::open() (page 653) method is called.

The DbSequence::initial_value() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

value

The initial value to set.

**Errors**

The DbSequence::initial_value() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
DbSequence::open()

```cpp
#include <db_cxx.h>

int DbSequence::open(DbTxn *txnid, Dbt *key, u_int32_t flags);
```

The DbSequence::open() method opens the sequence represented by the key. The key must be compatible with the underlying database specified in the corresponding call to DbSequence (page 642).

The DbSequence::open() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbSequence::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 DbSequence handle returned by DbSequence::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.
    
    Be aware that enabling this flag will serialize calls to DB when using the handle across threads. If concurrent scaling is important to your application we recommend opening separate handles for each thread (and not specifying this flag), rather than sharing handles between threads.

- **txnid**
  
  If the operation is part of an application-specified transaction, the txnid parameter is a transaction handle returned from DbEnv::txn_begin() (page 674); if the operation is part of...
a Berkeley DB Concurrent Data Store group, the \texttt{txnid} parameter is a handle returned from \texttt{DbEnv::cdsgroup_begin()} (page 666); 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 \texttt{DbSequence} handle require the \texttt{DbSequence} handle itself be transactionally protected during its open if the open creates the sequence.

\textbf{Class}

\texttt{DbSequence}

\textbf{See Also}

Sequences and Related Methods (page 641)
DbSequence::remove()

```c
#include <db_cxx.h>

int
DbSequence::remove(u_int32_t flags);
```

The `DbSequence::remove()` method removes the sequence from the database. This method should not be called if there are other open handles on this sequence.

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

The `DbSequence::remove()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

### Parameters

**txnid**

If the operation is part of an application-specified transaction, the `txnid` parameter is a transaction handle returned from `DbEnv::txn_begin()` (page 674); if the operation is part of a Berkeley DB Concurrent Data Store group, the `txnid` parameter is a handle returned from `DbEnv::cdsgroup_begin()` (page 666); 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 `DbSequence::remove()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

### Class

`DbSequence`
See Also

Sequences and Related Methods (page 641)
DbSequence::set_cachesize()

```c++
#include <db_cxx.h>

int DbSequence::set_cachesize(u_int32_t size);
```

Configure the number of elements cached by a sequence handle.

The `DbSequence::set_cachesize()` method may not be called after the `DbSequence::open()` (page 653) method is called.

The `DbSequence::set_cachesize()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

`size`

The number of elements in the cache.

**Errors**

The `DbSequence::set_cachesize()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbSequence`

**See Also**

`Sequences and Related Methods (page 641)`
DbSequence::set_flags()

```c
#include <db_cxx.h>

int DbSequence::set_flags(u_int32_t flags);
```

Configure a sequence. The flags are only effective when creating a sequence. Calling DbSequence::set_flags() is additive; there is no way to clear flags.

The DbSequence::set_flags() method may not be called after the DbSequence::open() (page 653) method is called.

The DbSequence::set_flags() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbSequence::set_flags() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  An invalid flag value or parameter was specified.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
DbSequence::set_range()

```
#include <db_cxx.h>

int DbSequence::set_range(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 `DbSequence::set_range()` method may not be called after the `DbSequence::open()` method is called.

The `DbSequence::set_range()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **min**
  Specifies the minimum value for the sequence.

- **max**
  Specifies the maximum value for the sequence.

**Errors**

The `DbSequence::set_range()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  An invalid flag value or parameter was specified.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
DbSequence::stat()

#include <db_cxx.h>

int
DbSequence::stat(DB_SEQUENCE_STAT **spp, u_int32_t flags);

The DbSequence::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 DbEnv::set_alloc() (page 294) 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 DbSequence::stat() may be out-of-date.

The DbSequence::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 DbSequence::stat() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

The statistics are stored in a structure of type DB_SEQUENCE_STAT. The following fields will be filled in:

- **u_int32_t st_cache_size;**
  The number of values that will be cached in this handle.

- **db_seq_t st_current;**
  The current value of the sequence in the database.

- **u_int32_t st_flags;**
  The flags value for the sequence.

- **db_seq_t st_last_value;**
  The last cached value of the sequence.

- **db_seq_t st_max;**
  The maximum permitted value of the sequence.

- **db_seq_t st_min;**
  The minimum permitted value of the sequence.
• `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_value;`
  The current cached value of the sequence.

• `uintmax_t st_wait;`
  The number of times a thread of control was forced to wait on the handle mutex.

**Parameters**

**flags**

The `flags` parameter must be set 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 printing their values.

**Class**

`DbSequence`

**See Also**

`Sequences and Related Methods (page 641)`
DbSequence::stat_print()

```
#include <db_cxx>

int DbSequence::stat_print(u_int32_t flags);
```

The DbSequence::stat_print() method prints diagnostic information to the output channel described by the DbEnv::set_msgfile() method.

The DbSequence::stat_print() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_ALL**
  
  Display all available information.

- **DB_STAT_CLEAR**
  
  Reset statistics after printing their values.

**Class**

DbSequence

**See Also**

Sequences and Related Methods (page 641)
Chapter 13. The DbTxn Handle

```cpp
class DbTxn {
public:
    DB_TXN *DbTxn::get_DB_TXN();
    const DB_TXN *DbTxn::get_const_DB_TXN() const;
    static DbTxn *DbTxn::get_DbTxn(DB_TXN *txn);
    static const DbTxn *DbTxn::get_const_DbTxn(const DB_TXN *txn);
    ...}
```

The DbTxn object is the handle for a transaction. Methods of the DbTxn handle are used to configure, abort and commit the transaction. DbTxn handles are provided to Db methods in order to transactionally protect those database operations.

DbTxn 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 DbTxn handle. Once the DbTxn::abort() (page 685) or DbTxn::commit() (page 686) 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 DbEnv::txn_begin() (page 674), DbTxn::abort() (page 685) and DbTxn::commit() (page 686).

Each DbTxn object has an associated DB_TXN struct, which is used by the underlying implementation of Berkeley DB and its C++ language API. The DbTxn::get_DB_TXN() method returns a pointer to this struct. Given a const DbTxn object, txnMget_const_DB_TXN() returns a const pointer to the same struct.

Given a DB_TXN struct, the DbTxn::get_DbTxn() method returns the corresponding DbTxn object, if there is one. If the DB_TXN object was not associated with a DbTxn (that is, it was not returned from a call to DbTxn::get_DB_TXN()), then the result of DbTxn::get_DbTxn is undefined. Given a const DB_TXN struct, DbTxn::get_const_DbTxn() returns the associated const DbTxn object, if there is one.

These methods may be useful for Berkeley DB applications including both C and C++ language software. It should not be necessary to use these calls in a purely C++ application.
## Transaction Subsystem and Related Methods

<table>
<thead>
<tr>
<th>Transaction Subsystem and Related Methods</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DbEnv::txn_recover()</code></td>
<td>Distributed transaction recovery</td>
</tr>
<tr>
<td><code>DbEnv::txn_checkpoint()</code></td>
<td>Checkpoint the transaction subsystem</td>
</tr>
<tr>
<td><code>DbEnv::txn_stat()</code></td>
<td>Return transaction subsystem statistics</td>
</tr>
<tr>
<td><code>DbEnv::txn_stat_print()</code></td>
<td>Print transaction subsystem statistics</td>
</tr>
<tr>
<td><code>DbTxn::set_timeout()</code></td>
<td>Set transaction timeout</td>
</tr>
</tbody>
</table>

### Transaction Subsystem Configuration

| `DbEnv::set_timeout()`, `DbEnv::get_timeout()` | Set/get lock and transaction timeout |
| `Db::get_transactional()`                     | Does the Db have transaction support |
| `DbEnv::cdsgroup_begin()`                     | Get a locker ID in Berkeley DB Concurrent Data Store |
| `DbEnv::set_tx_max()`, `DbEnv::get_tx_max()`  | Set/get maximum number of transactions |
| `DbEnv::set_tx_timestamp()`, `DbEnv::get_tx_timestamp()` | Set/get recovery timestamp |

### Transaction Operations

| `DbEnv::txn_begin()`                        | Begin a transaction |
| `DbTxn::abort()`                            | Abort a transaction |
| `DbTxn::commit()`                           | Commit a transaction |
| `DbTxn::discard()`                          | Discard a prepared but not resolved transaction handle |
| `DbTxn::id()`                               | Return a transaction's ID |
| `DbTxn::prepare()`                          | Prepare a transaction for commit |
| `DbTxn::set_name()`, `DbTxn::get_name()`     | Associate a string with a transaction |
| `DbTxn::set_priority()`, `DbTxn::get_priority()` | Set/get transaction's priority |
Db::get_transactional()

```cpp
#include <db_cxx.h>

int
Db::get_transactional()
```

The `Db::get_transactional()` method returns non-zero if the `Db` handle has been opened in a transactional mode, otherwise it returns 0.

The `Db::get_transactional()` method may be called at any time during the life of the application.

**Class**

Db

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbEnv::cdsgroup_begin()

```c
#include <db_cxx.h>

int
DbEnv::cdsgroup_begin(DbTxn **tid);
```

The `DbEnv::cdsgroup_begin()` method allocates a locker ID in an environment configured for Berkeley DB Concurrent Data Store applications. It copies a pointer to a `DbTxn` that uniquely identifies the locker ID into the memory to which `tid` refers. Calling the `DbTxn::commit()` (page 686) method will discard the allocated locker ID.

See Berkeley DB Concurrent Data Store applications for more information about when this is required.

This method may not be called on a sliced database.

The `DbEnv::cdsgroup_begin()` method may be called at any time during the life of the application.

The `DbEnv::cdsgroup_begin()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Errors**

The `DbEnv::cdsgroup_begin()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

If this method is called on a sliced database; or if an invalid flag value or parameter was specified.

**ENOMEM**

The maximum number of lockers has been reached.

**Class**

`DbEnv`, `DbTxn`

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbEnv::get_tx_max()

```c
#include <db_cxx.h>

int
DbEnv::get_tx_max(u_int32_t *tx_maxp);
```

The DbEnv::get_tx_max() method returns the maximum number of active transactions currently configured for the environment. You can manage this value using the DbEnv::set_tx_max() (page 669) method.

The DbEnv::get_tx_max() method may be called at any time during the life of the application.

The DbEnv::get_tx_max() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

*tx_maxp*

The DbEnv::get_tx_max() method returns the number of active transactions in tx_maxp.

**Class**

DbEnv, DbTxn

**See Also**

Transaction Subsystem and Related Methods (page 664), DbEnv::set_tx_max() (page 669)
DbEnv::get_tx_timestamp()

```cpp
#include <db_cxx.h>

int DbEnv::get_tx_timestamp(time_t *timestamp);
```

The `DbEnv::get_tx_timestamp()` method returns the recovery timestamp. This value can be modified using the `DbEnv::set_tx_timestamp()` (page 671) method.

The `DbEnv::get_tx_timestamp()` method may be called at any time during the life of the application.

The `DbEnv::get_tx_timestamp()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- `timestamp`
  
  The `DbEnv::get_tx_timestamp()` method returns the recovery timestamp in `timestamp`.

**Class**

`DbEnv`, `DbTxn`

**See Also**

- Transaction Subsystem and Related Methods (page 664), `DbEnv::set_tx_timestamp()` (page 671)
DbEnv::set_tx_max()

```cpp
#include <db_cxx.h>

int DbEnv::set_tx_max(u_int32_t max);
```

Configure the Berkeley DB database environment to support at least \texttt{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 multiversion 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 \texttt{DbEnv::txn_begin()} (page 674) will fail (until some active transactions complete). If \texttt{DbEnv::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 \texttt{DbEnv::set_tx_max()} method configures a database environment, not only operations performed using the specified \texttt{DbEnv} handle.

The \texttt{DbEnv::set_tx_max()} method may not be called after the \texttt{DbEnv::open()} (page 286) method is called. If the database environment already exists when \texttt{DbEnv::open()} (page 286) is called, the information specified to \texttt{DbEnv::set_tx_max()} will be ignored.

The \texttt{DbEnv::set_tx_max()} method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

\texttt{max}

The \texttt{max} parameter configures the minimum number of simultaneously active transactions supported by Berkeley DB database environment.

**Errors**

The \texttt{DbEnv::set_tx_max()} method may fail and throw a \texttt{DbException} exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

If the method was called after `DbEnv::open()` (page 286) was called; or if an invalid flag value or parameter was specified.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
DbEnv::set_tx_timestamp()

```cpp
#include <db_cxx.h>

int DbEnv::set_tx_timestamp(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 `DbEnv::set_tx_timestamp()` method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

The `DbEnv::set_tx_timestamp()` method may not be called after the `DbEnv::open()` (page 286) method is called.

The `DbEnv::set_tx_timestamp()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::set_tx_timestamp()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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**

`DbEnv, DbTxn`

**See Also**

Transaction Subsystem and Related Methods (page 664)
The DbEnv::txn_recover() method should only be called after the environment has been recovered.

Multiple threads of control may call DbEnv::txn_recover(), but only one thread of control may resolve each returned transaction, that is, only one thread of control may call DbTxn::commit() (page 686) or DbTxn::abort() (page 685) on each returned transaction. Callers of DbEnv::txn_recover() must call DbTxn::discard() (page 689) to discard each transaction they do not resolve.

On return from DbEnv::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_GID_SIZE];**
  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 DbEnv::txn_recover() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbEnv::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 DbEnv::txn_recover(). Calls to DbEnv::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 DbEnv::txn_recover() left off.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
DbEnv::txn_begin()

```cpp
#include <db_cxx.h>

int DbEnv::txn_begin(DbTxn *parent, DbTxn **tid, u_int32_t flags);
```

The `DbEnv::txn_begin()` method creates a new transaction in the environment and copies a pointer to a `DbTxn` that uniquely identifies it into the memory to which `tid` refers. Calling the `DbTxn::abort()` (page 685), `DbTxn::commit()` (page 686) or `DbTxn::discard()` (page 689) 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 `DbEnv::txn_begin()`, `DbTxn::abort()` (page 685) and `DbTxn::commit()` (page 686) — while it has active child transactions (child transactions that have not yet been committed or aborted).

The `DbEnv::txn_begin()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbTxn::prepare()` (page 694).

**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_BULK**

  Enable transactional bulk insert optimization. When this flag is set, the transaction avoids logging the contents of insertions on newly allocated database pages. In a transaction that inserts a large number of new records, the I/O savings of choosing this option can be significant.

  Users of this option should be aware of several issues. When the optimization is in effect, page allocations that extend the database file are logged as usual; this allows transaction aborts to work correctly, both online and during recovery. At commit time, the database’s pages are flushed to disk, eliminating the need to roll-forward the transaction during normal recovery. However, there are other recovery operations that depend on roll-forward, and care must be taken when DB_TXN_BULK transactions interact with them.

  In particular, DB_TXN_BULK is incompatible with replication, and is simply ignored when replication is enabled. Also, hot backup procedures must follow a particular protocol, introduced in Berkeley DB 11gR2.5.1, which is to set the DB_HOTBACKUP_IN_PROGRESS (page 324) flag in the environment before starting to copy files. It is important to note that incremental hot backups can be invalidated by use of the bulk insert optimization. For more information, see the section on Hot Backup in the Getting Started With Transaction Processing Guide and the description of the flag DB_HOTBACKUP_IN_PROGRESS (page 324) in DB_ENV->set_flags.

  The bulk insert optimization is effective only for top-level transactions. The DB_TXN_BULK flag is ignored when parent is non-null.

- **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 DbEnv::set_flags() (page 323) 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 DbEnv::set_flags() (page 323) 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). Snapshot isolation is not supported with replication.

  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 DbEnv::set_flags() (page 323) 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 DbEnv::set_flags() (page 323) 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 DbEnv::set_flags() (page 323) method. Any value specified to this method overrides that setting.
Errors

The DbEnv::txn_begin() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DbMemoryException or ENOMEM**

The maximum number of concurrent transactions has been reached.

DbMemoryException (page 368) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, ENOMEM is returned.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
DbEnv::txn_checkpoint()

```cpp
#include <db_cxx.h>

int
DbEnv::txn_checkpoint(u_int32_t kbyte, u_int32_t min,
                       u_int32_t flags) const;
```

If there has been any logging activity in the database environment since the last checkpoint, the DbEnv::txn_checkpoint() method flushes the underlying memory pool, writes a checkpoint record to the log, and then flushes the log.

The DbEnv::txn_checkpoint() method returns a non-zero error value on failure and 0 on success.

The DbEnv::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 DbEnv::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 DbEnv::txn_checkpoint() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv, DbTxn
See Also

Transaction Subsystem and Related Methods (page 664)
DbEnv::txn_stat()

```cpp
#include <db_cxx.h>

int
DbEnv::txn_stat(DB_TXN_STAT **statp, u_int32_t flags);
```

The DbEnv::txn_stat() method returns the transaction subsystem statistics.

The DbEnv::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 DbEnv::set_alloc() (page 294) 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:

- **u_int32_t st_inittxns;**
  The initial number of transactions configured.

- **DB_LSN st_last_ckpt;**
  The LSN of the last checkpoint.

- **u_int32_t st_last_txnid;**
  The last transaction ID allocated.

- **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.

- **u_int32_t st_maxtxns;**
  The maximum number of active transactions configured.

- **uintmax_t st_naborts;**
  The number of transactions that have aborted.

- **u_int32_t st_nactive;**
  The number of transactions that are currently active.

- **uintmax_t st_nbegins;**
  The number of transactions that have begun.
• `uintmax_t st_ncommits`;
  The number of transactions that have committed.

• `u_int32_t st_nrestores`;
  The number of transactions that have been restored.

• `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.

• `uintmax_t st_region_nowait`;
  The number of times that a thread of control was able to obtain the transaction region
  mutex without waiting.

• `uintmax_t st_region_wait`;
  The number of times that a thread of control was forced to wait before obtaining the
  transaction region mutex.

• `roff_t st_regsize`;
  The region size, in bytes.

• `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).

• `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_int8_t gid[DB_GID_SIZE]`;
    If the transaction was prepared using `DbTxn::prepare()` (page 694), then `gid` contains the
    transaction's Global ID. Otherwise, `gid's` contents are undefined.

  • `DB_LSN lsn`;
    The log sequence number of the transaction's first log record.

  • `u_int32_t mvcc_ref`;
    The number of buffer copies created by this transaction that remain in cache.

  • `char name[]`;
If a name was specified for the transaction, up to the first 50 bytes of that name, followed by a null termination byte.

- **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.

- **u_int32_t priority;**
  
  This transaction's deadlock resolution priority.

- **DB_LSN read_lsn;**
  
  The log sequence number of reads for snapshot transactions.

- **DB_TXN_ACTIVE_SLICE *slice_txns;**
  
  Either NULL, or a `DbEnv::get_slice_count()` (page 273) sized array. If the entry for a slice is non-zero, then that slice is participating in the transaction, and the value of the entry is the integer transaction ID of the subordinate transaction.

- **u_int32_t status;**
  
  Provides one of the following constants, indicating the transaction status:

  
  TXN_ABORTED
  TXN_COMMITTED
  TXN_NEED_ABORT
  TXN_PREPARED
  TXN_RUNNING

- **db_threadid_t tid;**
  
  The thread of control ID of the originator of the transaction.

- **u_int32_t txnid;**
  
  The transaction ID of the transaction.

- **u_int32_t xa_status;**
  
  Provides one of the following constants, which indicate the XA status:

  
  TXN_XA_ACTIVE
  TXN_XA_DEADLOCKED
  TXN_XA_IDLE
  TXN_XA_PREPARED
The `DbEnv::txn_stat()` method may not be called before the `DbEnv::open()` (page 286) method is called.

The `DbEnv::txn_stat()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 `DbEnv::txn_stat()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbTxn`

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbEnv::txn_stat_print()

```c
#include <db_cxx.h>

int
DbEnv::txn_stat_print(u_int32_t flags);
```

The `DbEnv::txn_stat_print()` method displays the transaction subsystem statistical information, as described for the `DbEnv::txn_stat()` method. The information is printed to a specified output channel (see the `DbEnv::set_msgfile()` method for more information), or passed to an application callback function (see the `DbEnv::set_msgcall()` method for more information).

The `DbEnv::txn_stat_print()` method may not be called before the `DbEnv::open()` method is called.

The `DbEnv::txn_stat_print()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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_ALLOC**
  
  Display allocation information. To display allocation information, both DB_STAT_ALLOC and DB_STAT_ALL need to be set.

- **DB_STAT_CLEAR**
  
  Reset statistics after displaying their values.

**Class**

DbEnv, DbTxn

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbTxn::abort()

#include <db_cxx.h>

int DbTxn::abort();

The DbTxn::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 DbEnv::set_app_dispatch() (page 296). Before DbTxn::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 DbTxn::abort() has been called, regardless of its return, the DbTxn handle may not be
accessed again.

The DbTxn::abort() method either returns a non-zero error value or throws an exception
that encapsulates a non-zero error value on failure, and returns 0 on success.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)


**DbTxn::commit()**

```cpp
#include <db_cxx.h>

int DbTxn::commit(u_int32_t flags);
```

The `DbTxn::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 `DbTxn::commit()` has been called, regardless of its return, the `DbTxn` handle may not be accessed again. If `DbTxn::commit()` encounters an error, the transaction and all child transactions of the transaction are aborted.

The `DbTxn::commit()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success. The errors values that this method returns include the error values of the `Dbc::close()` method and the following:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

**DbDeadlockException (page 365)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_DEADLOCK` is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

**DbLockNotGrantedException (page 366)** is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, `DB_LOCK_NOTGRANTED` is returned.

**DB_REP_LEASE_EXPIRED**

The operation failed because the site's replication master lease has expired.
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 `DbEnv::set_flags()` (page 323) method or for a single transaction using the `DbEnv::txn_begin()` (page 674) 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 `DbEnv::set_flags()` (page 323) method. This behavior may also be set for a single transaction using the `DbEnv::txn_begin()` (page 674) 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 possibility of data loss than does `DB_TXN_NOSYNC`.

  This behavior may be set for a Berkeley DB environment using the `DbEnv::set_flags()` (page 323) method or for a single transaction using the `DbEnv::txn_begin()` (page 674) method. Any value specified to this method overrides both of those settings.

Class

`DbEnv`, `DbTxn`
See Also

Transaction Subsystem and Related Methods (page 664)
The DbTxn::discard() method frees up all the per-process resources associated with the specified DbTxn handle, neither committing nor aborting the transaction. This call may be used only after calls to DbEnv::txn_recover() (page 672) when there are multiple global transaction managers recovering transactions in a single Berkeley DB environment. Any transactions returned by DbEnv::txn_recover() (page 672) that are not handled by the current global transaction manager should be discarded using DbTxn::discard().

All open cursors in the transaction are closed and the first cursor close error, if any, is returned.

The DbTxn::discard() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success. The errors values that this method returns include the error values of Dbc::close() and the following:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

DbDeadlockException (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See Db::set_lk_exclusive() (page 131) for more information.

DbLockNotGrantedException (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**EINVAL**

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

After DbTxn::discard() has been called, regardless of its return, the DbTxn handle may not be accessed again.

**Parameters**

**flags**

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

The `DbTxn::discard()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or 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

`DbEnv`, `DbTxn`

See Also

*Transaction Subsystem and Related Methods (page 664)*
DbTxn::get_name()

```cpp
#include <db_cxx.h>

int DbTxn::get_name(const char **namep);
```

The `DbTxn::get_name()` method returns the string associated with the transaction.

The `DbTxn::get_name()` method may be called at any time during the life of the application.

The `DbTxn::get_name()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**namep**

The `DbTxn::get_name()` method returns a reference to the string associated with the transaction in `namep`.

**Class**

`DbEnv`, `DbTxn`

**See Also**

*Transaction Subsystem and Related Methods (page 664)*
DbTxn::get_priority()

```cpp
#include <db_cxx.h>

int DbTxn::get_priority(u_int32_t *priority);
```

The `DbTxn::get_priority()` method gets the priority value of the specified transaction.

The `DbTxn::get_priority()` method may be called at any time during the life of the transaction.

The `DbTxn::get_priority()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**priority**

Upon return, the `priority` parameter will point to a value between 0 and 2\(^{32}-1\).

**Errors**

The `DbTxn::get_priority()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  An invalid flag value or parameter was specified.

**Class**

`DbEnv`, `DbTxn`

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbTxn::id()

#include <db_cxx.h>

u_int32_t
DbTxn::id();

The DbTxn::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 DbTxn::id() as the locker parameter to the DbEnv::lock_get() (page 398) or DbEnv::lock_vec() (page 412) calls.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
DbTxn::prepare()

```c
#include <db_cxx.h>

int DbTxn::prepare(u_int8_t gid[DB_GID_SIZE]);
```

The `DbTxn::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 `DbTxn::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 `DbTxn::prepare()` method can not be used with sliced environments.

The `DbTxn::prepare()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success. The errors that this method returns include the error values of `Dbc::close()` and the following:

**DbDeadlockException or DB_LOCK_DEADLOCK**

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

`DbDeadlockException` (page 365) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_DEADLOCK is returned.

**DbLockNotGrantedException or 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.

You attempted to open a database handle that is configured for no waiting exclusive locking, but the exclusive lock could not be immediately obtained. See `Db::set_lk_exclusive()` (page 131) for more information.

`DbLockNotGrantedException` (page 366) is thrown if your Berkeley DB API is configured to throw exceptions. Otherwise, DB_LOCK_NOTGRANTED is returned.

**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 DbEnv::txn_recover() (page 672) telling the application which global transactions must be resolved.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
DbTxn::set_name()

```
#include <db_cxx.h>

int
DbTxn::set_name(const char *name);
```

The DbTxn::set_name() method associates the specified string with the transaction. The string is returned by DbEnv::txn_stat() (page 680) and displayed by DbEnv::txn_stat_print() (page 684).

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 DbTxn::set_name() method may be called at any time during the life of the application.

The DbTxn::set_name() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**name**

The name parameter is the string to associate with the transaction.

**Class**

DbEnv, DbTxn

**See Also**

Transaction Subsystem and Related Methods (page 664)
DbTxn::set_priority()

```cpp
#include <db_cxx.h>

int
DbTxn::set_priority(u_int32_t priority);
```

The `DbTxn::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 `DbTxn::set_priority()` method may be called at any time during the life of the transaction.

The `DbTxn::set_priority()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

priority

The `priority` parameter must be a value between 0 and $2^{32}-1$.

**Errors**

The `DbTxn::set_priority()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  An invalid flag value or parameter was specified.

**Class**

`DbEnv, DbTxn`

**See Also**

`Transaction Subsystem and Related Methods (page 664)`
DbTxn::set_timeout()

```cpp
#include <db_cxx.h>

u_int32_t DbTxn::set_timeout(db_timeout_t timeout, u_int32_t flags);
```

The `DbTxn::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. Also, the database environment must enable the locking subsystem before timeout values can be specified. See `DbEnv::set_timeout()` (page 352) for more information.

The `DbTxn::set_timeout()` method configures operations performed on the underlying transaction, not only operations performed using the specified `DbTxn` handle.

The `DbTxn::set_timeout()` method may be called at any time during the life of the application.

The `DbTxn::set_timeout()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 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 DbTxn::set_timeout() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

Class

DbEnv, DbTxn

See Also

Transaction Subsystem and Related Methods (page 664)
Chapter 14. External Files

External file support is designed for efficient storage of large objects. An object is considered to be large if it is more than a third of the size of a page. Without external file support, large objects must be broken up into smaller pieces, and then reassembled and/or disassembled every time the record is read or updated. Berkeley DB external file support avoids this assembly/disassembly process by storing the large object in a special directory set aside for the purpose. The data itself is not kept in the database, nor is it placed into the in-memory cache.

External files can only be stored using the data portion of a key/data pair. They are supported only for Btree, Hash, and Heap databases, and only so long as the database is not configured for duplicate records, or duplicate sorted records. In addition, the Dbt that you use to access the external file data cannot be configured as a partial Dbt if you want to access the data using the external file's streaming interface (introduced below).

Note that if the environment is transactionally-protected, then all access to the external file is also transactionally protected.

External files cannot be used with in-memory-only databases.

In order to use Berkeley DB's external file support, you must set the external file threshold to a non-zero positive value. (The default value is 0, which means that external files are not enabled). You set the external file threshold using either Db::set_ext_file_threshold() (page 705) or DbEnv::set_ext_file_threshold() (page 718).

Once external files are enabled, there are two ways to create an external file record:

- Configure the Dbt used to access the external file data (that is, the Dbt used for the data portion of the record) with the DB_DBT_EXT_FILE (page 209) flag. This causes the data to be stored as an external file regardless of its size, so long as the database otherwise supports external files.

- Alternatively, creating a data item with a size greater than the external file threshold will cause that data item to be stored as an external file. Of course, for this method to work, the external file threshold must be greater than 0.

External files may be accessed in the same way as other Dbt data, so long as the data itself will fit into memory. More likely, you will find it necessary to use the external file streaming API to read and write external file data. You open an external file stream using the Dbc::db_stream() (page 707) method, close it with the DbStream::close() (page 709) method, write to it using the the DbStream::write() (page 713) method, and read it using the DbStream::read() (page 710) method.
## External Files and Related Methods

<table>
<thead>
<tr>
<th>External File Operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dbc::db_stream()</td>
<td>Create an external file stream</td>
</tr>
<tr>
<td>DbStream::close()</td>
<td>Close an external file stream</td>
</tr>
<tr>
<td>DbStream::read()</td>
<td>Read an external file stream</td>
</tr>
<tr>
<td>DbStream::size()</td>
<td>Returns the size of an external file</td>
</tr>
<tr>
<td>DbStream::write()</td>
<td>Write an external file stream</td>
</tr>
</tbody>
</table>

### External File Configuration

<table>
<thead>
<tr>
<th>Db::set_ext_file_dir(), Db::get_ext_file_dir()</th>
<th>Sets/gets the location where external file data is stored</th>
</tr>
</thead>
<tbody>
<tr>
<td>Db::set_ext_file_threshold(), Db::get_ext_file_threshold()</td>
<td>Sets/gets the size when data records are stored as external files</td>
</tr>
<tr>
<td>DbEnv::set_ext_file_dir(), DbEnv::get_ext_file_dir()</td>
<td>Sets/gets the location where external file data is stored</td>
</tr>
<tr>
<td>DbEnv::set_ext_file_threshold(), DbEnv::get_ext_file_threshold()</td>
<td>Sets/gets the default size for the environment when external files are used</td>
</tr>
</tbody>
</table>
Db::get_ext_file_dir()

```cpp
#include <db_cxx.h>

int Db::get_ext_file_dir(const char **dirp);
```

The `Db::get_ext_file_dir()` method returns the directory location where external file data is stored. The default location can be set using `Db::set_ext_file_dir()` (page 704). Use this method only if the database was not opened in an environment. If the database was opened within an encompassing environment, use `DbEnv::get_ext_file_dir()` (page 715) instead.

The `Db::get_ext_file_dir()` method always returns 0 to indicate success.

**Parameters**

**dirp**

The `dirp` parameter references memory into which is copied the path to the external file data directory.

**Class**

DbEnv

**See Also**

External Files and Related Methods (page 701)
Db::get_ext_file_threshold()

```cpp
#include <db_cxx.h>

int Db::get_ext_file_threshold(u_int32_t *bytes);
```

The `Db::get_ext_file_threshold()` method returns the threshold value, in bytes, beyond which data items are stored as external files. This value can be set using `Db::set_ext_file_threshold()` (page 705). A value of 0 indicates that external files are not in use for this database.

The `Db::get_ext_file_threshold()` method always returns 0 to indicate success.

**Parameters**

- **bytes**

  References memory into which is copied the external file threshold value.

**Class**

Db

**See Also**

External Files and Related Methods (page 701)
Db::set_ext_file_dir()

```c
#include <db_cxx.h>

int Db::set_ext_file_dir(const char *dir);
```

The Db::set_ext_file_dir() method sets the directory where external file data is stored. Use this method if the database is not opened within an encompassing environment. If an environment is in use, use DbEnv::set_ext_file_dir() (page 717) instead.

By default, if this method is not called then external file data is placed in a directory local to the current working directory.

Use Db::get_ext_file_dir() (page 702) to identify the current storage location used for external file data.

This method configures operations performed using the specified Db handle, not all operations performed on the underlying database.

This method may not be called after Db::open() (page 75) is called.

Unless otherwise specified, the Db::set_ext_file_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

**dir**

Provides the name of the directory where external file data is to be stored. If an absolute path is not provided, then the directory identified here is relative to the current working directory.

**Errors**

The Db::set_ext_file_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  If the database was opened within a named environment; or if the method was called after Db::open() (page 75) was called; or if an invalid parameter was specified.

**Class**

Db

**See Also**

External Files and Related Methods (page 701)
Db::set_ext_file_threshold()

```c++
#include <db_cxx.h>

int Db::set_ext_file_threshold(u_int32_t bytes, u_int32_t flags);
```

The Db::set_ext_file_threshold() method sets a size which is used to determine when a data item will be stored as an external file. Data items sized less than this threshold are stored as normal data within the database. Data items larger than this size are stored on-disk in a subdirectory set aside for the purpose.

If this threshold value is set to 0, then external files will never be used by the database.

It is illegal to set an external file threshold if any of the following flags were specified for the database: DB_DUP (page 118), and DB_DUPSORT (page 118).

It is also illegal to set an external file threshold if compression is turned on for the database. That is, if Db::set_bt_compress() (page 96) has been called for the database handle.

This method configures the underlying database. The external file threshold is stored in the database at database creation time. Any external file threshold set after creating the database is ignored.

This method may not be called after Db::open() (page 75) is called.

Unless otherwise specified, the Db::set_ext_file_threshold() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **bytes**

  The `bytes` parameter identifies the threshold size, in bytes, beyond which a data item is stored as an external file.

- **flags**

  The `flags` parameter must be set to 0.

**Errors**

The Db::set_ext_file_threshold() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after Db::open() (page 75) was called; or if an invalid flag or parameter was specified; or if compression is turned on for the database; or if the
database is configured with one or more of the following flags: DB_DUP (page 118), and DB_DUPSORT (page 118).

**Class**

Db

**See Also**

External Files and Related Methods (page 701)
Dbc::db_stream()

```cpp
#include <db_cxx.h>

int
Dbc::db_stream(DbStream **dbs, u_int32_t flags);
```

The `Dbc::db_stream()` method points to a key/value pair where the data item is an external file. Use the `flags` parameter to indicate whether the stream is to be opened for reading, or for reading and writing.

Once the stream is opened, you read it using `DbStream::read()` (page 710), and you write to it using `DbStream::write()` (page 713).

Close this stream using `DbStream::close()` (page 709).

If the data item is not an external file, this method returns an error.

Unless otherwise specified, the `Dbc::db_stream()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

Parameters

**dbs**

The `dbs` parameter references memory into which is copied the newly opened stream.

**flags**

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

- **DB_STREAM_READ**
  
  Indicates that the stream is to be opened for read-only access.

- **DB_STREAM_WRITE**
  
  Indicates that the stream is to be opened for read write access. The stream is sync'd to disc when the stream is closed.

- **DB_STREAM_SYNC_WRITE**
  
  Indicates that the stream is to be opened for read and write access. The stream is sync'd to disc after each write, instead of when the stream is closed.

Errors

The `Dbc::db_stream()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

If the data item is not an external file; or if an invalid flag or parameter is specified.

Class

DBC

See Also

External Files and Related Methods (page 701)
DbStream::close()

```cpp
#include <db_cxx.h>

int DbStream::close(u_int32_t flags);
```

The DbStream::close() method flushes any unwritten data to disk, frees allocated resources, and closes the underlying file which contains the external file. You open an external file stream using Dbc::db_stream() (page 707).

Once this method is called, the stream can not be used again even if this method returns an error.

Unless otherwise specified, the DbStream::close() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

flags

The flags parameter must be set to 0.

**Errors**

The DbStream::close() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**EINVAL**

An invalid flag value or parameter was specified.

**Class**

DbStream

**See Also**

External Files and Related Methods (page 701)
# DbStream::read()

```cpp
#include <db_cxx.h>

int
DbStream::read(Dbt *data, db_off_t offset, u_int32_t size,
                u_int32_t flags);
```

The `DbStream::read()` method reads `size` bytes from the external file, starting at `offset`, into the DBT `data`.

The stream is created using `Dbc::db_stream()` (page 707).

Unless otherwise specified, the `DbStream::read()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

## Parameters

**data**

The `data` parameter is the `Dbt` into which you want to place the data read using this method.

Note that the data DBT must be set with one of the following flags, or this method will return an error: `DB_DBT_MALLOC` (page 209), `DB_DBT_REALLOC` (page 209), `DB_DBT_USERMEM` (page 210), or `DB_DBT_APPMALLOC` (page 211).

**offset**

The `offset` parameter indicates the starting position, in bytes, from the beginning of the external file where you want the read to begin.

**size**

The `size` parameter indicates the number of bytes to read.

**flags**

The `flags` parameter must be set to 0.

## Errors

The `DbStream::read()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

**DB_BUFFER_SMALL**

The Dbt provided to this method was not configured appropriately, and so there is not enough space allocated for the bytes you are trying to read.

**EINVAL**

An invalid flag value or parameter was specified; or if the Dbt was configured with `DB_DBT_PARTIAL` (page 210).
Class

DbStream

See Also

External Files and Related Methods (page 701)
DbStream::size()

```cpp
#include <db_cxx.h>

int DbStream::size(Dbt *data, db_off_t *size, u_int32_t flags);
```

The DbStream::size() method returns the size of the external file in bytes.

Unless otherwise specified, the DbStream::size() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **size**
  
The `size` parameter references memory into which the size of the external file is copied.

- **flags**
  
The `flags` parameter must be set to 0.

**Errors**

The DbStream::size() method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  An invalid flag value or parameter was specified.

**Class**

DbStream

**See Also**

External Files and Related Methods (page 701)
DbStream::write()

```cpp
#include <db_cxx.h>

int DbStream::write(Dbt *data, db_off_t offset, u_int32_t flags);
```

The `DbStream::write()` method writes data to an existing external file object in the database. This method writes data contained in the `Dbt` `data` to the external file stream. Data is written into the stream starting at the position indicated by `offset`. The amount of data written is determined by the `size` field in the `Dbt`.

If this method writes data in the middle of the external file, it will overwrite existing data, instead of shifting it. If this method writes data to the end of the external file, the data is appended to the existing external file. You can determine how large an external file is using `DbStream::size()` (page 712).

To open a stream, use `Dbc::db_stream()` (page 707).

Unless otherwise specified, the `DbStream::write()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **data**
  
The `data` parameter is the `Dbt` containing the data to be written to the external file. The amount of data to be written is determined by the `Dbt`'s `size` field.

- **offset**
  
The `offset` parameter identifies the position in the external file where the write operation will begin.

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

    A sync to disk operation is performed on the stream at the end of the write operation. By default, the sync is performed only when the stream is closed using `DbStream::close()` (page 709). Note that this flag can also be specified when the stream is created using `Dbc::db_stream()` (page 707), in which case the sync behavior becomes the default behavior for this stream instance.

**Errors**

The `DbStream::write()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:
EINVAL

An invalid flag value or parameter was specified; or if the stream is read-only; or if the input Dbt was configured with DB_DBT_PARTIAL (page 210).

Class

DbStream

See Also

External Files and Related Methods (page 701)
DbEnv::get_ext_file_dir()

```cpp
#include <db_cxx.h>

int
DbEnv::get_ext_file_dir(const char **dirp);
```

The DbEnv::get_ext_file_dir() method returns the directory location where external file data is stored. The default location can be set using DbEnv::set_ext_file_dir() (page 717). Use this method only if the database was opened in an environment. If the database was not opened within an encompassing environment, use Db::get_ext_file_dir() (page 702) instead.

The DbEnv::get_ext_file_dir() method always returns 0 to indicate success.

### Parameters

- **dirp**

  The `dirp` parameter references memory into which is copied the path to the external file data directory. If DbEnv::set_ext_file_dir() (page 717) has not been called prior to calling this method, or if external files are not supported by this environment, then this parameter is set to NULL.

### Class

DbEnv

### See Also

External Files and Related Methods (page 701)
DbEnv::get_ext_file_threshold()

```cpp
#include <db_cxx.h>

int DbEnv::get_ext_file_threshold(u_int32_t *bytes);
```

The `DbEnv::get_ext_file_threshold()` method returns the threshold value, in bytes, beyond which data items are stored as external files. This value can be set using `DbEnv::set_ext_file_threshold()` (page 718). A value of 0 indicates that external files are not in use by default in this environment, unless `Db::set_ext_file_threshold()` (page 705) is called to set an external file threshold for a specific database.

The `DbEnv::get_ext_file_threshold()` method always returns 0 to indicate success.

**Parameters**

*bytes*

References memory into which is copied the external file threshold value.

**Class**

DbEnv

**See Also**

External Files and Related Methods (page 701)
DbEnv::set_ext_file_dir()

```cpp
#include <db_cxx.h>

int DbEnv::set_ext_file_dir(const char *dir);
```

The DbEnv::set_ext_file_dir() method sets the directory where external file data is stored. Use this method when the database is opened within an encompassing environment. If an environment is not in use, use Db::set_ext_file_dir() (page 704) instead.

By default, if this method is not called then external file data is placed in a subdirectory within the DB's environment.

Once this method has been used, you can call DbEnv::get_ext_file_dir() (page 715) to identify the current storage location used for external file data.

This method configures operations performed using the specified DbEnv handle, not all operations performed on the underlying database environment.

This method may not be called after DbEnv::open() (page 286) is called.

Unless otherwise specified, the DbEnv::set_ext_file_dir() method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **dir**

  Provides the name of the directory where external file data is to be stored. If an absolute path is not provided, then the directory identified here is relative to the environment’s home directory.

**Errors**

The DbEnv::set_ext_file_dir() method may fail and throw a DbException exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**

  If the method was called after Db::open() (page 75) was called; or if an invalid parameter was specified.

**Class**

DbEnv

**See Also**

External Files and Related Methods (page 701)
DbEnv::set_ext_file_threshold()

```
#include <db_cxx.h>

int
DbEnv::set_ext_file_threshold(u_int32_t bytes, u_int32_t flags);
```

The `DbEnv::set_ext_file_threshold()` method sets a default size for the environment which is used to determine when a data item will be stored as an external file. Data items sized less than this threshold are stored as normal data within the database. Data items larger than this size are stored on-disk in a subdirectory set aside for the purpose.

If this threshold value is set to 0, then external file support is turned off by default for databases created in the environment. If this method is never called, then the default external file threshold is 0.

This method only sets the default external file threshold for the environment. The external file threshold can be set for individual databases created within the environment using `Db::set_ext_file_threshold()` (page 705).

This method configures operations performed using the specified `DbEnv` handle, not all operations performed on the underlying database environment.

You may call this method at any time after the `DbEnv` handle has been created.

Unless otherwise specified, the `DbEnv::set_ext_file_threshold()` method either returns a non-zero error value or throws an exception that encapsulates a non-zero error value on failure, and returns 0 on success.

**Parameters**

- **bytes**
  
The `bytes` parameter identifies the threshold size, in bytes, beyond which a data item is stored as an external file.

- **flags**
  
The `flags` parameter must be set to 0.

**Errors**

The `DbEnv::set_ext_file_threshold()` method may fail and throw a `DbException` exception, encapsulating one of the following non-zero errors, or return one of the following non-zero errors:

- **EINVAL**
  
  If an invalid flag or parameter was specified.

**Class**

`DbEnv`
See Also

External Files and Related Methods (page 701)
Appendix A. Berkeley DB Command Line Utilities

The following describes the command line utilities that are available for Berkeley DB.
## Utilities

<table>
<thead>
<tr>
<th>Utility</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db_archive</td>
<td>Archival utility</td>
</tr>
<tr>
<td>db_checkpoint</td>
<td>Transaction checkpoint utility</td>
</tr>
<tr>
<td>db_convert</td>
<td>Database convert utility</td>
</tr>
<tr>
<td>db_deadlock</td>
<td>Deadlock detection utility</td>
</tr>
<tr>
<td>db_dump</td>
<td>Database dump utility</td>
</tr>
<tr>
<td>db_hotbackup</td>
<td>Hot backup utility</td>
</tr>
<tr>
<td>db_load</td>
<td>Database load utility</td>
</tr>
<tr>
<td>db_log_verify</td>
<td>Log verification utility</td>
</tr>
<tr>
<td>db_printlog</td>
<td>Transaction log display utility</td>
</tr>
<tr>
<td>db_recover</td>
<td>Recovery utility</td>
</tr>
<tr>
<td>db_replicate</td>
<td>Replication utility</td>
</tr>
<tr>
<td>db_sql_codegen</td>
<td>SQL schema to Berkeley DB code in C</td>
</tr>
<tr>
<td>dbsql</td>
<td>Command line interface to libdb_sql</td>
</tr>
<tr>
<td>db_stat</td>
<td>Statistics utility</td>
</tr>
<tr>
<td>db_tuner</td>
<td>Suggest a page size for optimal operation in a btree database</td>
</tr>
<tr>
<td>db_upgrade</td>
<td>Database upgrade utility</td>
</tr>
<tr>
<td>db_verify</td>
<td>Verification utility</td>
</tr>
<tr>
<td>sqlite3</td>
<td>Command line tool for wrapper library libsqlite3</td>
</tr>
</tbody>
</table>
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.

**Note**

If the application(s) that use the environment make use of any of the following methods:

- `DbEnv::add_data_dir()` (page 230)
- `DbEnv::set_data_dir()` (page 303)
- `DbEnv::set_lg_dir()` (page 444)
- `DbEnv::set_region_dir()` (page 492)

then in order for this utility to run correctly, you need a DB_CONFIG file which sets the proper paths using the `add_data_dir` (page 774), or `set_lg_dir` (page 798), or `set_region_dir` (page 814) configuration parameters.

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 748) 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 `DbEnv::log_cursor()` (page 425) 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, `DbEnv::log_put()` (page 431) and `DbTxn::abort()` (page 685) 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_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 `DbEnv::open()` (page 286) method.
**db_checkpoint**

```
```

The `db_checkpoint` utility is a daemon process that monitors the database log, and periodically calls `DbEnv::txn_checkpoint()` (page 678) to checkpoint it.

**Note**

This utility cannot be used to checkpoint a database environment which is replicated using the base replication API.

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_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 `DbEnv::open()` (page 286) method.
The `db_convert` utility converts one or more files and the databases they contain to a specified byte order. If the byte order is not specified, databases are converted to the native byte order of the machine.

The options are as follows:

- `-b`
  Convert the files and databases to the big endian order.

- `-h`
  Specify a home directory for the database environment; by default, the current working directory is used.

- `-l`
  Convert the files and databases to the little endian order.

- `-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.

- `-V`
  Write the library version number to the standard output, and exit.

- `-v`
  Run in verbose mode, displaying a message for each successful convert.

It is important to realize that Berkeley DB database conversions are done in place, and so are potentially destructive. This means that if the system crashes during the convert procedure, or if the convert procedure runs out of disk space, the databases may be left in an inconsistent and unrecoverable state.

The `db_convert` 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_convert` should always be given the chance to detach from the environment and exit gracefully. To cause `db_convert` to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The `db_convert` 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 DbEnv::open() (page 286) 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.
  
  ```plaintext
  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_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 `DbEnv::open()` (page 286) method.
**db_dump**

```
db_dump [-klnPrrV] [-b ext_file_dir]
       [-d a | h | r] [-f output] [-h home] [-P password] [-s database]
       [-D bytes] [-v o | v] file

db_dump [-kNpV] [-d a | h | r] [-f output] [-h home] [-v o | v]
       -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` 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:

- **-b**
  Specifies the directory where external file data is stored for the database you are dumping.

- **-d**
  Dump the specified database in a format helpful for debugging the Berkeley DB library routines.

  - **a**
    Display all information. See also the `-D` option.

  - **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.

- **-D**
  Specifies the maximum number of bytes to dump for each key/data item found in the specified database. This option is only valid when `-da` is also specified. This option overrides the value set for the "set_data_len" parameter in your DB_CONFIG file, if any.

- **-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.
Note that this option causes the utility to verify the integrity of the database before performing the database dump. If this verification fails, the utility will exit with error return `DB_VERIFY_BAD` even though the database is successfully dumped. If you are dumping a database known to be corrupt, you can safely ignore a `DB_VERIFY_BAD` error return.

• `-r`

Salvage data from a possibly corrupt file. When used on an uncorrupted database, this option should return equivalent data to a normal dump, but most likely in a different order.

Note that this option causes the utility to verify the integrity of the database before performing the database dump. If this verification fails, the utility will exit with error return `DB_VERIFY_BAD` even though the database is successfully dumped. If you are dumping a database known to be corrupt, you can safely ignore a `DB_VERIFY_BAD` error return.

• `-s`

Specify a single database to dump. If no database is specified, all databases in the database file are dumped.

• `-v`

Verify the specified database before performing a normal dump. Unlike the `-R` or `-r` option, no data is dumped if the database is corrupt. Use this option when you are not sure about the integrity of the database.

• `-o`

Skip the database checks for btree and duplicate sort order and for hashing. See also the `-o` option for `db_verify`.

• `-V`

Perform the standard verification on the database as performed by `db_verify` without options.

• `-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` 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. Note that this utility might return `DB_VERIFY_BAD` if the `-R`, `-r` or `-v` command line options are used. This indicates a corrupt database. However, the dump may still have been successful.

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 `DbEnv::open()` (page 286) method.
**db_hotbackup**

```
  db_hotbackup [-cDEguVv] [-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. Hot backups can also be performed using the `DbEnv::backup()` (page 232) or `DbEnv::dbackup()` (page 239) methods.

The `db_hotbackup` utility performs the following steps:

1. Sets the `DB_HOTBACKUP_IN_PROGRESS` (page 324) flag in the home database environment.

2. If the `-c` option is specified, checkpoint the source home database environment, and remove any unnecessary log files.

3. 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.

4. 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.

5. 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.

6. Perform catastrophic recovery in the target directory for the backup.

7. Remove any unnecessary log files from the target directory for the backup.

8. Reset the `DB_HOTBACKUP_IN_PROGRESS` (page 324) flag in the environment.

The `db_hotbackup` utility does not resolve pending transactions that are in the prepared state. Applications that use `DbTxn::prepare()` (page 694) must specify `DB_RECOVER_FATAL` when opening the environment, and run `DbEnv::txn_recover()` (page 672) 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.
Note

This option cannot be used when backing up environments replicated using the base replication API.

- **-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 `DbEnv::add_data_dir()` (page 230) 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.

- **-F**
Directly copy from the filesystem. This option can CORRUPT the backup if used while the environment is active and the operating system does not support atomic file system reads. This option is known to be safe only on UNIX systems, not Linux or Windows systems.

- **-g**
  Turn on debugging options. In particular this will leave the log files in the backup directory after running recovery.

- **-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. If a relative path is specified, the path is evaluated relative to the home directory.

- **-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. If applications that update the environment are using the transactional bulk insert optimization, this option must be used with special care. For more information, see the section on Hot Backup in the Getting Started With Transaction Processing Guide.

- **-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 DbEnv::open() (page 286) method.
**db_load**

```
    db_load [-nTV] [-b ext_file_dir] [-c name=value]
    [-f file] [-h home] [-P password] [-o ext_file_threshold]
    [-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` utility or as specified by the `-T` option below.

The options are as follows:

- `-b`
  Identifies the directory where external file data is stored. If this option is not specified, then external file data is placed in a subdirectory within the DB's environment. See also the `-o` option.

- `-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 `DbEnv::open()` (page 286) (This means that `db_load` can be used to load data into databases while they are in use by other processes.) If the `DbEnv::open()` 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.

- `-o`
  Identifies the external file threshold in bytes. This threshold determines when a data item will be stored as an external file. Data items sized less than this threshold are stored as
normal data within the database. Data items larger than this size are stored on-disk in a subdirectory set aside for the purpose. Use the \-b command line option to identify where external file data is stored.

- \-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 \texttt{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 730) 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 DbEnv::open() (page 286) method.

Supported Keywords

The following keywords are supported for the -c command-line option to the db_load utility. See the DbEnv::open() (page 286) 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 147) 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


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.

**Note**

If the application(s) that use the environment make use of the `DbEnv::set_lg_dir()` (page 444) method, then in order for this utility to run correctly, you need a DB_CONFIG file which sets the proper paths using the `set_lg_dir` (page 798) configuration parameter.

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 `DbEnv::open()` (page 286) method.
db_printlog

```
db_printlog [-NrV] [-b start-LSN] [-e stop-LSN] [-h home] [-P password]
             [-D bytes]
```

The `db_printlog` utility is a debugging utility that dumps Berkeley DB log files in a human-readable format.

**Note**

If the application(s) that use the environment make use of the `DbEnv::set_lg_dir()` method, then in order for this utility to run correctly, you need a DB_CONFIG file which sets the proper paths using the `set_lg_dir` configuration parameter.

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.

- **-D**
  
  Specifies the maximum number of bytes to display for each key/data item found in the log. This option overrides the “set_data_len” parameter found in your DB_CONFIG file, if any.

- **-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 `DbEnv::open()` (page 286) method.
db_recover

```
```

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.

**Note**

If the application(s) that use the environment make use of any of the following methods:

- `DbEnv::add_data_dir()` (page 230)
- `DbEnv::set_data_dir()` (page 303)
- `DbEnv::set_lg_dir()` (page 444)
- `DbEnv::set_region_dir()` (page 492)

then in order for this utility to run correctly, you need a DB_CONFIG file which sets the proper paths using the `add_data_dir (page 774)`, or `set_lg_dir (page 798)`, or `set_region_dir (page 814)` configuration parameters.

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 `DbEnv::open()` (page 286) method.
**db_replicate**

```
db_replicate [-MVv] [-h home] 
[-L file] [-P password] [-T num_threads] [-t secs]
```

The `db_replicate` utility is a daemon process that provides replication/HA services on a transactional environment. This utility enables you to upgrade an existing Transactional Data Store application to an HA application with minor modifications. For more information on the `db_replicate` utility, see the Running Replication Using the `db_replicate` Utility section in the *Berkeley DB Programmer's Reference Guide*.

**Note**

This utility is not supported for use with the DB SQL APIs.

The options are as follows:

- **-h**
  
  Specify a home directory for the database environment; by default, the current working directory is used.

- **-L**
  
  Log the execution of the `db_replicate` 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_replicate: ### Wed Jun 15 01:23:45 EDT 1995
  ```

  Additionally, events such as site role changes will be noted in the log file. This file will be removed if the `db_replicate` utility exits gracefully.

- **-M**
  
  Start the `db_replicate` utility to be the master site of the replication group. Otherwise, the site will be started as a client replica.

- **-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**
  
  Specify the number of replication message processing threads.

- **-t**
  
  Specify how often (in seconds) the utility will check for program interruption and resend the last log record.
• -V  
  Write the library version number to the standard output, and exit.

• -v  
  Turn on replication verbose messages. These messages will be written to the standard output and will be quite voluminous.

The **db_replicate** 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_replicate** should always be given the chance to detach from the environment and exit gracefully. To cause **db_replicate** to release all environment resources and exit cleanly, send it an interrupt signal (SIGINT).

The **db_replicate** 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_replicate** utility should be started. The application must use the **DB_INIT_REP** (page 287) and **DB_THREAD** (page 290) flags when creating the environment.

The **db_replicate** 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 **DbEnv::open()** (page 286) 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-language 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:

```sql
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 require 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 664).

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[]`
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":

```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

```shell
  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:

```c
#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

```sh
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](#).

**Command Line Features Unique to `dbsql`**

This section describes pre-defined query statements that can be executed from the `dbsql` command line. These queries take the form of:

```
.stat ITEM
```

where `ITEM` is an optional parameter that indicates what statistics to print. If `ITEM` is not specified, then this command prints statistics for the Berkeley DB environment, followed by statistics for all tables and indexes within the database.

If `ITEM` is the name of a table or index, then this command prints statistics for the table or index using the `Db::stat_print()` (page 165) method.

Otherwise, `ITEM` can be one of several keywords. They are:

- `-env:`

  ```
dbsql> .stat :env:
  ```

  Causes this command to print statistics for the Berkeley DB environment using the `DbEnv::stat_print()` (page 360) method.

- `-rep:`

  ```
dbsql> .stat :rep:
  ```

  Causes this command to print a summary of replication statistics.
db_stat

```
db_stat -d file [-fN] [-h home] [-P password] [-s database]
        [-X A] [-P password]
```

The **db_stat** utility displays statistics for Berkeley DB environments.

The options are as follows:

- **-a**
  
  Display allocation information.

- **-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 `DbEnv::lock_stat()` (page 404) method.

- **-d**
  
  Display database statistics for the specified file, as described in the `Db::stat()` (page 157) 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 `DbEnv::log_stat()` (page 437) 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 `DbEnv::memp_stat()` (page 471) 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 `DbEnv::rep_stat()` (page 601) 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 `DbEnv::txn_stat()` (page 680) method.

- **-V**
  - Write the library version number to the standard output, and exit.

- **-X**
  - Display detailed information about the mutex subsystem.
    - **A**
      - Display all information.

- **-x**
  - Display mutex subsystem statistics, as described in the `DbEnv::mutex_stat()` (page 542) 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.

For information on the statistics feature for Berkeley DB SQL interface, see Command Line Features Unique to `dbsql` (page 760).

### 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 `DbEnv::open()` (page 286) method.
**db_tuner**

```
db_tuner [-c cachesize] -d file [-h home] [-s database] [-v]
```

The `db_tuner` utility analyzes the data in a btree database, and suggests a page size that is likely to deliver optimal operation.

**Note**

The `db_tuner` utility assumes that databases are compacted when analysing the data. The analysis is based on a static view of the data and the data access and update patterns are not take into account.

The options are as follows:

- **-c**
  Specify a value of the cachesize, otherwise, the default value will be set.

- **-d**
  Display database statistics for the specified file. 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.

- **-h**
  Specify a home directory for the database environment.

- **-s**
  Display page size recommendation for the specified database contained in the file specified with the -d flag.

- **-v**
  Display verbose information.
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 730) and `db_load` (page 738) 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 `DbEnv::open()` (page 286) method.
The **db_verify** utility verifies the structure of one or more files and the databases they contain. If the `-m` option is specified, it verifies one or more named in-memory databases.

The options are as follows:

- **-b**
  
  Specify the directory where external file data is stored. By default, if using a database environment, the external file data is placed in a subdirectory within the environment, otherwise, it is placed in a directory local to the current working directory.

- **-h**
  
  Specify a home directory for the database environment; by default, the current working directory is used.

- **-m**
  
  Verify the named in-memory databases.

- **-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 172) 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 `DbEnv::open()` (page 286) 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. DB_CONFIG Parameter Reference

The following DB_CONFIG parameters can be used to manage various aspects of your application's database environment.
## DB_CONFIG Parameters

<table>
<thead>
<tr>
<th>Description</th>
<th>Parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adds a directory to be searched for database files</td>
<td>add_data_dir</td>
</tr>
<tr>
<td>Sets the home directory for a database slice</td>
<td>home_dir</td>
</tr>
<tr>
<td>Sets the mutex alignment</td>
<td>mutex_set_align</td>
</tr>
<tr>
<td>Configures the number of additional mutexes to allocate</td>
<td>mutex_set_increment</td>
</tr>
<tr>
<td>Configures the total number of mutexes to allocate</td>
<td>mutex_set_max</td>
</tr>
<tr>
<td>Specifies the number of times the test-and-set mutexes should spin without blocking</td>
<td>mutex_set_tas_spins</td>
</tr>
<tr>
<td>Sets the clock skew ratio</td>
<td>rep_set_clockskew</td>
</tr>
<tr>
<td>Configures the Berkeley DB replication subsystem</td>
<td>rep_set_config</td>
</tr>
<tr>
<td>Sets record transmission throttling</td>
<td>rep_set_limit</td>
</tr>
<tr>
<td>Specifies the total number of sites in a replication group</td>
<td>rep_set_nsites</td>
</tr>
<tr>
<td>Specifies the database environment's priority</td>
<td>rep_set_priority</td>
</tr>
<tr>
<td>Sets a threshold before requesting retransmission of a missing message</td>
<td>rep_set_request</td>
</tr>
<tr>
<td>Specifies a variety of replication timeout values</td>
<td>rep_set_timeout</td>
</tr>
<tr>
<td>Specifies how master and client sites will handle acknowledgment</td>
<td>repmgr_set_ack_policy</td>
</tr>
<tr>
<td>Configure the Replication Manager incoming queue size limit</td>
<td>repmgr_set_incoming_queue_max</td>
</tr>
<tr>
<td>Identifies a Replication Manager host</td>
<td>repmgr_site</td>
</tr>
<tr>
<td>Sets the size of the shared memory buffer pool</td>
<td>set_cachesize</td>
</tr>
<tr>
<td>Sets the maximum size for set_cachesize parameter</td>
<td>set_cache_max</td>
</tr>
<tr>
<td>Sets the directory path to create the access method database files</td>
<td>set_create_dir</td>
</tr>
<tr>
<td>Sets the maximum number of bytes displayed by some utilities</td>
<td>set_data_len</td>
</tr>
<tr>
<td>Configures a database environment</td>
<td>set_flags</td>
</tr>
<tr>
<td>Configures the directory permissions</td>
<td>set_intermediate_dir_mode</td>
</tr>
<tr>
<td>Sets the size of the in-memory log buffer</td>
<td>set_lg_bsize</td>
</tr>
<tr>
<td><strong>DB_CONFIG Parameters</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>----------------------------</td>
<td>---------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>set_lg_dir</code></td>
<td>Sets the path of the directory for logging files</td>
</tr>
<tr>
<td><code>set_lg_filemode</code></td>
<td>Sets the absolute file mode for created log files</td>
</tr>
<tr>
<td><code>set_lg_max</code></td>
<td>Sets the maximum size of a single file in the log</td>
</tr>
<tr>
<td><code>set_lg_regionmax</code></td>
<td>Sets the size of the underlying logging area</td>
</tr>
<tr>
<td><code>set_lk_detect</code></td>
<td>Sets the maximum number of locking entities</td>
</tr>
<tr>
<td><code>set_lk_max_lockers</code></td>
<td>Sets the maximum number of locking entities</td>
</tr>
<tr>
<td><code>set_lk_max_locks</code></td>
<td>Sets the maximum number of locks supported by the Berkeley DB environment</td>
</tr>
<tr>
<td><code>set_lk_max_objects</code></td>
<td>Sets the maximum number of locked objects</td>
</tr>
<tr>
<td><code>set_lk_partitions</code></td>
<td>Sets the number of lock table partitions in the Berkeley DB environment</td>
</tr>
<tr>
<td><code>set_slice_count</code></td>
<td>Sets the number of slices used for a sliced database</td>
</tr>
<tr>
<td><code>slice</code></td>
<td>Sets a configuration option for a database slice</td>
</tr>
<tr>
<td><code>log_set_config</code></td>
<td>Configures the Berkeley DB logging subsystem</td>
</tr>
<tr>
<td><code>set_mp_max_openfd</code></td>
<td>Limits the number of file descriptors the library will open concurrently when flushing dirty pages from the cache</td>
</tr>
<tr>
<td><code>set_mp_max_write</code></td>
<td>Limits the number of sequential write operations</td>
</tr>
<tr>
<td><code>set_mp_mmapsize</code></td>
<td>Sets the maximum file size</td>
</tr>
<tr>
<td><code>set_open_flags</code></td>
<td>Initializes specific subsystems of the Berkeley DB environment</td>
</tr>
<tr>
<td><code>set_region_dir</code></td>
<td>Sets the path of the directory for disk backed region files</td>
</tr>
<tr>
<td><code>set_shm_key</code></td>
<td>Configures the database environment’s base segment ID</td>
</tr>
<tr>
<td><code>set_thread_count</code></td>
<td>Declares an approximate number of threads in the database environment</td>
</tr>
<tr>
<td><code>set_timeout</code></td>
<td>Sets timeout values for locks or transactions</td>
</tr>
<tr>
<td><code>set_tmp_dir</code></td>
<td>Specifies the directory path of temporary files</td>
</tr>
<tr>
<td><code>set_tx_max</code></td>
<td>Configures support of simultaneously active transactions</td>
</tr>
<tr>
<td><code>setVerbose</code></td>
<td>Enables/disables the Berkeley DB message output</td>
</tr>
</tbody>
</table>
add_data_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 75) 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.

The syntax of this parameter in the DB_CONFIG file is a single line with the string add_data_dir, one or more whitespace characters, and the directory name.

For more information, see DbEnv::add_data_dir() (page 230).
**home_dir**

Sets the home directory where a database slice environment will be physically placed. This parameter can only be set using the slice (page 808) parameter.
**mutex_set_align**

Sets the mutex alignment, in bytes. It is sometimes advantageous to align mutexes on specific byte boundaries in order to minimize cache line collisions. This parameter specifies an alignment for mutexes allocated by Berkeley DB.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::mutex_set_align() (page 534).
mutex_set_increment

Configures the number of additional mutexes to allocate. If an application will allocate mutexes for its own use, this parameter is used to add a number of mutexes to the default allocation.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::mutex_set_increment() (page 536).
mutex_set_max

Configures 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. This parameter is used to specify an absolute number of mutexes to allocate.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::mutex_set_max() (page 539).
mutex_set_tas_spins

Specifies the number of times the test-and-set mutexes should spin without blocking. The value defaults to 1 time on uniprocessor systems and to 50 times the number of processors on multiprocessor systems, up to a maximum of 200.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_tas_spins, one or more whitespace characters, and the number of spins.

For more information, see DbEnv::mutex_set_tas_spins() (page 541).
rep_set_clockskew

Sets the clock skew ratio among replication group members based on the fastest and slowest measurements among the group for use with master leases.

The syntax of this parameter in the DB_CONFIG 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
```

Sets the fast_clock to 102 and the slow_clock to 100 if a group of sites has a 2% variance.

For more information, see DbEnv::rep_set_clockskew() (page 577).
rep_set_config

Configures the Berkeley DB replication subsystem.

The syntax of this parameter in the DB_CONFIG file is a single line with the string rep_set_config, one or more whitespace characters, and the method 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 on
```

or

```
rep_set_config DB_REP_CONF_NOWAIT
```

Configures the Berkeley DB replication subsystem such that the method calls that would normally block while clients are in recovery will return errors immediately.

The method parameters are:
- DB_REP_CONF_AUTOINIT
- DB_REP_CONF_BULK
- DB_REP_CONF_DELAYCLIENT
- DB_REP_CONF_INMEM
- DB_REP_CONF_LEASE
- DB_REP_CONF_NOWAIT
- DB_REPMGR_CONF_ELECTIONS
- DB_REPMGR_CONF_PREFMAS_CLIENT
- DB_REPMGR_CONF_PREFMAS_MASTER
- DB_REPMGR_CONF_2SITE_STRICT

For more information, see DbEnv::rep_set_config() (page 579).
**rep_set_limit**

Sets record transmission throttling. This is a bytecount 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` method.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see `DbEnv::rep_set_limit()` (page 583).
rep_set_nsites

Specifies the total number of sites in a replication group. This parameter is ignored for Replication Manager applications.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::rep_set_nsites() (page 585).
rep_set_priority

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 syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::rep_set_priority() (page 587).
**rep_set_request**

Sets a threshold for the minimum and maximum time that a client waits before requesting retransmission of a missing message.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `rep_set_request`, one or more whitespace characters, and the request time specified in two parts: the min and the max. 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.

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.

For more information, see DbEnv::rep_set_request() (page 589).
**rep_set_timeout**

Specifies a variety of replication timeout values.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `rep_set_timeout`, one or more whitespace characters, and the flag 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 as 15 seconds.

The flag value can be any one of the following:

- `DB_REP_ACK_TIMEOUT`
- `DB_REP_CHECKPOINT_DELAY`
- `DB_REP_CONNECTION_RETRY`
- `DB_REP_ELECTION_TIMEOUT`
- `DB_REP_ELECTION_RETRY`
- `DB_REP_FULL_ELECTION_TIMEOUT`
- `DB_REP_HEARTBEAT_MONITOR`
- `DB_REP_HEARTBEAT_SEND`
- `DB_REPLEASE_TIMEOUT`

For more information, see `DbEnv::rep_set_timeout()` (page 591).
repmgr_set_ack_policy

Specifies how master and client sites will handle acknowledgment of replication messages which are necessary for "permanent" records.

The syntax of this parameter in the DB_CONFIG 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
```

Specifies that the master should wait until all replication clients have acknowledged each permanent replication message.

The ack_policy parameters are:

- DB_REPMGR_ACKS_ALL
- DB_REPMGR_ACKS_ALL_AVAILABLE
- DB_REPMGR_ACKS_ALL_PEERS
- DB_REPMGR_ACKS_NONE
- DB_REPMGR_ACKS_ONE
- DB_REPMGR_ACKS_ONE_PEER
- DB_REPMGR_ACKS_QUORUM

For more information, see DbEnv::repmgr_set_ack_policy() (page 618).
repmgr_set_incoming_queue_max

Sets a byte-count limit on the amount of dynamic memory used by the Replication Manager incoming queue.

The syntax of this parameter in the DB_CONFIG file is a single line with the string repmgr_set_incoming_queue_max, one or more whitespace characters, and the limit specified in two parts: the gigabytes and the bytes values. For example:

```
repmgr_set_incoming_queue_max 0 104857600
```

Sets a 100 megabyte limit.

For more information, see `DbEnv::repmgr_set_incoming_queue_max()` (page 620).
repmgr_site

Identifies a Replication Manager site.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `repmgr_site`, one or more whitespace characters, the host and port parameters specified as a string and an integer respectively. This can optionally be followed by one or more space-delimited keywords and `on/off`. For example:

```
repmgr_site example.com 49200 db_local_site on db_legacy off
```

Available keywords are:

- **db_bootstrap_helper**
  
  If turned on, the identified site may be used as a "helper" when the local site is first joining the replication group. Once the local site has been established as a member of the group, this setting is ignored.

- **db_group_creator**
  
  If turned on, this site should create the initial group membership database contents, defining a "group" of just the one site, rather than trying to join an existing group when it starts for the first time. This setting is only used on the local site, and is ignored when configured on a remote site.

- **db_legacy**
  
  If turned on, specifies that the site is already part of an existing group. This setting causes the site to be upgraded from a previous version of Berkeley DB. All sites in the legacy group must specify this setting for themselves (the local site) and for all other sites currently existing in the group. Once the upgrade has been completed, this setting is no longer required.

- **db_local_site**
  
  If turned on, specifies that this site is the local site within the replication group. The application must identify exactly one site as the local site before replication is started.

- **db_repmgr_peer**
  
  If turned on, specifies that the site may be used as a target for "client-to-client" synchronization messages. This setting is ignored if it is turned on for the local site.

For more information, see `DbSite::set_config()` (page 562).
set_cachesize

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 value specified for this parameter is the maximum value that your application will be able to use for your in-memory cache. If your application does not have enough data to fill up the amount of space specified here, then your application will only use the amount of memory required by the data that your application does have.

For the DB, the default cache size is 8MB. You cannot specify a cache size value of less than 100KB.

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.)

It is possible to specify cache sizes 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 size supplied to this parameter will be rounded to the nearest multiple of the region size and may not be larger than the maximum possible cache size configured for your application (use the set_cache_max (page 791) to do this). The ncache parameter is ignored when resizing the cache.

The syntax of this parameter in the DB_CONFIG 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 1
```

Creates a single 2.5GB physical cache.

Note that this parameter is ignored unless it is specified before you initially create your environment, or you re-create your environment after changing it.

For more information, see DbEnv::set_cachesize() (page 482).
set_cache_max

Sets the maximum size that the set_cachesize parameter is allowed to set. 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 set_cachesize parameter. If no value is specified, it defaults to the initial cache size.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_cache_max`, one or more whitespace characters, and the maximum cache size in bytes, specified in two parts: the gigabytes of cache and the additional bytes of cache. For example:

```
set_cache_max 2 524288000
```

Sets the maximum cache size to 2.5GB.

This parameter can be changed with a simple restart of your application; you do not need to re-create your environment for it to be changed.

For more information, see `DbEnv::set_cache_max()` (page 480).
**set_create_dir**

Sets the path of a directory to be used as the location to create the access method database files.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_create_dir`, one or more whitespace characters, and the directory name.

For example:

```c
set_create_dir /b/data2
```

Sets data2 as the location to create the access method database files. When the `Db::open()` (page 75) function is used to create a file, it will be created relative to this path.

For more information, see `DbEnv::set_create_dir()` (page 305).
**set_data_len**

Limits the amount of data displayed when `DbEnv::lock_stat_print()` (page 410) is called with the `DB_STAT_ALL` flag.

If the `db_printlog` (page 746) or `db_dump` (page 730) utility uses a `DB_CONFIG` file with this setting, it sets the the default for the amount of data displayed for each key/data item. This value may be overridden using the `-D` option for both utilities.

The value set here must be greater than 0. The default value is 100.

The syntax of this parameter in the `DB_CONFIG` file is a single line with the string `set_data_len`, one or more whitespace characters, and the directory name.

For example:

```
set_data_len 1048576
```
set_flags

Configures a database environment.

The syntax of the entry in the DB_CONFIG file is a single line with the string set_flags, one or more whitespace characters, the method flag parameter as a string, 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 method flag parameters are as follows:

- **DB_AUTO_COMMIT**  
  Enables/disables to automatically enclose those DB handle operations for which no explicit transaction handle was specified, and which modify databases in the database environment, within a transaction.

- **DB_CDB_ALLDB**  
  Enables/disables Berkeley DB Concurrent Data Store applications to perform locking on an environment-wide basis rather than on a per-database basis.

- **DB_DIRECT_DB**  
  Enables/disables turning off system buffering of Berkeley DB database files to avoid double caching.

- **DB_DSYNC_DB**  
  Enables/disables configuring 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.

- **DB_MULTIVERSION**  
  Enables/disables all databases in the environment from being opened as if DB_MULTIVERSION is passed to the DB->open method. This flag will be ignored for queue databases for which DB_MULTIVERSION is not supported.

- **DB_NOMMAP**  
  Enables/disables Berkeley DB from copying read-only database files into the local cache instead of potentially mapping them into process memory.

- **DB_REGION_INIT**  
  Enables/disables Berkeley DB to 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.

- **DB_TIME_NOTGRANTED**  
  Enables/disables those database calls timing out based on lock or transaction timeout values to 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.
• **DB_TXN_NOSYNC**
  Enables/disables Berkeley DB to not write or synchronously flush the log on transaction commit.

• **DB_TXN_NOWAIT**
  Enables/disables the operation to return DB_LOCK_DEADLOCK if a lock is unavailable for any Berkeley DB operation performed in the context of a transaction.

• **DB_TXN_SNAPSHOT**
  Enables/disables all transactions in the environment to be started as if DB_TXN_SNAPSHOT were passed to the DB_ENV->txn_begin method, and all non-transactional cursors to be opened as if DB_TXN_SNAPSHOT were passed to the DB->cursor method.

• **DB_TXN_WRITE_NOSYNC**
  Enables/disables Berkeley DB to write, but not synchronously flush, the log on transaction commit.

• **DB_YIELDCPU**
  Enables/disables Berkeley DB to yield the processor immediately after each page or mutex acquisition.

For more information, see [DbEnv::set_flags() (page 323)](#).
set_intermediate_dir_mode

Configures the database environment's intermediate directory permissions.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_intermediate_dir_mode, one or more whitespace characters, and 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 rwrxrwx--- 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.

For more information, see DbEnv::set_intermediate_dir_mode() (page 330).
**set_lg_bsize**

Sets the size of the in-memory log buffer, in bytes.

For the DB, when the logging subsystem is configured for on-disk logging, the default size of the in-memory log buffer is approximately 32KB. For the BDB SQL interface, when the logging subsystem is configured for on-disk logging, the default size of the in-memory log buffer is approximately 64KB. Log information is stored in-memory until the storage space fills up or a 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 syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lg_bsize`, one or more whitespace characters, and the log buffer size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see DbEnv::set_lg_bsize() (page 442).
set_lg_dir

Sets the path of the 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.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_lg_dir, one or more whitespace characters, and the directory name.

For more information, see DbEnv::set_lg_dir() (page 444).
**set_lg_filemode**

Sets 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, use the set_lg_filemode parameter to set the mode of created log files to an absolute value.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lg_filemode`, one or more whitespace characters, and the absolute mode of created log files.

For more information, see `DbEnv::set_lg_filemode()` (page 446).
set_lg_max

Sets the maximum size of a single file in the log, in bytes. The value set for this parameter 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.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_lg_max, one or more whitespace characters, and the maximum log file size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see DbEnv::set_lg_max() (page 447).
**set_lg_regionmax**

Sets 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 syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lg_regionmax`, one or more whitespace characters, and the log region size in bytes.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see `DbEnv::get_lg_regionmax()` (page 422).
set_lk_detect

Sets the maximum number of locking entities supported by the Berkeley DB environment. This value is used by Berkeley DB to estimate how much space to allocate for various lock-table data structures. When using the DB, the default value is 2,000 lockers.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_lk_detect, one or more whitespace characters, and the method detect parameter as a string. 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.

For example:

```bash
set_lk_detect DB_LOCK_OLDEST
```

Sets the deadlock detector such that the lock request for the locker ID with the oldest lock is rejected.

The detect parameter values are:

- `DB_LOCK_DEFAULT`
- `DB_LOCK_EXPIRE`
- `DB_LOCK_MAXLOCKS`
- `DB_LOCK_MAXWRITE`
- `DB_LOCK_MINLOCKS`
- `DB_LOCK_MINWRITE`
- `DB_LOCK_OLDEST`
- `DB_LOCK_RANDOM`
- `DB_LOCK_YOUNGEST`

For more information, see `DbEnv::set_lk_detect()` (page 383).
**set_lk_max_lockers**

Sets the maximum number of locking entities supported by the Berkeley DB environment. This value is used by Berkeley DB to estimate how much space to allocate for various lock-table data structures. When using the DB, the default value is 1,000 lockers. When using the BDB SQL interface, the default value is 2,000 lockers.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lk_max_lockers`, one or more whitespace characters, and the number of lockers.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see `DbEnv::set_lk_max_lockers()` (page 385).
set_lk_max_locks

Sets the maximum number of locks supported by the Berkeley DB environment. This value is used to estimate how much space to allocate for various lock-table data structures. When using the DB, the default value is 1000 locks. When using the BDB SQL interface, the default value is 10,000 locks.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_lk_max_locks, one or more whitespace characters, and the number of locks.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see DbEnv::set_lk_max_locks() (page 387).
**set_lk_max_objects**

Sets the maximum number of locked objects supported by the Berkeley DB environment. This value is used to estimate how much space to allocate for various lock-table data structures. When using the DB, the default value is 1000 objects. When using the BDB SQL interface, the default value is 10,000 objects.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lk_max_objects`, one or more whitespace characters, and the number of objects.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see `DbEnv::set_lk_max_objects()` (page 389).
**set_lk_partitions**

Sets 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.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_lk_partitions`, one or more whitespace characters, and the number of partitions.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.

For more information, see `DbEnv::set_lk_partitions()` (page 391).
set_slice_count

Sets the number of slices to use for a sliced database environment. The value provided here must be a positive integer. However, for any performance benefits to be realized through the use of slices, this value should be greater than 1. For best results, set this value equal to the number of available cores.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_slice_count, one or more whitespace characters, and the number of slices that you want to use.

If slice support is not configured for the database at compile time, then this parameter is ignored. In addition to this parameter, Db handles must be open with the DB_SLICED flag in order for the database to be sliced.

If the database environment already exists when this parameter is changed, it is ignored. To change this value after the environment has been created, re-create your environment.
**slice**

Sets a configuration parameter for a specific slice (identified by a positive integer) or for all slices by using the keyword **all**.

The syntax of this parameter in the DB_CONFIG file is a single line with the string **slice**, one or more whitespace characters, the slice number or **all**, one or more whitespace characters, and the parameter you want to set for the slice. Note that slice numbers start at 0. For example:

```
set_slice_count 4
slice 0 home_dir /export/dbenv/slices/slice0
slice 1 home_dir /export/dbenv/slices/slice1
slice 2 home_dir /export/dbenv/slices/slice2
slice 3 home_dir /export/dbenv/slices/slice3
slice all set_cachesize 2 524288000 1
```

If slice support has not been compiled into the database, then this parameter is ignored.
log_set_config

Configures the Berkeley DB logging subsystem.

The syntax of this parameter in the DB_CONFIG file is a single line with the string log_set_config, one or more whitespace characters, method flag parameter as a string, optionally one or more whitespace characters, and the string on or off. If the optional string is omitted, the default is on.

The method flag parameters are:

- DB_LOG_DIRECT
  Turns off system buffering of Berkeley DB log files to avoid double caching.

- DB_LOG_DSYNC
  Configures 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.

For more information, see DbEnv::log_set_config() (page 433).
set_mp_max_openfd

Limits the number of file descriptors the library will open concurrently when flushing dirty pages from the cache.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_max_openfd, one or more whitespace characters, and the number of open file descriptors.

For more information, see DbEnv::get_mp_max_openfd() (page 462).
set_mp_max_write

Limits the number of sequential write operations scheduled by the library when flushing dirty pages from the cache.

The syntax of this parameter in the DB_CONFIG 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.

For more information, see DbEnv::set_mp_max_write() (page 485).
set_mp_mmapsize

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 syntax of this parameter in the DB_CONFIG file is a single line with the string set_mp_mmapsize, one or more whitespace characters, and the size in bytes.

For more information, see DbEnv::set_mp_mmapsize() (page 487).
set_open_flags

Initializes specific subsystems of the Berkeley DB environment.

The syntax of the entry in the DB_CONFIG is a single line with the string set_open_flags, one or more whitespace characters, the method flag parameter as a string, 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_open_flags DB_INIT_REP or set_open_flags DB_INIT_REP on. Because the DB_CONFIG file is read when the database environment is opened, it will silently overrule configuration done before that time.

The method flag parameters are as follows:

- **DB_INIT_REP**
  Enables/disables DB_INIT_REP in the DB_ENV->open method. For example:
  
  ```
  set_open_flags DB_INIT_REP on
  ```

  This enables initializing the replication subsystem. This subsystem should be used whenever an application plans on using replication. This setting overwrites the DB_INIT_REP flag passed from the application's DB_ENV->open method.

- **DB_PRIVATE**
  Enables/disables DB_PRIVATE in the DB_ENV->open method. For example:
  
  ```
  set_open_flags DB_PRIVATE on
  ```

  This enables region memory allocation 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 setting overwrites the DB_PRIVATE flag passed from the application's DB_ENV->open method.

- **DB_THREAD**
  Enables/disables DB_THREAD in the DB_ENV->open method. For example:
  
  ```
  set_open_flags DB_THREAD on
  ```

  This enables the DB_ENV handle returned by the DB_ENV->open method to be free-threaded; that is, concurrently usable by multiple threads in the address space. This setting overwrites the DB_THREAD flag passed from the application's DB_ENV->open method.
**set_region_dir**

Sets the path of the directory to be used as the location of disk-backed region files. Region files created by the Memory Pool subsystem will be created in this directory. If no region directory is specified, region files are created in the environment home directory.

The syntax of this parameter in the DB_CONFIG file is a single line with the string `set_region_dir`, one or more whitespace characters, and the directory name.

For more information, see `DbEnv::set_region_dir()` (page 492).
set_shm_key

Configures the database environment’s base segment ID. 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.

See Shared Memory Regions for more information.

The syntax of the entry in the DB_CONFIG file is a single line with the string `set_shm_key` one or more whitespace characters, and the ID.

For more information, see `DbEnv::set_shm_key()` (page 344).
**set_thread_count**

Declares an approximate number of threads in the database environment.

The syntax of the entry in the DB_CONFIG file is a single line with the string `set_thread_count`, one or more whitespace characters, and the thread count. The DB_CONFIG file is read when the database environment is opened, and hence it silently overrules configuration done before that time.

For more information, see `DbEnv::set_thread_count()` (page 346).
**set_timeout**

Sets timeout values, in microseconds, for locks or transactions in the database environment, the wait time for a process to exit the environment when DB_REGISTER recovery is needed, and how frequently to check for failed processes during mutex waits.

The syntax for setting timeout value for database environment's lock, before recovery is started, and transaction is as follows:

- **DB_SET_LOCK_TIMEOUT**

  Configures the database environment's lock timeout value. The syntax of the entry in the DB_CONFIG file is a single line with the string `set_lock_timeout`, one or more whitespace characters, and the lock timeout value.

- **DB_SET_MUTEX_FAILCHK_TIMEOUT**

  If failchk broadcasting has been configured, then this sets the timeout value on how long a thread will wait for a mutex lock before checking whether `DbEnv::failchk()` (page 248) has marked the mutex as failed. The default is to check once every second. The syntax of the entry in the DB_CONFIG file is a single line with the string `set_mutex_failchk_timeout`, one or more whitespace characters, and the wait timeout value.

  If failchk broadcasting has not been configured, then setting this timeout value results in an error.

- **DB_SET_REG_TIMEOUT**

  Sets the timeout value on how long to wait for processes to exit the environment before recovery is started. The syntax of the entry in the DB_CONFIG file is a single line with the string `set_reg_timeout`, one or more whitespace characters, and the wait timeout value.

- **DB_SET_TXN_TIMEOUT**

  Sets the timeout value for transactions in this database environment. The syntax of the entry in the DB_CONFIG file is a single line with the string `set_txn_timeout`, one or more whitespace characters, and the transaction timeout value.

For more information, see `DbEnv::set_timeout()` (page 352).
set_tmp_dir

Specifies 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.

The syntax of the entry in the DB_CONFIG file with the string set_tmp_dir, one or more whitespace characters, and the directory name.

For more information, see DbEnv::set_tmp_dir() (page 355).
set_tx_max

Configures the Berkeley DB database environment to support at least the minimum number of simultaneously active transactions supported by Berkeley DB database environment. This value bounds the size of the memory allocated for transactions. Child transactions are counted as active until they either commit or abort.

The syntax of this parameter in the DB_CONFIG file is a single line with the string set_tx_max, one or more whitespace characters, and the number of transactions.

For more information, see DbEnv::set_tx_max() (page 669).
set_verbose

Enables/disables specific additional informational and debugging messages in the Berkeley DB message output.

The syntax of the entry in the DB_CONFIG file is a single line with the string set_verbose, one or more whitespace characters, the method flag parameter as a string, 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
```

Enables display of additional information when performing recovery.

The method flag parameters are as follows:

- DB_VERB_DEADLOCK
- DB_VERB_FILEOPS
- DB_VERB_FILEOPS_ALL
- DB_VERB_RECOVERY
- DB_VERB_REGISTER
- DB_VERB_REPLICATION
- DB_VERB_REP_ELECT
- DB_VERB_REP_LEASE
- DB_VERB_REP_MISC
- DB_VERB_REP_MSGS
- DB_VERB_REP_SYNC
- DB_VERB_REP_SYSTEM
- DB_VERB_REPMGR_CONNFAIL
- DB_VERB_REPMGR_MISC
- DB_VERB_WAITSFOR

For more information, see `DbEnv::set_verbose()` (page 357).