The Dxstore Database System

Distributed Systems Software

Edition 0.2

for **Dxstore**, Version 0.2 (Alpha)

This is Edition 0.2 of the *Dxstore Database System Manual*, for **Dxstore** Version 0.2 (Alpha).
Last updated May, 2000

Published by DSS Distributed Systems Software, Inc.

1 Introduction to the Dxstore Database System

The **Dxstore** Database System is a database in the tradition of `dbm`, `ndbm`, and several others (collectively, "`*dbm`"). For brevity, the **Dxstore** Database System is simply called **Dxstore** and most API functions, variables, and constants are prefixed by "`Dx_`".

Dxstore provides a superset of the simple, easy-to-understand application program interface (API) of its predecessors, with some syntactic differences. But that's where the similarity ends, because it is a complete redesign and re-implementation and has greatly extended functionality. It is not a plug-in replacement for `*dbm`, but existing code can easily be modified to use **Dxstore** and it's quite likely that, in the process of doing so, significant bunches of old code can be replaced by a few **Dxstore** function calls.

The design of **Dxstore** concentrates on providing very fast yet flexible access to a huge number of data items and vast amounts of data. Its philosophy generally favours performance over storage requirements. While **Dxstore** is not cavalier in its use of disk storage, disks are cheap and getting cheaper. Time is always expensive.

Why should you use **Dxstore**? If your application is very simple, performance is not an issue, database administration isn't a factor, or you don't expect to encounter any database implementation limits, then there may not be a compelling reason to use **Dxstore** rather than a a big commercial database or a `*dbm` library.

On the other hand, you may find some of **Dxstore**'s most significant features to be invaluable:

- Immense databases striped over a huge number of file systems can be created (RAID level 0). When configured to use 64 bits internally, you will almost certainly reach disk space capacity limits or operating system limits before exceeding **Dxstore**'s capacity; specifically, **Dxstore** can address over 2^64 data items, each one of those data items can be on the order of 2^64 bytes in size, and the data can be spread over 2^32 file systems. This is far greater than the commonly-found terabyte limits. See Section 7.7 [Limits], page 41.
- A **Dxstore** database looks like any another file on the system, so the usual commands can be used to control access to the database, rename or copy it, and so on. Any file system that supports the handful of system calls that **Dxstore** requires can be used. No special hardware or dedicated disk partitions are required and disk space for a database does not need to be preallocated. Installation is simple and there is practically no database administration.
- There's no complicated query language to learn or use. A data item, which can be anything, is stored and accessed using a simple key.

- Flexible ways of storing and retrieving data are provided. New data can be inserted anywhere in a stored value. Any contiguous region of a stored value can be retrieved.
- In some databases, the size of a stored value must be known at the time the data is given to the database and the data must all reside in contiguous memory. With **Dxstore**, data can be streamed into and out of the database; the application does not need to tell **Dxstore** how much data there will be and the data does not need to be in memory all at once.
- Sophisticated data editing functionality is provided. A region of a stored value can be overwritten or replaced with new data. If a stored value represents a table, for example, a single row can be replaced.
- Keys can be segregated into *keyspaces*, allowing small databases to be combined into a single physical database.
- *Frame memory* provides a rock-solid way of preventing memory leakage and allows dynamically allocated memory to truly be freed and released to the operating system. As a bonus, persistent dynamically-allocated memory functionality is provided. The frame memory functions are largely orthogonal to the database functions; they can be used with little or no knowledge of how to use the database.
- Descriptive error codes are made available to the user.
- High performance is paramount. See Section 7.1 [Performance and Benchmarks], page 38. Databases running out of main memory, rather than the file system, can be used.
- **Dxstore** has a small memory footprint.
- A **Dxstore** database can be configured to be portable so that it can be moved to a platform with a different processor architecture.

No query language is provided, per se; a data item (which might be considered an object or BLOB by the user of **Dxstore**) is named by an arbitrary binary string. **Dxstore** is intended to be used by programmers as a core component of a larger system, acting as the data storage and retrieval engine, or in applications where support for complicated queries need not be provided. It can be used by a CGI script, integrated into scripting languages such a **Perl**, **Tcl**, and **PHP**, used by a full-blown database, object store, or storage area network server. **Dxstore** could even be used to implement a user-space file system[1]

Note that a database created by **Dxstore** cannot be used with `dbm`, `ndbm`, or any other database and **Dxstore** cannot directly use a database created by them. It is, however, possible to convert a database created by `dbm`, `ndbm`, `gdbm`, and `Berkeley db` to **Dxstore** and some conversion programs are provided (See Section 7.5 [Conversion], page 41).

[1] This hasn't been attempted, but shouldn't be difficult to do and ought to perform better than **Dxstore** on top of a file system. It would require O/S support for using `mmap(2)` on a disk device. You would lose the advantages of having a volume as a normal file, of course.

Some knowledge of a `*dbm` database will be helpful, but not essential, when reading this document or using **Dxstore**.

This document describes the Alpha release of **Dxstore**. Development of **Dxstore** is ongoing and major extensions and improvements are planned.

2 Copying Conditions

The following are the terms for copying and using the documentation and software for the **Dxstore Database System**.

2.1 Conditions for Using, Copying, and Distributing the Documentation

2.2 Conditions for Copying, Using, and Distributing the Software

- Redistributions in binary form must reproduce, verbatim, the above copyright notice, this list of conditions, and the warranty disclaimer in the documentation and/or other materials provided with the distribution.
- Redistributions in any form must be accompanied by information on how to obtain complete source code for the **Dxstore Database System** and any accompanying software that uses the **Dxstore Database System**. The source code must either be included in the distribution or be available for no more than the cost of distribution plus a nominal fee, and must be freely redistributable under reasonable conditions. For an executable file, complete source code means the source code for all modules it contains. It does not include source code for modules or files that typically accompany the major components of the operating system on which the executable file runs.
- The name of DSS Distributed Systems Software, Inc. may not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.
- The author (DSS Distributed Systems Software, Inc.) makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty. See Chapter 3 [No Warranty], page 6.

To license this software for use under different terms, if you have any questions about the terms of non-commercial use, or to purchase support for this software, please contact:

DSS Distributed Systems Software, Inc.
3253 Georgia St.
Richmond, BC V7E 2R4
Canada

dxstore@dss.bc.ca
http://www.dss.bc.ca/dxstore

3 No Warranty

This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY.

THIS SOFTWARE IS PROVIDED BY DSS DISTRIBUTED SYSTEMS SOFTWARE, INC. AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL DSS DISTRIBUTED SYSTEMS SOFTWARE, INC. OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

4 Overview

This chapter only gives an overview of basic concepts and features. The next chapter discusses the API in detail.

The `README` and `INSTALL` files, which you should have received with the distribution, will also contain important information.

See Section 7.7 [Limits], page 41 for a list of implementation limits.

4.1 Terminology: Keys, Values, Datums, and Items

An application uses **Dxstore** to store *key/value* pairs called *data items* or simply *items*. Keys and values are arbitrary binary strings; the database considers them to simply be a sequence of bytes. Using a null-terminated ASCII string for a key is often convenient. A zero-length key and a key longer than (2^32)-1 bytes are invalid. Each *value* has a *key* associated with it that is used to find the *value*. The `Dx_datum` data structure is used to describe both *keys* and *values*. See Section 5.4 [The Datum], page 18.

4.2 Persistent vs. Volatile Databases

A *persistent database* is "durable"; it continues to exist after **Dxstore** terminates. On the other hand, a *volatile database* and all of its contents vanish after termination. A volatile database may offer better performance than a persistent database, under appropriate conditions, because considerably less file system activity is required. It's possible to create a persistent copy of a volatile database.

A volatile database may be given a name but usually it is *anonymous*; since it cannot be re-opened, there is no need for it to be named.

4.3 Volumes

A database can correspond to a single file or be spread across many files. Each file constituting a database is called a *volume* of the database. Volumes may be used to avoid limitations imposed

by a file system, quotas, device capacities, etc. They may also result in improved performance if the volumes reside on different physical disk drives. The database tries to approximately balance data storage across the volumes. The number of volumes making up a particular database are determined at the time the database is created.

The file corresponding to a volume must be an acceptable argument to the `mmap(2)` system call.

For a persistent database, the same volume order must be given each time a database is reopened. This requirement is checked when a database is re-opened. The volumes can be renamed after they are created as long as the relative order in which they are given to `Dx_open()` doesn't change.

Like a persistent database, a volatile database can consist of a set of volumes. Using this probably won't improve the performance of a volatile database, but it may be used to alleviate file system limitations.

Other than when opening a database, volumes aren't referred to when using the database.

Many databases can be used simultaneously by an application, including mixtures of persistent, volatile, read-only, and multi-volume databases.

On file systems such as Linux's ext2 and FreeBSD's ufs, volumes may have "holes" in them, making them appear to use more disk space than they actually do. A utility program, `dxcp`, is provided to copy a volume and preserve any holes it may have.

4.4 Read-Only Databases

At any point in time, a database can be used by either:

- exactly one process capable of modifying it or
- any number of processes that may only read the database.

By default, a database is locked (using `flock(2)`) for reading and writing when it is opened, and so it cannot be used by any other process. A volatile database cannot be read-only and therefore cannot be shared.

When opened in read-only mode, no files are modified, so write permission is not necessary to use the database and abnormal termination will not corrupt it. Again, the database is locked using `flock(2)`.

4.5 Duplicate Keys

The database allows multiple `items` with the same `key` to be stored. A means by which a particular duplicate can be specified for retrieval, replacement, deletion, and so on is provided. A sequential ordering is imposed on the duplicates and the first key, last key, or the Nth key can be selected.[1]

4.6 Points and Regions

A *point*, which is specified by a `Dx_point` structure, is a particular position within a *value*. Data can be inserted at a point, for example.

A *region*, which is specified by a `Dx_region` structure, is a byte range within a *value*. A region of a `value` can be retrieved or deleted, for example.

4.7 Retrieving

A *key* may be retrieved in one of two ways: as a direct pointer or as a copy. A *value* may be retrieved in a third way, via streaming, which is discussed below.

As described later, a flag is set to indicate whether a direct pointer or a copy has been retrieved and another flag option (`DX_ALWAYS_COPY`) may be given to force copying.

Because it is not write-protected, the data pointed to by a direct pointer can be overwritten by the application. Modifying a key is definitely a bad idea, but at least in this version of **Dxstore**, a value can be modified provided its length is not changed.

[1] The capability for a user-supplied duplicate sorting function is planned.

4.8 Streamed Input and Output

When storing a *value*, a user-specified callback function can be called to provide data. This can be both useful and efficient when it is inconvenient or impossible to know how much data there will be or when the volume of data is huge. It removes the burden and expense of the application having to buffer the data completely before writing it to the database.

When fetching a *value*, a user-specified callback function can be invoked to stream data to the application. This can also increase efficiency, especially in situations where the data is merely being written (e.g., writing it to the network, storing in a *different* database), because the data does not need to be copied any more than is strictly necessary. It also makes it easy to filter the data stream as it is read from the database; for example, data could be decrypted, decompressed, checksummed, etc. without having to have first read the entire value into memory.

4.9 Keyspaces

Keyspaces can be used to help reduce disk space requirements. Because some storage resources are preallocated when a database is created, a **Dxstore** database has a minimum size. If a program or set of cooperating programs use many small databases, space efficiency will be low because of the minimum overhead per database. Combining several physical databases into one can use disk space more efficiently and simplify database backup and administration. Also, if several programs access the same set of databases, programming and database efficiencies might be realized by combining the set of databases into a single one.

Keyspaces create disjoint name spaces for items within a single physical database. By default, all items belong to the same keyspace, so if the user doesn't care to use keyspaces they are almost invisible.

When it is created, an item can be assigned to a particular keyspace; it will be completely inaccessible from any other keyspace, almost as if it was stored in a separate physical database together with all other items in the same keyspace. Items that have identical keys but which are in different keyspaces are totally distinct.

As a simple illustration, suppose you wanted to store all prime numbers less than 10,000,000 in the database so that you could retrieve the Nth prime or find out whether a particular number is the first, second, etc. prime. This could be done by creating a keyspace called `"prime-nth"` and then storing items such as (key=`"2"`, value=`"1"`), (key=`"3"`, value=`"2"`), (key=`"5"`, value=`"3"`), etc. into that keyspace. Another keyspace called `"nth-prime"` might be created, with items (key=`"1"`,

value="2"), (key="2", value="3"), (key="3", value="5). Note that some of the same keys appear in both keyspaces. After selecting the "prime-nth" keyspace, a fetch using a number as the key will indicate whether that number is a prime, and if it is, where it appears in the sequence of primes. If the "nth-prime" keyspace is selected, a fetch using the number N as the key will retrieve the Nth prime.

A user-defined keyspace is named by a null-terminated string and must be created before it can be used. The initial default keyspace is named by either a zero-length string or NULL. Keyspaces reserved for internal use by **Dxstore** begin with a "*" and a user-defined keyspace name cannot begin with that character.

When an item is created, its keyspace is either explicitly specified or a database-wide default keyspace is used.

Since only a single physical database is used for all the keyspaces, any user access granted to the database by the operating system will have access to all keyspaces in the database, so one should carefully consider whether it is appropriate to use keyspaces to combine databases that originally have different file access permissions.

4.10 Caveats

Like most databases in the `*dbm` family, a **Dxstore** database is fragile. If a software crash, hardware crash, signal, or other abnormal termination of the application occurs while an update is in progress, a persistent database may be corrupted. In this event, it will have to be restored from a backup or regenerated from raw data. This is admittedly a serious drawback for some applications, though it's not an uncommon limitation. Also, many applications build a database (so that information can be located quickly) and then use it in read-only mode (e.g., `sendmail`'s alias file, password files, and so on).

Future versions of **Dxstore** will address this problem, but in the meantime judicious use of the `Dx_sync()` function can help in some situations. See Section 5.18 [Synchronization], page 30 Alternatively, the application could modify a copy of the database (i.e., each of its volumes), and, once the application has successfully closed the database, make the modified copy the "current" copy (perhaps by renaming the current copy).

Dxstore is not multi-thread safe.

5 Database Functions

The following database functions are provided to applications by **Dxstore**. Each function is discussed in detail in the following sections.

The file `dx.h`, which must be included by all files that use **Dxstore**, contains declarations for these functions; it is usually found in the '`/usr/local/include`' directory, but its actual location depends on how your system administrator installed **Dxstore**.

The **Dxstore** distribution includes several applications that use the database and demonstrate commonly used functions and features.

```
#include <dx.h>

Dx_rc Dx_get_configuration(Dx *dx, Dx_config *config);
Dx_rc Dx_open(char **volume_paths, Dx_config *config, Dx **dxp);
Dx_rc Dx_close(Dx *dx);

Dx_rc Dx_store(Dx *dx, Dx_datum *key, Dx_datum *value, Dx_access_mode mode);
Dx_rc Dx_storex(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
Dx_rc Dx_fetch(Dx *dx, Dx_datum *key, Dx_datum *value);
Dx_rc Dx_fetchx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);

Dx_rc Dx_delete(Dx *dx, Dx_datum *key);
Dx_rc Dx_deletex(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);

Dx_rc Dx_firstkey(Dx *dx, Dx_datum *key);
Dx_rc Dx_firstkeyx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
Dx_rc Dx_nextkey(Dx *dx, Dx_datum *key);
Dx_rc Dx_nextkeyx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);

Dx_rc Dx_key(Dx_datum *keyp, void *keyval, Dx_len keylen);
Dx_rc Dx_value(Dx_datum *valuep, void *valueval, Dx_len valuelen);
Dx_rc Dx_key_first(Dx_datum *keyp);
Dx_rc Dx_key_next(Dx_datum *keyp);

void Dx_set_point(Dx_point *point, Dx_len start_val);
void Dx_set_point_at_start(Dx_point *point);
void Dx_set_point_from_end(Dx_point *point, Dx_len end_val);
void Dx_set_point_at_end(Dx_point *point);

void Dx_set_region_from_start(Dx_region *region);
void Dx_set_region_start(Dx_region *region, Dx_len start_val);
void Dx_set_region_end(Dx_region *region, Dx_len end_val);
void Dx_set_region_to_end(Dx_region *region);
```

```
void Dx_set_region_from_end(Dx_region *region, Dx_len end_val);
void Dx_set_region_len(Dx_region *region, Dx_len dlen);
void Dx_set_region_all(Dx_region *region);

Dx_rc Dx_create_keyspace(Dx *db, char *ksn);
Dx_rc Dx_set_keyspace(Dx *db, Dx_datum *key, char *ksn);
char *Dx_get_keyspace(Dx *db, Dx_datum *key);
Dx_rc Dx_get_keyspace_list(Dx *db, char ***ksn_vec, void **mem);
Dx_rc Dx_delete_keyspace(Dx *db, char *ksn);

Dx_rc Dx_reorganize(Dx *dx);

Dx_rc Dx_sync(Dx *dx);

Dx_rc Dx_create_persistent_copy(Dx *old_db, char **volume_paths);

Dx_rc Dx_stats_reset(Dx *dx);
Dx_stats_counters *Dx_stats_get(Dx *dx);
```

Some functions take a `mode` argument. This consists of one *operation* logically ORed with zero or more modifier flags. The appropriate operation and modifiers to use with a function depend on the function being called. The default operation and modifiers are selected by the `DX_DEFAULT` mode.

The following sections describe each of the database functions.

5.1 Configuration

```
void Dx_get_configuration(Dx *dx, Dx_config *config);

typedef struct {
  Dx_ui32 flags;
  mode_t mode;
  Dx_ui32 page_size;
  Dx_ui32 max_stripe_npages;
  Dx_endian endian;
  Dx_ui32 *volume_weights;
  Dx_hash (*hashfunc)(Dx_ui8 *str, Dx_len len, Dx_hash init);
} Dx_config;
```

`Dx_get_configuration()` either:

- If `Dx` is `NULL`, initializes a `Dx_config` data structure to default values to prepare for modifying

some of the structure elements prior to passing the structure to `Dx_open()`. If the default values are satisfactory, this function doesn't need to be called.

- If `Dx` is not `NULL`, retrieves the current configuration from the specified database.

The following configuration flags, which can be logically ORed, may be assigned to the `flags` element:

`DX_CONFIG_TRUNCATE_ON_OPEN`
: When opening a database, delete volumes that already exist.

`DX_CONFIG_ERASE_ON_DELETE`
: When deleting keys and values, overwrite the old data with zeroes. This might be used to ensure confidential data is not left behind, for example. Note that this can be relatively expensive for a large datum.

`DX_CONFIG_READ_ONLY`
: Selects the more restrictive read-only mode of operation. It is an error to try to open a database in read-only mode if it does not already exist (therefore a volatile database cannot be opened in read-only mode).

`DX_CONFIG_PERSISTENT`
: The database will continue to exist after it is closed. A volatile (non-persistent) database is destroyed when it is closed.

`DX_CONFIG_MUST_EXIST`
: All volumes must already exist; they must not be created. This is implied by `DX_CONFIG_READ_ONLY` and doesn't need to be given if `DX_CONFIG_READ_ONLY` is flagged. It is an error to try to open a volatile database if this flag is present.

`DX_CONFIG_DEFAULT_FLAGS`
: Selects the default flags (don't truncate, persistent, don't erase on delete, permit updates).

The `mode` element is passed to `open(2)` to set the file mode if database files are created. The default mode, `DX_DEFAULT_OPEN_MODE`, is `0644`.

The `page_size` is the length, in bytes, of one of the major space allocation units within a database. Items that are smaller than a certain proportion of `page_size` (called "direct items") are stored in a block of memory of this size called a "page"; bigger items are called "indirect items" with only the item's meta data stored in the page. If the database already exists, this value is completely ignored. It must be a power of 2 between 4096 and 65536, inclusive, and must be a multiple of the operating system's page size (see `getpagesize(3)`). Setting it to zero selects the default page size, which is probably the best choice.

The maximum number of contiguous pages in a single stripe allocation is `max_stripe_npages`. The default value, `DEFAULT_MAX_STRIPE_NPAGES`, is 16 pages. When very large values are being stored, a value yielding the maximum stripe size should be selected. A value that is too small or too big will lead to inefficiencies. This value may be changed for a particular database on different calls to `Dx_open()`. Setting the field to zero selects the default value. **Dxstore** enforces a maximum stripe size, currently 8MB.

If `volume_weights` is `NULL`, **Dxstore** tries to allocate approximately the same amount of space on each volume. By setting `volume_weights` to point to a vector of weights at the time a database is created, this default behaviour can be changed. The value of `volume_weights[0]` is the allocation weight for the first volume, `volume_weights[1]` is the allocation weight for the second volume, and so on, with one weight given for each volume.

The weights work as follows. Assume that `total` is the sum of the weights. The approximate proportion of space allocated from the first volume will be `volume_weights[0] / total`, for the second volume `volume_weights[1] / total`, and so on. If `volume_weights` is `NULL`, this is equivalent to setting each of the `volume_weights` to one.

After database creation, the `volume_weights` are obtained from the database and are ignored; they cannot be changed.

An error may occur if space cannot be allocated on a volume. **Dxstore** is currently unable to compensate by instead allocating on the remaining volumes that still have free space.

The default hash function can be overridden by setting `hashfunc` to point to another function. Using a poor or slow hash function will likely result in extremely poor database performance. The default hash function is normally very good, but it's conceivable that it might perform poorly on some kinds of keys. If you are going to change the hash function used with a database, you must do so at the time the database is created and then use only that hash function with the database. Its first argument points to the string being hashed, the second argument is the number of bytes to use in the hash, and the third argument is the initial hash value to use (normally zero).

5.1.1 Byte Ordering of Meta Data

The `endian` element determines the byte ordering in which persistent meta data is stored. This configuration value is only meaningful at the time the database is created; the meta data byte ordering for a database is never changed after it is created.

The default, DX_NATIVE_ENDIAN, causes meta data to be stored in the byte order used by the processor running the software. If there are no plans to use a database on a different architecture, use DX_NATIVE_ENDIAN. This will be substantially more efficient, since it avoids needless conversions.

If DX_PORTABLE_ENDIAN is selected, a processor independent data encoding is used, which may or may not require conversions to and from the native byte order.[1]

Either of DX_BIG_ENDIAN and DX_LITTLE_ENDIAN can be chosen to force meta data to be stored in those byte orderings, regardless of the native byte order.[2]

If **Dxstore** is run on a processor architecture with a byte ordering different from the one used to create a particular database, the necessary conversions will be performed automatically. Note that this conversion *is not applied* to keys or values, only to **Dxstore**'s internal data structures; it is the user's responsibility to take care of byte ordering issues for stored keys and values.

5.2 Opening the database

```
Dx_rc Dx_open(char **volume_paths, Dx_config *config, Dx **dx);
```

Dx_open() opens and possibly creates a database and must be called before a database can be used. The volumes that make up the database being opened are given as volume_paths, a vector of ordered strings. The vector is terminated by a NULL pointer or a zero-length string. If the operation is successful, the **Dxstore** argument will point to a new Dx structure that will be used in subsequent operations on the database.

If the database has not been opened before, new volumes will be created and initialized (if it's persistent, one file per volume).

If the database has been opened before, a check is made to try to insure that the correct number of volumes is specified and that they're given in the same order as when the database was created.

To specify a volatile database, a NULL argument can be given for volume_paths, or volume_paths can be given but specify no volumes (the *anonymous* volatile database). This always overrides the DX_CONFIG_PERSISTENT flag in the Dx_config structure. Alternatively, one or more strings can

[1] Currently, DX_PORTABLE_ENDIAN is equivalent to DX_BIG_ENDIAN.

[2] Intel processors are little endian.

be provided by `volume_paths` if the `DX_CONFIG_PERSISTENT` flag in the `Dx_config` structure is zero.

Here's an example that creates or opens a three volume persistent database using the default configuration:

```
Dx *dx;
char *volumes[4];
Dx_rc rc;

volumes[0] = "/fs1/mydb_vol1";
volumes[1] = "/fs2/mydb_vol2";
volumes[2] = "/fs3/mydb_vol3";
volumes[3] = NULL;

rc = Dx_open(volumes, NULL, &dx);
```

This example creates a volatile database using the default configuration:

```
Dx *dx;
Dx_rc rc;

rc = Dx_open(NULL, NULL, &dx);
```

This third example creates a volatile database with a custom configuration:

```
Dx_config config;
Dx *dx;
char *volumes[3];
Dx_rc rc;

volumes[0] = "volatile1";
volumes[1] = "volatile2";
volumes[2] = NULL;

Dx_get_configuration(NULL, &config);
config.flags &= (~DX_CONFIG_PERSISTENT);
config.max_stripe_npages = 24;
rc = Dx_open(volumes, &config, &dx);
```

5.3 Closing the database

```
Dx_rc Dx_close(Dx *dx);
```

`Dx_close()` closes a database, and in the case of a persistent database, may write data to disk and unlock the volumes. A volatile database is destroyed. *Terminating a program without calling this function may corrupt the database, so it is important that it always be called.*

5.4 The Datum

```
typedef struct Dx_datum {
  void *dptr;
  Dx_len dlen;
  Dx_ui32 ind;
  Dx_ui32 count;
  Dx_datum_flags_t flags;
  Dx_keyspace *keyspace;
} Dx_datum;

  DX_DATUM_ALIGN2  = 1,                    /* Evenly divisible by 2 */
  DX_DATUM_ALIGN4  = 2,                    /* Evenly divisible by 4 */
  DX_DATUM_ALIGN8  = 3,                    /* Evenly divisible by 8 */
  DX_DATUM_ALIGN16 = 4,                    /* Evenly divisible by 16 */
  DX_DATUM_ALIGNXX = 5,                    /* Unused */
  DX_DATUM_ALIGNYY = 6,                    /* Unused */
  DX_DATUM_ALIGNZZ = 7,                    /* Unused */
  DX_DATUM_ALIGN_MAX = DX_DATUM_ALIGN16    /* Maximum supported alignment */

void Dx_key(Dx_datum *keyp, void *keyval, Dx_len keylen);
void Dx_value(Dx_datum *valuep, void *valueval, Dx_len valuelen);
```

A `Dx_datum` is used to describe a key or a value. The `dptr` element points to the data and the `dlen` element provides the length, in bytes, of the data.

The `ind` element is used to identify and select a key based on its index (position) in the sequence of duplicate keys. It may be set by the user before a function is called and some functions set its value to describe the result; documentation for a function will fully describe how this element is used.

The `count` element is used by some functions to return the number of duplicates there are for a key. Documentation for a function will fully describe how this element is used.

The `flags` element describes several characteristics of the `Dx_datum`. When a `Dx_datum` is stored, the `flags` indicate its memory alignment requirements and when a `Dx_datum` is retrieved, its alignment requirement is also provided. On some processor architectures, some data types must be stored such that they begin on particular memory boundaries, such as an address that is evenly divisible by four. On other architectures, there is a performance penalty when unaligned data is accessed.[3] Specifying an alignment requirement when a `Dx_datum` is stored lets the database preserve that alignment so that when the `Dx_datum` is retrieved it can be directly accessed without having to make a properly aligned copy of it.

If no particular alignment is necessary, `DX_DATUM_ALIGN0` can be given. For example, ordinarily, character strings need not be specially aligned. The values `DX_DATUM_ALIGN2`, `DX_DATUM_ALIGN4`, `DX_DATUM_ALIGN8`, and `DX_DATUM_ALIGN16` specify an alignment evenly divisible by two, four, eight, and sixteen, respectively. The value `DX_DATUM_ALIGN_MAX` specifies the most restrictive alignment supported (currently DX_DATUM_ALIGN16).

After some functions that retrieve a key or value, the `DX_DATUM_MALLOC` bit may be set in the `Dx_datum`'s `flags` element to indicate that the `Dx_datum` was copied and the application is responsible for releasing it (via `free(3)`). If the bit is not set, the `Dx_datum` was not copied - it must not be modified by the application. Furthermore, the key or value to which it points may not be accessible after the next **Dxstore** function call, so if it's needed it must be copied. The `DX_ALWAYS_COPY` modifier forces retrieved keys and values to be copied.

`Dx_key()` initializes a key datum by setting `dptr` to `keyval` and `dlen` to `keylen`, and initializing other elements of the `Dx_datum` appropriately. Similarly, `Dx_value()` is used to initialize a value datum. It is not necessary to use these functions, but doing so is strongly encouraged because it will help avoid errors caused by improper initialization and will avoid problems that might be introduced if future versions of **Dxstore** change the `Dx_datum` structure.

The `keyspace` field is discussed separately. See Section 4.9 [Keyspaces], page 10. See Section 5.15 [Keyspace Functions], page 28.

5.5 Inserting and replacing items in the database

```
Dx_rc Dx_store(Dx *dx, Dx_datum *key, Dx_opmode mode, Dx_datum *value);
```

[3] On the Intel architecture, words, doublewords, and quadwords do not have to be aligned but misaligned access is slower.

`Dx_store()` is used to save an item in a database. (See Section 5.10 [Storex], page 23.) Its default mode is `(DX_INSERT | DX_ADD_FIRST)`, which adds the item even if the key is already present in the database. If the key is a duplicate, it is made the first in the sequence (`DX_ADD_FIRST`).

Selecting a `mode` of `DX_REPLACE` deletes an existing item and replaces it with the new one. Technically, it creates a new item in the database. Its default modifier is `DX_ADD_FIRST`, which will select the first key in the sequence for replacement. The `DX_ADD_LAST` modifier will select the last key and `DX_ADD_INDEX` will select key number "`ind`" (an element of the key `Dx_datum`).

The `DX_INSERT_UNIQUE` mode inserts the item unless the key is already in the database, in which case nothing is done and an error code is returned. In situations where a key must remain unique but an earlier value shouldn't be replaced by a later one, this can save having to first check if the key exists before doing the store operation.

Unlike `dbm` and `ndbm`, the maximum `Dx_datum` size that can be stored is extremely large (See Section 7.7 [Limits], page 41).

```
Dx_key(&key, "Hello", 6);
Dx_value(&value, "World", 6);
rc = Dx_store(dx, &key, DX_INSERT, &value);

Dx_key(&key, "Hello", 6);
key.ind = 2;
Dx_value(&value, "Goodbye", 8);
rc = Dx_store(dx, &key, DX_INSERT, &value);
```

5.6 Retrieving an item from the database

```
Dx_rc Dx_fetch(Dx *dx, Dx_datum *key, Dx_datum *value);
```

`Dx_fetch()` is a simplified interface to retrieve a value given its key. See Section 5.11 [Fetchx], page 25.

The retrieved data may or may not be a copy. The retrieved value datum's `flags` has the `DX_DATUM_MALLOC` bit set whenever the value has been copied and the caller is responsible for releasing the memory using `free(3)`. If the data has not been copied, the pointer is only guaranteed to be valid until just before the next **Dxstore** function is called.

5.7 Specifying Insertion Points

```
/* Set the point to byte offset START_VAL. */
void Dx_set_point(Dx_point *point, Dx_len start_val);

/* Set the point at the start of the value (byte offset zero). */
void Dx_set_point_at_start(Dx_point *point);

/* Set the point to -END_VAL bytes from the end. */
void Dx_set_point_from_end(Dx_point *point, Dx_len end_val);

/* Set the point to byte following the last byte of the value. */
void Dx_set_point_at_end(Dx_point *point);
```

These functions are convenient for setting a `Dx_point`. A `Dx_point` argument can be passed to `Dx_storex()` to specify where, within an existing value, new data is to be inserted or overwritten.

A point may line beyond the end of a value, creating a hole which will be zero-filled.

5.8 Specifying Regions for Fetching and Deletion

```
/* Set the start point of the region to the start of the value. */
void Dx_set_region_from_start(Dx_region *region);

/* Set the start point of the region to START_VAL. */
void Dx_set_region_start(Dx_region *region, Dx_len start_val);

/* Set the end point of the region to offset END_VAL (relative to 0). */
void Dx_set_region_end(Dx_region *region, Dx_len end_val);

/* Set the end point of the region to the end of the value. */
void Dx_set_region_to_end(Dx_region *region);

/* Set the end point to -END_VAL bytes from the end.*/
void Dx_set_region_from_end(Dx_region *region, Dx_len end_val);

/* Set the end point to DLEN bytes from the start point. */
void Dx_set_region_len(Dx_region *region, Dx_len dlen);

/* Set the region to the entire value. */
void Dx_set_region_all(Dx_region *region);
```

These functions are convenient for setting a `Dx_region`. They are used in pairs (except for `Dx_set_region_all()`): one to set the start of the region and the other to set its end point. A `Dx_region` argument can be passed to `Dx_fetchx()` to specify an area within a value to retrieve or to `Dx_storex()` to specify an area within a value to be deleted.

To be valid, a region cannot be outside the boundary of a value. A zero-length region is valid.

5.9 Streamed I/O

```
typedef struct Dx_desc {
  Dx_len dlen;
  Dx_datum_flags flags;
  int error_occurred;
  int last_errno;
} Dx_desc;

typedef struct Dx_user_io {
  Dx_len (*func)(void *arg, void *dptr, Dx_len io_dlen, Dx_desc *desc);
  void *arg;
} Dx_user_io;
```

The `Dx_storex()` and `Dx_fetchx()` functions allow a value to be provided by a user-specified callback function (a "get-function" that `"gets"` data for **Dxstore**) when storing or passed to a user-specified callback function (a "put-function" that `"puts"` data from **Dxstore**) when fetching, respectively. This eliminates the need to have the entire value in memory before storing it or having the entire value in memory after fetching it.

In either case, the memory pointed to by `dptr` should be considered valid only within the context of the callback function to which that address was passed.

The `Dx_user_io` structure specifies the callback function and an `arg` that is passed as the first argument to it; the `arg` is purely for use by the callback function. The second argument is a pointer to a buffer to be filled (when storing) or emptied (when fetching). The third argument is the amount of data, in bytes, requested to be copied into the buffer (when data is being stored in the database) or emptied from it (when data is being fetched from the database). The fourth argument, if it is not `NULL`, can be used by the callback function to indicate that an error has occurred. The callback function is expected to return the actual number of bytes copied into the buffer (when storing) or emptied from the buffer (when fetching).

When storing a value, the `io_dlen` argument specifies the maximum number of bytes that may be copied into the buffer by the callback function. If the value returned by the callback function is less than `io_dlen`, then end-of-file has been reached or an error has occurred. If the `Dx_desc *` argument is non-`NULL`, the caller may optionally set the `dlen` field to the total size of the value and the `flags` field to the required alignment. The given size is taken as an efficiency hint and needn't be the actual size, but the closer it is to the actual size, the more efficient the operation will be. If it is much different than the actual size, the operation may be less efficient than it would otherwise be. If an error occurs within the callback function, it may set `error_occurred` to any non-zero value and `last_errno` to an error code; both of those fields are already zero when the callback function is called. The error code is made available via `Dx_last_errno()`.

When fetching a value, the `io_dlen` argument specifies the number of bytes that the callback function *must* remove from the buffer. The callback function should return `io_dlen`, unless an error occurred. The value of `io_dlen` may be different on different callback function invocations, but will never exceed the configured `page_size` for the database multiplied by the largest `max_stripe_npages` value configured for it. If the `Dx_desc *` argument is non-`NULL`, it indicates to the callback function the total size of the region being fetched and its alignment requirement. The callback function flags an error condition in the same manner as described above.

A callback function must not invoke any operations on the database it is providing input or output to.

5.10 Extended store operation

```
Dx_rc Dx_storex(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
```

`Dx_storex()` provides extended functionality for storing an item, including:

- deletion of a region
- insertion of new data
- overwriting old data with new data
- streamed input

`Dx_storex()` takes a variable number of arguments (either one or two) of various types, depending on the `mode` argument. The `mode` is an operation logically ORed with zero or more modifier flags. See Section 5.5 [Store], page 19. Here is the syntax for the arguments that follow the `mode`:

```
[Dx_point * | Dx_region * | ], [Dx_user_io * | Dx_datum *]
```

Immediately following the `mode` is a `Dx_region *` (if the operation is `DX_EDIT`), `Dx_point *` (if the operation is `DX_STORE`), or this argument is omitted. Following that is either a `Dx_user_io *` or `Dx_datum *`; if the `DX_UIO` flag is present, a `Dx_user_io *` argument is expected, otherwise a `Dx_datum *` argument is expected.

The `mode` determines the expected syntax of the next argument, if any, and selects one of the following operations:

`DX_STORE` A `Dx_point *` argument is expected and specifies an insertion point for the data. If it is `NULL`, new data is appended.

`DX_DEFAULT`
This is equivalent to `DX_STORE`.

`DX_EDIT` This operation is used to delete a region or insert new data, or both. A `Dx_region *` argument is expected and specifies a region to delete before (optionally) inserting new data after the last deleted byte. If it is `NULL`, there is no region to delete.

`DX_INSERT`
This is equivalent to `Dx_store()` with a mode of `DX_INSERT` except that streamed input is allowed. No `Dx_point *` or `Dx_region *` argument is passed.

`DX_INSERT_UNIQUE`
This is equivalent to `Dx_store()` with a mode of `DX_INSERT_UNIQUE` except that streamed input is allowed. No `Dx_point *` or `Dx_region *` argument is passed.

`DX_REPLACE`
This is equivalent to `Dx_store()` with a mode of `DX_REPLACE` except that streamed input is allowed. No `Dx_point *` or `Dx_region *` argument is passed.

The operations listed above may be logically ORed with the following modifier flags, but only in certain combinations:

`DX_OVERWRITE`
Rather than inserting new data, after any deletion is done existing data is overwritten by new data. The new data may go "past the end" of the existing data, making the resulting datum larger. Overwriting may be more efficient than deleting and inserting. This is not allowed with `DX_INSERT`, `DX_INSERT_UNIQUE`, or `DX_REPLACE`.

`DX_MAY_CREATE`
For the `DX_STORE` and `DX_EDIT` operations, the item must ordinarily already exist. This flag allows it to be created if it does not exist. If it needs to be created, it is inserted (`DX_INSERT | DX_ADD_FIRST`) with no alignment specified (`DX_DATUM_ALIGN0`) for the

value. You might be tempted to use this to create an item with a non-zero insertion point, but it's probably best not to do it this way; instead, create a zero-length item with the desired characteristics and then do a `DX_EDIT` function on it.

`DX_ADD_FIRST`
: See Section 5.5 [Store], page 19

`DX_ADD_LAST`
: See Section 5.5 [Store], page 19

`DX_ADD_INDEX`
: See Section 5.5 [Store], page 19

5.11 Extended fetch operation

```
Dx_rc Dx_fetchx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
```

`Dx_fetchx()` provides extended functionality for fetching an item, including:

- fetching of a region
- streamed output
- retrieving information about a value
- checking for the existence of an item

`Dx_fetchx()` takes a variable number of arguments (either one or two) of various types, depending on the `mode` argument. The `mode` is an operation logically ORed with zero or more modifier flags. Here is the syntax for the arguments that follow the `mode`:

```
[Dx_region * | ], [Dx_user_io * | Dx_datum *]
```

Immediately following the `mode` is a `Dx_region *` or this argument is omitted. Following that is either a `Dx_user_io *` or `Dx_datum *`. If the `DX_UIO` flag is present, a `Dx_user_io *` argument is expected, otherwise a `Dx_datum *` argument is expected.

The key's `ind` field selects the duplicate of interest.

The `mode` determines the expected syntax of the next argument, if any, and selects one of the following operations:

`DX_FETCH` The value, or a region of it, is retrieved.

`DX_PROBE` If a region is specified, it is checked for validity. If an item exists (and if a region was specified, it also exists), meta data for the key and value are returned. The `flags` field for the key and value are set and the length of the value (`dlen`) is set. The key's `count` field is set to the number of items with this key.

`DX_EXISTS`
This is similar to `DX_PROBE` except no meta data is returned.

`DX_DEFAULT`
This is equivalent to `DX_FETCH`.

`DX_REGION`
Indicates that a `Dx_region *` argument is given to specify the portion of the value to retrieve or check for existence. A region value of `NULL` selects the entire value.

`DX_ALWAYS_COPY`
This flag forces a retrieved `Dx_datum` to be copied into memory obtained by `malloc(3)`. Values that are "large" (the exact size depends on the database configuration) are always copied. The retrieved value datum's `flags` has the `DX_DATUM_MALLOC` bit set whenever the value has been copied and the caller is responsible for releasing the memory using `free(3)`. If the data has not been copied, the pointer is only guaranteed to be valid until just before the next **Dxstore** function is called.

5.12 Deleting records from the database

```
Dx_rc Dx_delete(Dx *dx, Dx_datum *key);
```

To remove an item from the database, you can use `Dx_delete()`.

5.13 Extended delete operation

```
Dx_rc Dx_deletex(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
```

This is a convenient shorthand for doing an edit without an insertion. If `region` is the area being deleted, it is approximately equivalent to the following:

```
Dx_storex(db, key, DX_EDIT, &region, NULL)
```

Note that explicitly specifying a region that covers the entire value leaves a zero-length item; the item still exists.

The item is removed from the database by either of the following statements, which are equivalent to `Dx_delete()`:

```
Dx_deletex(db, key, DX_DELETE);
Dx_deletex(db, key, DX_DELETE | DX_REGION, NULL);
```

5.14 Iterating through items in the database

```
Dx_rc Dx_firstkey(Dx *dx, Dx_datum *key);
Dx_rc Dx_nextkey(Dx *dx, Dx_datum *key);

Dx_rc Dx_firstkeyx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
Dx_rc Dx_nextkeyx(Dx *dx, Dx_datum *key, Dx_opmode mode, ...);
```

`Dx_firstkey()` and `Dx_nextkey()` are used to iterate through each key in the database. `Dx_firstkey()` is called first, and then each successive call to `Dx_nextkey()` returns another key until the error code `DX_KEY_NOT_FOUND` is returned. A direct pointer to the key or a copy of the key may be returned (See Section 5.4 [The Datum], page 18).

`Dx_key()` should be called before `Dx_firstkey()` or `Dx_firstkeyx()` to initialize `key`.[4].

The order in which keys are returned is apparently random; it depends on hash values, database configuration, and database state variables. The `dlen` field will be set to the length of the key, the `ind` field identifies the key amongst duplicates, the `flags` field gives the key's alignment and whether it was copied (`DX_DATUM_MALLOC`).

`Dx_firstkeyx()` and `Dx_nextkeyx()` iterate through keys in a similar way, except they include the functionality of `Dx_fetchx()`. The `mode` and the arguments that follow are the same as for `Dx_fetchx()`. See Section 5.11 [Fetchx], page 25.

`Dx_firstkey()` is equivalent to this statement:

```
Dx_firstkeyx(db, &key, DX_PROBE, &value)
```

[4] Note that after initializing the key, `Dx_set_keyspace()` can be called to iterate through a particular keyspace. See Section 5.15 [Keyspace Functions], page 28

These functions are intended to visit the database in read-only algorithms; adding or deleting items while iterating through a database may return the same key twice or exhibit other peculiar behaviour. Items may be edited, however.

5.15 Creating logical databases

```
Dx_rc Dx_create_keyspace(Dx *db, char *ksn);
Dx_rc Dx_set_keyspace(Dx *db, Dx_datum *key, char *ksn);
char *Dx_get_keyspace(Dx *db, Dx_datum *key);
Dx_rc Dx_get_keyspace_list(Dx *db, char ***ksn_vec, void **mem);
Dx_rc Dx_delete_keyspace(Dx *db, char *ksn);
```

The initial default keyspace, which is named by a zero-length string or `NULL`, is used unless the application either changes the default keyspace or selects a specific keyspace when storing an item. The default keyspace can be reset to the initial default keyspace or explicitly selected for a key.

A user-defined keyspace must be created using `Dx_create_keyspace()` before it can be used. A user-defined keyspace name must be unique (both textually and in its hash value) and may not begin with the "`*`" character.

The default keyspace can be set to a user-defined keyspace (after it has been created) at any time using `Dx_set_keyspace()` with `NULL` as the `key` argument. Passing a pointer to a key sets the keyspace for that key only; the default keyspace is unchanged.

If a `key` is used in any store function with `key.keyspace` non-`NULL`, the new item is assigned to that keyspace. If `key.keyspace` is `NULL`, then the default keyspace is used.

`Dx_get_keyspace()` returns the name of the keyspace for `key` or the default keyspace (if `key` is `NULL`).

`Dx_get_keyspace_list()` returns a list of all created keyspace names (`ksn_vec[0]`, `ksn_vec[1]`, etc.). The memory used to hold the keyspace names is pointed to by `mem` and it should be freed using `free(3)`. The vector of pointers, `ksn_vec`, should likewise be freed.

`Dx_delete_keyspace()` is not implemented.[5]

[5] There is some question about whether the user or **Dxstore** should be responsible for checking that a keyspace is empty before deleting the name.

An item can only be accessed through the appropriate keyspace setting in a key or the default keyspace. This includes the iteration functions. See Section 5.14 [Iteration], page 27. The only way to change an item's keyspace is by fetching it and storing it in a different keyspace.

If there are a huge number of items in a database but only a few of them are in a particular keyspace, iterating through the small keyspace will be relatively slow.[6]

5.16 Reorganization

```
Dx_rc Dx_reorganize(Dx *dx);
```

Dxstore sometimes postpones making certain changes to the database until it has to. This can slightly reduce performance for some items, but it also tends to reduce the size of the database. Calling `Dx_reorganize()` forces some of these changes to occur immediately.

It's never necessary to call this function; it may be removed or made more useful in a future release.

5.17 Instrumentation

```
typedef struct Dx_stats_counters {
  int nsplits;
  int nadds;
  int nmoves;
  int nsearches;
  int nprobes;
  int nreorgs;
} Dx_stats_counters;

Dx_rc Dx_stats_reset(Dx *dx);
Dx_stats_counters *Dx_stats_get(Dx *dx);
```

Some simple counters are associated with each database. They are incremented to count events as database functions execute. At present, they are primary intended for debugging purposes and aren't generally very useful, but are included here for completeness.

[6] Currently, a brute force search is required. If this turns out to be a problem, a different internal organization will be investigated.

Calling `Dx_stats_reset()` resets them and `Dx_stats_get()` retrieves them.

5.18 Synchronization

```
Dx_rc Dx_sync(Dx *dx);
```

Modifications made to a persistent database may not be written to its volumes until the database is closed. If the application terminates before `Dx_close()` is called successfully, it is possible for the database to become corrupted and unusable.

Forcing all modifications to be written synchronously, however, would hurt performance and would still not guarantee that volumes always had correct data (unless the modifications were made atomically).

The `Dx_sync()` function forces all modifications made to the database but not yet written to the volumes to be immediately flushed to disk. When used wisely, this may help lower the probability of database corruption without negatively impacting performance too severely. It is still not perfect, however, because disk drives may cache data internally.

5.19 Making a Persistent Copy

```
Dx_rc Dx_create_persistent_copy(Dx *old_db, char **volume_paths);
```

At any time, any volatile database (`old_db`) can be converted into a new, identical persistent copy using `Dx_create_persistent_copy()`. This might be done to periodically checkpoint a volatile database or to improve performance by avoiding file system overhead until just before the application exits.

The `volume_paths` argument is as for `Dx_open()`. There must be the same number of volumes given in `volume_paths` as are in `old_db`. Each of the files named in `volume_paths` is truncated if it already exists. The configuration of the persistent copy is almost identical to that of `old_db` (e.g., it will have the same file creation `mode`, `page_size`, `endian`, etc.).

5.20 Errors

```
char *Dx_strerror(Dx_rc rc);
int Dx_last_errno(Dx *db);
```

Most functions return a `Dx_rc` result code. All user-visible result codes are prefixed by "`Dx_`".

In addition, the most recent operating system error code (`errno(3)`) is retained and initialized to zero at the start of every function.

The error code `DX_OK` indicates that a function was successful.

To convert a `Dx_rc` error code into English text, use `Dx_strerror()`.

The most recent `errno` operating system error code is available through `Dx_last_errno()`. In particular, if a function returns `DX_OS_ERROR`, `Dx_last_errno()` should be called to find out what happened. All **Dxstore** function calls that return a `Dx_rc` reset the operating system error code.

The frame memory functions make error codes available in a different way. See Section 6.1 [Frame Manipulation], page 32.

If a call to `malloc(3)` fails, **Dxstore** will print a message and call `abort(3)`.

The `SIGSEGV` signal is caught and **Dxstore** will call `abort(3)`.

5.21 Setting options

There is currently no mechanism for changing the database configuration after it has been opened. A future release may support toggling between read-only and read-write mode, and so on.

5.22 Variables and Symbols

```
extern const char *Dx_version;
```

The global variable `Dx_version` is a printable string that provides the version identification for your **Dxstore**, including whether it is a 32-bit or 64-bit configuration.

6 Frame Memory Management

At its simplest, frame memory lets a programmer make a set of memory allocations and later free them all at once, without having to pass the pointers that were returned by each of those allocations.

A *frame* is a "container" for zero or more allocations of memory. There is an implicit stack of frames; many frame functions operate implicitly on the top of this stack. A frame can also be popped off the stack and later pushed back on. The unit for freeing memory allocations is the frame; an individual allocation cannot be freed.

Frame memory has two main features:

- It allows dynamic memory allocations to be grouped together and freed in one operation. This makes it easy to prevent memory leakage (for example, in server processes) and simplifies error handling code.
- It allows dynamic memory allocations to be made using a volatile database or the standard system `malloc(3)` mechanism. The former lets freed memory be returned to the operating system, something that is not done by most implementations of `malloc(3)`.

You cannot use memory allocated outside of the frame memory package with memory allocated within it, and vice versa. For example, do not try to call `Dx_realloc()` with a `ptr` argument that was returned by `malloc(3)`.

Unlike the **Dxstore** database functions, most frame memory functions do not return a `Dx_rc` result code; consistency was sacrificed so that the frame memory functions could be used more idiomatically. When an error occurs, the global variable `Dx_last_rc` is set appropriately (otherwise it is reset to `DX_OK`). Functions of type `int` return `-1` to flag an error and functions that return a pointer return `NULL` to indicate that an error occurred.

6.1 Frame Manipulation

```
int Dx_new_frame(Dx *db, char *name);
void *Dx_pop_frame(void);
int Dx_push_frame(void *ptr);
void Dx_exch_frame(void);
int Dx_unite_frames(void *frame1, void *frame2);
```

```
extern Dx_rc Dx_last_rc;
```

Initially, the frame stack is empty. Some functions will create a frame on an empty stack if necessary, but functions are provided to explicitly manipulate the frame stack.

`Dx_new_frame()` creates a new frame and pushes it onto the stack. The `db` argument selects the default source of memory for allocations associated with the new frame: if `db` is `NULL`, `malloc(3)` will be the source, otherwise the given database will be the source. The `name` argument is currently only useful when debugging and may be `NULL` or a zero-length string.

`Dx_pop_frame()` removes the frame from the top of the stack and returns it. `NULL` is returned if the stack is empty.

`Dx_push_frame()` puts a previously popped frame back on top of the stack.

`Dx_exch_frame()` exchanges the frame on top of the stack with the one directly underneath it. If there aren't at least two frames on the stack, nothing is done.

`Dx_unite_frames()` combines `frame1` with allocations on `frame2`, leaving `frame1` and destroying `frame2`. If either is `NULL`, it refers to the top frame.

6.2 Frame Memory Allocation

```
void *Dx_malloc(size_t size);
void *Dx_malloc_src(Dx *db, size_t size);
void *Dx_frame_malloc(void *ptr, size_t size);
void *Dx_frame_malloc_src(void *ptr, Dx *db, size_t size);

void *Dx_calloc(size_t nmemb, size_t size);
void *Dx_calloc_src(Dx *db, size_t nmemb, size_t size);
void *Dx_frame_calloc(void *ptr, size_t nmemb, size_t size);
void *Dx_frame_calloc_src(void *ptr, Dx *db, size_t nmemb, size_t size);

void *Dx_realloc(void *ptr, size_t size);
void *Dx_realloc_src(Dx *dbsrc, void *ptr, size_t size);
void *Dx_frame_realloc(void *frame, void *ptr, size_t size);
void *Dx_frame_realloc_src(void *frame, Dx *db, void);
```

These functions correspond to the standard `malloc(3)`, `calloc(3)`, and `realloc(3)` functions. There are four variants of each: one that is associated with and uses defaults inherited from the frame on top of the stack; one that is associated with the frame on top of the stack but uses an explicitly specified memory source; one that is associated with a specified frame and that frame's defaults; and one that is associated with a specified frame and a specified memory source. Allocated memory is suitably aligned for any kind of variable.

`Dx_malloc()` allocates `size` bytes of memory associated with the frame on top of the stack. If no frame exists, one is created and its default memory source is `malloc(3)`. An allocation of zero bytes is allowed. `Dx_malloc_src()` allocates `size` bytes of memory from memory source `db` (`NULL` means use `malloc(3)`) associated with the frame on top of the stack. `Dx_frame_malloc()` is the same as `Dx_malloc()` except that the given frame is used instead of the top of stack (passing the `NULL` value for `ptr` identifies the top of stack). Lastly, `Dx_frame_malloc_src()` allocates `size` bytes of memory from the given memory source and associated with the given frame (again, `ptr == NULL` refers to the top of stack and `db == NULL` selects `malloc(3)`).

Arguments to `Dx_calloc()`, `Dx_calloc_src()`, `Dx_frame_calloc()`, and `Dx_frame_calloc_src()` are analogous, except these functions provide semantics similar to `calloc(3)`.

Arguments to `Dx_realloc()`, `Dx_realloc_src()`, `Dx_frame_realloc()`, and `Dx_frame_realloc_src()` are analogous, except these functions provide semantics similar to `realloc(3)`. If `ptr` is `NULL`, each of these is equivalent to the corresponding `Dx_malloc()` function. If `ptr` is not `NULL`, it must have been returned by a previous call to one of the frame memory allocation functions. If `size` is zero, nothing is done. Note that the memory source for a reallocation may be different from the source used to allocate the memory pointed to by `ptr`.

6.3 Freeing Frames

```
int Dx_free(void);
int Dx_free_frame(void *ptr);
int Dx_free_frames(void);
int Dx_free_all_frames(void);
```

`Dx_free()` frees all memory associated with the top frame and then removes and destroys the top frame. `Dx_free_frame()` frees all memory associated with the given frame, which must have been popped, and then destroys the frame. `Dx_free_frames()` frees all memory associated with all frames on the stack and destroys the frames. `Dx_free_all_frames()` frees all memory associated with all frames, whether on the stack or not, and destroys the frames.

6.4 Persistent Frames

Basic support is provided for persistent frames; that is, frames associated with a persistent database and memory allocation from those frames that persist after the application terminates.[1]

```
int Dx_frame_export(Dx *db, void *ptr, void **buf, size_t *buflen);
int Dx_frame_export_frame(Dx *db, void *frame, void *ptr, void **buf,
  size_t *buflen);
```

If calls to the frame memory functions allocate memory associated with a persistent database, it is necessary to *export* that memory so that it can be accessed when the database is reopened. `Dx_frame_export()` locates all allocations that are associated with `db` on the top frame, uses `malloc(3)` to allocate a buffer containing meta information for these allocations, sets `buf` to point to the buffer, and sets `buflen` to the size of the buffer. If `ptr` is not `NULL`, it is assumed to be a pointer to any memory allocation from the top frame. The caller is responsible for saving the buffer contents, presumably within the database.

The purpose of the `ptr` argument is to later give the user a handle on an area of allocated memory that can be used to restore other application-level state.

`Dx_frame_export_frame()` performs the same function using the given frame (the top frame, if `frame` is `NULL`).

After exporting a frame, no further allocations should be made on the frame and the frame must not be freed.

```
int Dx_frame_import(Dx *db, void *buf, size_t buflen, void **ptr);
void *Dx_frame_import_frame(Dx *db, void *buf, size_t buflen, void **ptr);
```

`Dx_frame_import()` is used to reinstate the state of saved frames from `db` that was previously saved by `Dx_frame_export()` or `Dx_frame_export_frame()`. The same buffer contents and length returned by `Dx_frame_export()` or `Dx_frame_export_frame()` when saving the frames are used. If a non-`NULL` memory pointer argument was given to `Dx_frame_export()` or `Dx_frame_export_frame()`, and `ptr` is not `NULL`, it is set to point to the same allocation.

`Dx_frame_export()` pushes the new frame on top of the stack, `Dx_frame_export_frame()` instead returns the new frame. The new frame has `db` as its default memory source.

[1] The ability to do this fell out of **Dxstore**'s implementation, so the feature was easy to provide. Because it's not the main thrust of this project, it isn't particularly well developed.

Note that no conversions, such as taking care of byte-ordering or pointer swizzling, are performed on the user's allocated data. Pointer swizzling and unswizzling functions are provided, however.

```
Dx_rc Dx_swizzle(Dx *db, void **frame_mem_ptr);
Dx_rc Dx_frame_swizzle(Dx *db, void *frame, void **frame_mem_ptr);
Dx_rc Dx_unswizzle(Dx *db, void **swizzledp);
Dx_rc Dx_frame_unswizzle(Dx *db, void *frame, void **swizzledp);
```

`Dx_swizzle()` operates on the top frame and `Dx_frame_swizzle()` operates on the frame argument. If no error occurs, `frame_mem_ptr`, which is a pointer to frame memory associated with `db`, is replaced by a swizzled pointer; obviously, it must not be used until it is unswizzled.

`Dx_unswizzle()` operates on the top frame and `Dx_frame_unswizzle()` operates on the frame argument. The swizzled pointer `swizzledp` is replaced with an unswizzled pointer to frame memory. If no error occurs, the returned pointer may be used normally.

The following illustrates a simple example without error checking, where `db` is a persistent database.

```
void *fbuf;
size_t buflen;
char **vec;
Dx_datum key, value;

Dx_new_frame(db, "persistent frame");
vec = (char **) Dx_malloc(10 * sizeof(char *));
vec[0] = (char *) Dx_malloc(11);
strcpy(vec[0], "abcdefghij");
vec[1] = (char *) Dx_malloc(50);
strcpy(vec[1], "Auggie+Harley");
vec[2] = (char *) Dx_malloc(33);
strcpy(vec[2], "Hello world.");
vec[3] = (char *) Dx_malloc(11);
strcpy(vec[3], "0123456789");
vec[4] = NULL;

Dx_frame_export(db, vec, &fbuf, &buflen);
Dx_swizzle(db, (void **) &vec[0]);
Dx_swizzle(db, (void **) &vec[1]);
Dx_swizzle(db, (void **) &vec[2]);
Dx_swizzle(db, (void **) &vec[3]);
Dx_swizzle(db, (void **) &vec[4]);
Dx_key(&key, "mykey", 6);
Dx_value(&value, fbuf, buflen);
```

```
Dx_store(db, &key, DX_INSERT_UNIQUE, &value);
```

At some later time, after `db` is reopened, the following code might be used:

```
Dx_key(&key, "mykey", 6);
Dx_fetch(db, &key, &value);
Dx_frame_import(db, value.dptr, value.dlen, (void *) &vec);
Dx_unswizzle(db, &vec[0]);
Dx_unswizzle(db, &vec[1]);
Dx_unswizzle(db, &vec[2]);
Dx_unswizzle(db, &vec[3]);
Dx_unswizzle(db, &vec[4]);

for (i = 0; vec[i] != NULL; i++)
  printf("%s\n", vec[i]);
```

7 Other Topics

7.1 Performance and Benchmarks

Once database meta data has been pre-loaded into memory, fetching a direct item or retrieving any item's meta data is very efficient, typically requiring one disk access. When streamed I/O is used, storing and fetching very large items should be relatively efficient.

Here are some performance tips:

- Select a `page_size` that is the same as the operating system's virtual memory page size. Making it much bigger probably won't help, but it might be worth trying.
- Use the native byte ordering (compile **Dxstore** with `ENDIAN=DX_NATIVE_ENDIAN`).
- Use the 32-bit **Dxstore** configuration if possible. Fewer 64-bit operations and data will be used.
- Compile **Dxstore** with full optimization enabled, including function inlining.
- Using a keyspace is (currently) less efficient, more so if the keys within a keyspace are very long.
- Using very long keys will be less efficient.
- Using a value for `max_stripe_npages` that is too small for the values that are being stored will be less efficient.
- Stream larger values in and out of the database.
- Heavily editing a value may cause it to become seriously fragmented, resulting in less efficient access to it. A future enhancement might automatically defragment the data, but in the meantime the only solution is to fetch the entire value, delete it, and store it again.
- The main performance benefit of streamed I/O won't be realized until multiple stripes can be read and written concurrently.
- Don't request special alignment of keys or values unless it is necessary.

Since each `*dbm` database has a different feature set and different strengths, it is difficult to draw many conclusions from these simple-minded benchmarks. Also, these tests used an alpha version of **Dxstore**, which hasn't been particularly well-tuned.

None of the libraries were modified or tuned for the benchmarks, they were completely "out of the box" and used whatever defaults they were distributed with. It's possible that any of the databases might be coaxed to do better by changing compile-time or run-time configuration. On

the other hand, one might reasonably expect to see typical performance numbers using a database's default configuration when running a benchmark that isn't too peculiar.

All libraries were compiled without modification using GCC 2.8.1 with full optimization (-O3). Programs were run on a 266 MHz Pentium II, 128 MB memory, with a 2.5 GB Quantum Sirocco 2550A disk running on Linux 2.0.36. Each test was run at least five consecutive times on an otherwise idle system and the shortest elapsed time was recorded. The user (u), system (s), and elapsed (e) times, in seconds, are given.

Five databases were tested:

`ndbm`	version 5.3 (author: Berkeley Software Distribution). The successor to the original Unix `dbm` library.
`sdbm`	distributed with perl 5.005_03 (author: Ozan Yigit). An `ndbm` clone.
`gdbm`	version 1.8.0 (author: Philip A. Nelson and Jason Downs). Similar to `ndbm`, but with some important limitations removed and added features.
`db`	Berkeley DB version 3.0.55 (author: Sleepycat Software). A full-featured embedded database.
`dx`	version 0.1, 32-bit mode (author: Distributed Systems Software)

Four benchmarks were run:

1. Load 72,470 items, each of which is a unique English word, with the value the same as the key. The total number of bytes loaded (keys+values) is 133,4782. The database was deleted between runs.
2. Preload the database using Benchmark 1. Load another 72,470 items, each of which is the same as before except a '?' is appended to each key and value. The total number of bytes loaded (keys+values) is 1,479,722.
3. Preload the database using Benchmark 1. Iterate through the database, counting the number of items and total number of bytes (keys+values).
4. Load 832 files from `/usr/bin`, using pathnames as keys and the file contents as the value. The total number of bytes loaded (keys+values) is 21,995,279. The database was deleted between runs. `ndbm` and `sdbm` are not able to run this benchmark because the sum of the key and value lengths is too large.
5. Preload the database using Benchmark 1. Delete all 72,470 items, in random order.

Again, these results are for one particular environment; your mileage may vary.

#	**ndbm**	**sdbm**	**gdbm**	**db**	**Dxstore**
1	0.89u, 0.83s, 1.74e	1.08u, 0.63s, 1.73e	2.48u, 9.72s, 16.17e	7.51u, 14.15s, 24.46e	1.41u, 0.34s, 1.80e
2	1.69u, 3.23s, 4.98e	1.57u, 3.05s, 4.64e	2.56u, 12.93s, 23.36e	7.27u, 17.03s, 53.06e	1.58u, 0.70s, 2.31e
3	0.53u, 0.24s, 0.79e	0.41u, 0.02s, 0.45e	0.51u, 0.40s, 0.92e	1.83u, 0.20s, 2.04e	0.72u, 0.02s, 0.76e
4	-	-	0.09u, 2.48s, 8.58e	0.58u, 2.53s, 11.04e	0.33u, 3.76s, 10.13e (not streamed) 0.24u, 1.73s, 3.47e (streamed)
5	0.80u, 2.67s, 3.46e	1.04u, 2.61s, 3.66e	2.40u, 12.59s, 16.34e	3.18u, 14.36s, 19.15e	0.42u, 0.13s, 0.56e

The benchmark programs (but not the other databases) are included with the **Dxstore** distribution.

7.2 Requirements

The most important requirements are a working `mmap(2)` system call (and associated calls), 64 bit integers (whether longs or long longs), and 8 bit chars.

7.3 Installation

Please consult the `INSTALL` file for details.

7.4 Compatibility with `*dbm` databases

A **Dxstore** database cannot be used with a `*dbm` database nor can a `*dbm` database file be used directly with **Dxstore**. See Section 7.5 [Conversion], page 41.

7.5 Conversion

The software distribution includes several programs to convert `*dbm` databases to **Dxstore** format.

7.6 Backups

At present, standard Unix backup programs should be used on an inactive database or a database opened in read-only mode. This may cause holes in a database (which don't actually use disk storage) to be filled. A utility program, `dxcp`, is provided to copy a volume and preserve any holes it may have.

7.7 Limits

Both 32-bit and a 64-bit **Dxstore** configurations can be created. The main difference is that the 32-bit configuration uses unsigned 32-bit integers for the lengths of keys, values, and internal objects whereas the 64-bit configuration uses unsigned 64-bit integers for them. The 64-bit configuration will obviously use more memory, both on disk and in memory, and there will likely be a slight performance hit when running on a 32-bit architecture. The 64-bit configuration does *not* require 64-bit system calls or C library calls.

A 64-bit configuration cannot use a database created with the 32-bit configuration, and vice versa.[1] If you're fairly sure you won't need the increased capacity of a 64-bit database, you're better off using a 32-bit database which will have lower time and space overhead.

To use the 32-bit configuration, define `DX_BITS` to be 32 when compiling files that use **Dxstore** and link with the 32-bit library ('`libdx32.a`'). To use the 64-bit configuration, define `DX_BITS` to be 64 when compiling files that use **Dxstore** and link with the 64-bit library ('`libdx64.a`').

[1] It will be possible to dump a 32-bit database and reload it into a 64-bit database, however.

- The number of simultaneously open databases depends on your operating system and the maximum number of open file descriptors.
- The number of items with identical hash values (which includes duplicate keys, of course) is limited. The actual limit depends on the length of the keys and the `page_size`. It could be as low as two.[2]
- The maximum key length is (2^32)-1.
- The maximum size of a volatile database depends on how much swap space is available.

Using very long keys will hurt efficiency somewhat.

A **Dxstore** database may have holes in it and so appear to be bigger than the amount of disk space actually allocated.

7.8 Problems and bugs

If you have problems with **Dxstore** or think you've found a bug, a bug report would be appreciated, but please reread the documentation first to make sure you aren't using the software incorrectly.

Before reporting a bug or trying to fix it yourself, please try to find the simplest possible program and input that demonstrates the problem. Send the demonstration code, input, and exact results **Dxstore** gave for it. Also identify the operating system, hardware architecture, and compiler that you are using. Please also describe the result you expected.

Please report the problem by sending email to

```
dxstore-bug@dss.bc.ca
```

Please include the version number of **Dxstore** you are using. You can get this information by printing the variable `Dx_version` (see Variables).

Questions about the software or documentation and suggestions for improvements are also welcome.

[2] A future release will probably remove the limit on identical hash values. This capability has been designed but not implemented.

You may contact the author at:

dxstore@dss.bc.ca

7.9 Bibliography

- Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture, 1999.
- "The UBC OSI Distributed Application Programming Environment: User Manual", Technical Report 90-37, Department of Computer Science, University of British Columbia, January, 1991. Some of the original ideas for frame memory came from this project.

7.10 Acknowledgements

Thanks to the following people who have contributed in various ways.

- Mike Sample (suggestions for improving functionality and usability),
- David Finkelstein (discussions and suggestions),
- Ozan Yigit <oz@nexus.yorku.ca> (author of the public domain `sdbm` hash function), and
- Bob Jenkins <bob_jenkins@compuserve.com> (author of the 64-bit hash function).

Function Index

A

C

D

F

G

M

O

R

Data Type Index

D

Variable Index

Concept Index

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z

Table of Contents