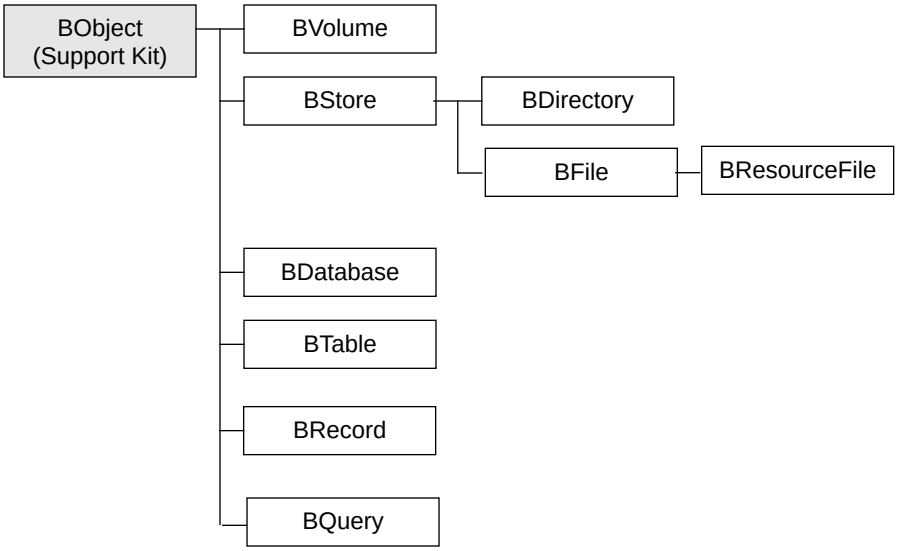
3 The Storage Kit

Storage Kit Inheritance Hierarchy



3 The Storage Kit

The Storage Kit lets your application store and retrieve *persistent* data. Persistent data doesn't disappear with your application; it's stored on a long-term storage device, such as a hard disk, floppy disk, CD-ROM, and so on, so you can return to it later.

The classes provided by the Kit fall into three categories:

- The database classes (BDatabase, BTable, BRecord, and BQuery) let you store data as "structured entries" or *records*. The content of a record—the number of individual datums it contains, and the type of values each datum can assume—depends on the record's structure. The description of this structure is given by the table to which the record conforms. Because records ares structured, you can easilyr and quickly locate a specific record based on the values that are stored in the record.

- The file system classes (BStore, BDirectory, BFile, and BResourceFile) provide a means for storing data in files. The data in a file can be unstructured (instances of BFile) or structured (BResourceFile).

- Instances of the BVolume class represent the actual storage devices themselves. BVolumes objects are used in both database and file-system applications.

It's suggested that you explore the Storage Kit with by visiting the BVolume class description, and then proceed to the database or file system classes in the orders given above.

BDatabase

Derived from: public BObject

Declared in: <storage/Database.h>

Overview

A BDatabase object represents a collection of structured, persistent data called a *database*. Each BDatabase object that you introduce to your application corresponds to an actual database and gives you access to it. Databases are contained within *volumes*, where a volume is a storage medium such as a hard disk, floppy disk, or CD-ROM. The relationship between databases and volumes is one-to-one: Each volume contains exactly one database. Part of the system's disk-formatting routine includes the creation of a database for the volume on the disk.

Finding a BDatabase

You never construct BDatabase objects yourself; instead, you ask the system to construct them and return them to you. There are two ways to do this:

- *You can ask a BVolume object for its BDatabase.* The BVolume **Database()** function returns the BDatabase object that represents the volume's database. Of course, this methodology merely shifts the burden to finding BVolume objects: You can walk down your application's "volume list" through repeated calls to the global **volume_at()** function. You can then pluck the BDatabase from each BVolume, as demonstrated below:

```
void DatabasePlucker(BList *dList)
{
   BVolume this_vol;
   BDatabase *this_db;
   long index;

   for (index = 0; this_vol = volume_at(index); index++)
   {
      this_db = this_vol.Database();
      dList->AddItem(this_db);
   }
}
```

- *You can retrieve a BDatabase based on a database ID.* Every database is identified by a unique integer of type **database_id**. By passing a valid database ID to the

global **database_for()** function, you can retrieve the BDatabase object that represents the identified database.

An important feature of Database ID numbers is that they're persistent: If you cache a **database_id** value and reboot your machine, the cached valued will refer to the same database when your machine comes back up. You can retrieve a BDatabase's ID number (the **database_id** value of the underlying database) through the **ID()** function.

Just as you never construct BDatabase objects, so do you not destroy them. These tasks are performed automatically by the Storage Kit.

After you've retrieved a BDatabase object, you may wonder what you should do with it. A BDatabase has essentially two purposes: It acts as a key to the Storage Server, and it lets you find and create tables (BTable objects). These activities are described below.

BDatabase as a Key to the Storage Server

Every transaction with the Storage Server requires a database ID—every time you retrieve a record or search for a file (as two examples), you need to tell the Storage Server which database to look in. Curiously, however, BDatabase objects don't appear in your application very often. This is because almost every Storage Kit object is created (or "validated" for whatever that means to the object) in reference to a paticular BDatabase which it (the newly created object) remembers for future use. In other words, BDatabase objects show up when you're creating other objects, but you can pretty much ignore them beyond that.

To give you a better idea of how this works, the following sections examine the relationships between BDatabase objects and instances of the other Storage Kit classes.

The Database Side: BTable, BRecord, and BQuery

These three classes, along with BDatabase itself, comprise the "database" side of the Storage Kit. BTable objects are created for you—each BDatabase object contains a list of BTable objects (as described in the next section). This proprietary relationship (between a BTable and the BDatabase that "owns" it) means that a BTable always knows how to get to a database.

BRecord objects are born knowing about the facts of lfe: Each of the four versions of the BRecord constructor takes an argument that, directly or indirectly, identifies a database.

Unlike the others, a BQuery object can be constructed without reference to a database. But but such an object is essentially useless until you tell it which database it should operate on.

The File System Side: BVolume and BStore

BVolume and BStore along with BStore's derivations, BDirectory, BFile, and BResourceFile, are the Storage Kit's file system classes. The relationship between databases and volumes was described earlier: A BVolume object always knows its database.

The BStore class is similar to BQuery in that you can create an instance of (a class derived from) BStore without reference to a database, but the object will be useless until its database is set. You do this by setting the object's **record_ref** structure. The structure uniquely identifies a record in a database by listing (as structure fields) the database ID and the record ID (record ID numbers are unique within a database). **record_ref** structures (or *refs*) are the primary means for identifying a file; they're visited again in the BStore class description.

Finding and Creating Tables

As mentioned earlier, BTable objects live within BDatabase objects: When you "open" a database (by asking for the BDatabase that represents it), the tables that are stored within are automatically represented in your BDatabase object as BTable objects. To get a BTable from a BDatabase, you can ask for it by name, through the **FindTable()** function, or you can step through the BDatabase's "table list" by using **CountTables()** and **TableAt()**.

To create a table, you call the **CreateTable()** function. The function tells the Storage Server to manufacture a table in the database, and then constructs a BTable object to represent it, adds it to the BDatabase's table list, and returns the new object.

A BDatabase's table list can fall out of step with the database. Specifically, your object's table list isn't automatically updated when another application adds a new table to the database. To update your object's table list, you call BDatabase's **Sync()** function.

Constructor and Destructor

The BDatabase constructor and destructor are private. You never construct BDatabase objects directly; instead, you retrieve them from the system through the global **database_for()** function, or through BVolume's **Database()** function.

Member Functions

CountTables()

long **CountTables(**void**)**

Returns the number of BTables in the BDatabase's table list.

See also: **TableAt()**, **FindTable()**, **Sync()**

CreateTable()

BTable ***CreateTable(**char **tableName***)**
BTable ***CreateTable(**char **tableName*, char ***parentName***)**
BTable ***CreateTable(**char **tableName*, BTable ***parentTable***)**

Creates a table in the database, names it *tableName*, and constructs (and returns) a BTable object to represent it. The table that's created by the first version of this function will be empty—it won't contain any fields. In the other two versions, the new table will "inherit" the fields of the parent table (there's no functional difference between these two versions—they simply give you two ways to designate the parent table).

The BDatabase doesn't check to make sure that the name of the new table is unique: You can create a table with a given name even if that name identifies an existing table. If you want to make sure that your table's name won't collide with that of an existing table, you should call **FindTable()** first—and if you really want to be scrupulous, you should call **Sync()** just before that:

```
/* Create a uniquely-named table called "Phylum". */
a_db->Sync();
if (a_db->FindTable("Phylum") == NULL)
    a_table = a_db->CreateTable("Phylum");
```

Furthermore, if you designate a parent but the parent isn't found, the new table is created without a parent. Again, you can check to make sure that the parent exists:

```
/* Create a uniquely-named table that inherits from the
 * existing table called "Kingdom".
 */
a_db->Sync();
if (a_db->FindTable("Phylum") == NULL &&
    a_db->FindTable("Kingdom") != NULL)
    a_table = a_db->CreateTable("Phylum", "Kingdom");
```

If **CreateTable()** can't create the table—this should only happen if the Storage Server can no longer communicate with the database—it returns **NULL**.

You never explicitly delete a BTable object. Constructing and deleting BTable objects is the BDatabase's responsibility.

See also: **FindTable()**

FindTable()

BTable ***FindTable(***char **table_name***)**

Looks in the BDatabase's table list for the BTable that represents the named table. Returns the BTable if it's there; **NULL** if not. The table list includes all tables that live in the database—it isn't just a compilation of tables that were created by this particular object.

If you want to make sure that the list is up-to-date before looking for a table, you should first call BDatabase's **Sync()** function.

See also: **TableAt()**, **Sync()**

ID()

database_id **ID(**void**)**

Returns an identifier that uniquely and persistently identifies the BDatabase's database. The value is meaningful system-wide—you can send it to other applications so they can find the same database, for example. The persistence of the value is eternal: The database that's identified by a particular **database_id** number today will still be identified by that number long after you've forgotten everything you ever knew.

Database ID numbers appear most commonly as the **database** fields of **record_ref** structures. A **record_ref** structure uniquely identifies a record among all records in all databases.

See also: **BStore::SetRef()**

IsValid()

bool **IsValid(**void**)**

Returns **TRUE** if the BDatabase's database is (still) available; otherwise, it returns **FALSE.** The object will become invalid if the volume on which the database lives is unmounted.

Warning: Currently, this function always returns **TRUE**.

PrintToStream()

void **PrintToStream(**void**)**

Displays, to standard output, information about the BTables that are contained in the BDatabase's table list. The information is displayed in this format:

```
| index-table <name>, id #
        | fieldName1
        | fieldName2
```

```
        | fieldName3
        ...
```

For example, if the first BTable in the list is named “Shirts” and contains fields named “color,” “texture,” and “buttonCount,” the display will look like this:

```
| 0-table <Shirts>, id 0
|  | color
|  | texture
|  | buttonCount
```

A BTable that inherits from another BTable is indented beneath its parent, and repeats the inherited fields:

```
| 0-table <Shirts>, id 0
|  | color
|  | texture
|  | buttonCount
|  | 1-table <TackyShirts>, id 1
|  |  | color
|  |  | texture
|  |  | buttonCount
|  |  | hasStripes
|  |  | isHawaiian
```

PrintToStream() is meant to be used as a debugging tool and party game.

Sync()

void **Sync(**void**)**

Synchronizes the BDatabase object with the database that it represents by doing the following:

- Updates the BDatabase’s table list so its contents match that of the database’s list.
- Makes sure that all “committed” record data (in the sense of the word as defined by the BRecord class) has been flushed to the underlying storage media (in other words, it writes your changes to the disk).

Calling **Sync()** is the only way to update the BDatabase’s table list, whereas it isn’t necessary to **Sync()** in order write committed data. Such data will (eventually) be written to the disk as a matter of routine (within seconds, typically); **Sync()**, in this regard, is a sop for the anxious.

See also: **BRecord::Commit()**

TableAt()

BTable ***TableAt(**long *index***)**

Returns the *index*'th BTable object in the BDatabase's table list (zero-based).

If you want to make sure that the list is up-to-date before looking for a table, you should first call BDatabase's **Sync()** function.

See also: **CountTables()**, **Sync()**

VolumeID()

long **VolumeID(**void**)**

Returns the ID of the volume that contains the database that's represented by this BDatabase object.

BDirectory

Derived from: public BStore

Declared in: <storage/Directory.h>

Overview

The BDirectory class defines objects that represent directories in a file system. A directory can contain files and other directories, and is itself contained within a directory (its "parent"). As with all BStore objects, a BDirectory is useless until its ref is set.

You use BDirectory objects to browse the file system, and to create and remove files (and directories). These topics are examined in the following sections. After that it's nap time.

Browsing the File System

Directories are the essence of a hierarchical file system. By placing directories inside other directories, you increase the depth of the hierarchy (currently, the nesting can be 64 levels deep). Some of the rules that govern the Be file system hierarchy are:

- Every file system has exactly one "root" directory. The root directory stands at the base of the hierarchy: If you ask any file for its parent, and then ask the parent for its parent, and so on, the directory you arrive at, when you run out of parents, is the root directory.
- Except for the root directory, every file system entity (every file and directory) has exactly one parent (every file is contained in exactly one directory).
- As a corollary to this, the hierarchy's nesting is directed and acyclic: If you follow a path of directories, you won't find yourself re-tracing your steps. (Note that the Be file system doesn't currently support symbolic links; such links can cause cyclic recursion)

Descending the Hierarchy

If we wanted to browse an entire file system, we get a root directory, and recursively ask for its contents and the contents of the directories it contains.

First, we get a root directory from a volume by calling BVolume's **GetRootDirectory()** function; in the example here, we get the root directory of the boot volume:

```
/* We'll get the root directory of the boot volume. */
BVolume boot_vol = boot_volume();
BDirectory root_dir;

boot_vol.GetRootDirectory(&root_dir);
```

Since the Be file system is acyclic, we can implement the hierarchy descent in a single recursive function. In this simple implementation we ask the argument directory for its contents (first its files, then its directories), print the name of each entry, and then re-call the function for each of its directories. The *level* argument is used to indent the names to make the nesting clear:

```
void descend(BDirectory *dir, long nest_level)
{
    long index = 0, nester;
    BFile a_file;
    BDirectory a_dir;
    char name_buf[B_FILE_NAME_LENGTH];
```

First we print the name of this directory (followed by a distinguishing slash):

```
    dir->GetName(name_buf);
    for (nester = 0; nester < nest_level; nester++)
        printf(" ");
    printf("%s/\n", name_buf);
```

Now we get the files; **GetFile()** returns **B_ERROR** when the index argument is out-of-bounds:

```
    while (dir->GetFile(index++, &a_file) == B_NO_ERROR) {
        a_file.GetName(name_buf);
        for (nester = 0; nester < nest_level + 1; nester++)
            printf(" ");
        printf("%s\n", name_buf);
    }
```

Finally, we call **descend()** for each sub-directory:

```
    index = 0;
    while (dir->GetDirectory(index++, &a_dir) == B_NO_ERROR)
        descend(&a_dir, nest_level + 1);
}
```

The example demonstrates the use of **GetFile()** and **GetDirectory()**. There are two versions of each of these functions: The version of each shown here gets the *index*'th item in the calling directory. The other version finds an item by name (see the **GetFile()** description for details).

Getting Many Files at a Time

GetFile() and **GetDirectory()** are reasonably efficient—but they're not as fast as **GetFiles()** and **GetDirectories()**. As their names imply, the latter functions retrieve more than one

item at a time; each set of files that's retrieved requires fewer messages to the Storage Server, thus the retrieval is much faster than getting each file individually.

The **GetFiles()** function doesn't retrieve files as BFile objects; instead, it writes their **record_ref**s into a vector that you pass (as a **record_ref** pointer) to the function. You then use the **record_ref** values to refer BFile objects to the underlying files. (This is also true, modulo store type, for **GetDirectories()**.)

Here, we modify the **descend()** function to use **GetFiles()** and **GetDirectories()**:

```
void descend(BDirectory *dir, long nest_level)
{
    long index, nester;
    BFile a_file;
    BDirectory a_dir;
    long file_count, dir_count;
    record_ref *ref_vector;
    char name_buf[B_FILE_NAME_LENGTH];

    dir->GetName(name_buf);
    for (nester = 0; nester < nest_level; nester++)
        printf(" ");
    printf("%s/\n", name_buf);
```

We have to allocate the ref vector; rather than do it twice (once for files, once for directories), we grab enough to accommodate the larger of the two sets. The **CountFiles()** and **CountDirectories()** functions, used below, do pretty much what we expect:

```
    file_count = dir->CountFiles();
    dir_count = dir->CountDirectories();
    ref_vector = (record_ref *)malloc(sizeof(record_ref) *
                    max(dir_count, file_count));
```

GetFiles() gets the refs for the files. The first two arguments are 1) an offset into the directory's list of files, and 2) the number of refs we want to retrieve.

```
    dir->GetFiles(0, file_count, ref_vector);

    for (index = 0; index < file_count; index++) {
        if (a_file.SetRef(ref_vector[index]) < B_NO_ERROR)
            continue;
        a_file.GetName(name_buf);
        for (nester = 0; nester < nest_level+1; nester++)
            printf(" ");
        printf("%s\n", name_buf);
    }
```

Now do the same for directories:

```
        dir->GetDirectories(0, dir_count, ref_vector);

        for (index = 0; index < dir_count; index++) {
            if (a_dir.SetRef(ref_vector[index]) < B_NO_ERROR)
                continue;
            descend(&a_dir, nest_level + 1);
        }
        /* Don't forget to free the vector. */
        free(ref_vector);
    }
```

Path Names and File Names

Although **record_ref**s are the common currency for finding and accessing files in the file system, it's also possible to get around using path names and file names. The BDirectory class provides a number of functions that operate on names:

- **GetFile()** and **GetDirectory()**, as mentioned above, come in flavors that take names rather than indices. The functions look for a file or directory, within the invoked-upon BDirectory, that goes by the name given in the first argument.
- **GetRefForPath()** takes a path name as its first argument and returns, by reference in its second argument, the **record_ref** that identifies the named file or directory.
- **Contains()** is a convenient boolean function that takes a name as it's only argument and returns **TRUE** if the invoked-upon BDirectory contains an item of that name.

Modifying the File System

The BDirectory class provides functions that let you create new files and add them to the files system, and remove existing files.

- To create a new file, you call the **Create()** function.
- To remove an existing file, you call **Remove()**.

While BDirectory's **Remove()** is the *only* way to programatically remove an item from the file system, files can be created as copies of other files through BFile's **CopyTo()** function. You can't copy a directory.

Constructor and Destructor

BDirectory()

BDirectory(record_ref *ref***)**
BDirectory(void**)**

The two BDirectory constructors create and return pointers to newly created BDirectory objects. The version that takes a **record_ref** argument attempts to refer the new object to the argument; the no-argument version creates an unreferenced object. In the latter case, you must set the BDirectory's ref in a subsequent manipulation. This you can do thus:

- By invoking the object's **SetRef()** function (the function is inherited from the BStore class).
- By passing the object as an argument to the BDirectory functions **Create()** or **GetDirectory()**.
- By passing it as an argument to BVolume's **GetRootDirectory()** function.

~BDirectory()

virtual **~BDirectory(**void**)**

Destroys the BDirectory object; this *doesn't* remove the directory that the object corresponds to. (To remove a directory, use BDirectory's **Remove()** function; note that you can't remove a volume's root directory.)

Member Functions

Contains()

bool **Contains(**const char **name***)**

Looks in the BDirectory for a file or directory named *name*. If the item is found, the function returns **TRUE**, otherwise it returns **FALSE**. If you need to know whether the item is a file or a directory, you should follow this call (if it returns **TRUE**) with a call to **IsDirectory()**, passing the same name:

```
if (aDir->Contains("Something"))
    if (aDir->IsDirectory("Something"))
        /* It's a directory. */
    else
        /* It's a file. */
```

See also: **IsDirectory()**, **GetFile()**, **GetDirectory()**

CountDirectories() *see* CountFiles()

CountFiles(), CountDirectories(), CountStores()

long **CountFiles(**void**)**
long **CountDirectories(**void**)**
long **CountStores(**void**)**

Returns a count of the number of files, directories, or both that are contained in this BDirectory.

See also: **GetFile()**, **GetFiles()**

CountStores() *see* CountFiles()

Create()

long **Create(**const char ***newName**,
BStore ***newItem**,
const char ***tableName** = NULL**,**
store_creation_hook ***hookFunc** = NULL**,**
void ***hookData** **= NULL)**

Creates a new file system item, names it *name*, and adds it to the directory represented by this BDirectory. The ref of the *newItem* argument is set to represent the added item. *newItem* must either be a BFile or BDirectory object—the object's class dictates whether the function will create a file or a directory.

The other three arguments (*tableName*, *hookFunc*, and *hookData*) are infrequently used—you should only need them if you want your file system records to conform to a custom tables. See "The Store Creation Hook" on page 73 (in the BStore class) for more information.

The function returns **B_NO_ERROR** if the item was successfully created.

GetDirectory() *see* GetFile()

GetFile(), GetDirectory()

long **GetFile(**const char **name*, BFile **file***)**
long **GetFile(**long *index*, BFile **file***)**
long **GetDirectory(**const char **name*, BDirectory **dir***)**
long **GetDirectory(**long **index*, BDirectory **dir***)**

Looks for the designated file or directory (contained in this BDirectory) and, if it's found, sets the second argument's ref to represent it. The second argument must point to an allocated object—these functions won't allocate it for you.

The *name* versions of the functions search for the appropriate item with the given name. For example, the call

```
BFile *aFile = new BFile();
if (aDir->GetFile("something", aFile) < B_NO_ERROR)
    /* Not found. */
```

looks for a file named "something". It ignores directories. Similarly, the **GetDirectory()** function looks for a named directory and ignores files. As implied by the example, the function returns **B_NO_ERROR** if the named item was found.

The *index* versions return the *index*'th file or directory. For example, this

```
if (aDir->GetFile(0, aFile) < B_NO_ERROR)
    ...
```

gets the first file, while this

```
BDirectory *aSubDir = new BDirectory();
if (aDir->GetDirectory(0, aSubDir) < B_NO_ERROR)
    ...
```

gets the first directory.

The index versions return a less-than-**B_NO_ERROR** value if the index is out-of-bounds.

See also: **Contains()**, **IsDirectory()**, **GetFiles()**

GetFiles(), GetDirectories(), GetStores()

long **GetFiles(**long *index*, long *count*, record_ref **refVector***)**
long **GetDirectories(**long *index*, long *count*, record_ref **refVector***)**
long **GetStores(**long *index*, long *count*, record_ref **refVector***)**

These functions retrieve a vector of refs that identify some number of files, directories, or both within the this BDirectory. The *index* and *count* arguments tell the functions where, within a list of items, to start plucking refs and how many refs to pluck; the plucked refs are placed in *refVector*. For example, **GetFiles()** makes a list of all the file refs in this BDirectory; it then places, in *refVector*, the *index*'th through the (*index*+*count*)'th refs.

If you set *index* and *count* such that all or part of the desired range is out-of-bounds, these functions don't complain: They retrieve as many refs as are in-bounds and return those to you. Thus, the number of refs that are passed back to you may be less than the number you asked for.

You must allocate *refVector* before you pass it into these functions. It's the caller's responsibility to free the vector.

The functions return **B_ERROR** if the BDirectory's ref hasn't been set. Otherwise, they return **B_NO_ERROR**.

See "Getting Many Files at a Time" on page 16 for an example of the use of **GetFiles()** and **GetDirectories()**.

See also: **GetFile()**

GetRefForPath()

long **GetRefForPath(**const char ***pathName*, record_ref ***ref***)**

Searches for the files that's named by the given path name. If the file is found, it's ref is placed in *ref* (which must be allocated before it's passed in).

If the path is relative, the search starts at this BDirectory (the path name is appended to the path that leads to this object). If it's absolute, the search starts at the root directory. In the absolute case, the receiving BDirectory doesn't figure into the search: An absolute path name search invoked on any BDirectory object yields the same result.

Path names are constructed by concatenating directory and file names separated by slashes ("/"). Absolute path names have an initial slash; relative path names don't. Keep in mind that an absolute path name must include the root directory name.

Warning: This function fails if the path name ends in a slash, even if it otherwise identifies a legitimate directory.

The function returns **B_NO_ERROR** if a ref was successfully found; otherwise, it returns **B_ERROR**. Note that the BDirectory's ref must be set for this function to succeed, even if the path name is absolute.

IsDirectory()

bool **IsDirectory(**const char ***name***)**

Returns **TRUE** if the BDirectory contains a directory named *name*; if the object doesn't contain an item with that name, if the item is a file, or if other impediments obtain, the function returns **FALSE**.

See also: **Contains()**

Remove()

long **Remove(**BStore **anItem***)*

Removes the given item from the object's directory, removes the item's record from the database, and frees the (disk) space that it was using. If *anItem* is a BFile, the object is closed before it's removed. The item must be a member of the target BDirectory.

You can't remove a volume's root directory (it doesn't have a parent, so there's no way to try). Also, you can't remove a directory that isn't empty.

The function returns **B_NO_ERROR** if the item was successfully removed; otherwise, it returns **B_ERROR**.

BFile

Derived from: public BStore

Declared in: <storage/File.h>

Overview

The BFile class defines objects that represent files in the file system. Files are containers of data that live in directories. A file can live in only one directory at a time.

BFile inherits from BStore; the basic concepts of how file system objects work are explained in the BStore description. The most important points, applied to BFiles, are these:

- Every item in the file system has a database record associated with it. The record contains information about the item, such as its name and where it's located.
- A record is uniquely identified across all databases by its **record_ref** structure. Posing as a value, a **record_ref** is called a "ref".
- A BFile object is associated with an actual file by referring to the ref of the file's record. This association can be performed through the BFile constructor, through the **BStore::SetRef()** function, as well as through a number of other BStore-related functions.
- More than one BFile object can be associated with (or can "refer to") the same underlying file. This is simply a matter of setting the refs of the various BFile objects to the same value.
- Conversely, the same BFile object can be re-used to refer to any number of different files (although only one file at a time).

BFile Data

BFiles contain "flat" or unstructured data. They're commonly used to store ASCII documents, for example. If you want to associate structured header information with a file (if you want a complementary "resource fork"), you can do one (or more) of the following:

- *Use an instance of BResourceFile*. The BResourceFile class inherits from BFile. The data in a BResourceFile is completely structured; the structure can be defined dynamically. Each "slot" in the structure of a BResourceFile is called a *resource*.

To use a BResourceFile so that it emulates a data/resource fork pair, you would install the flat data as one of the file's resources. An important drawback to using a BResourceFile is that the structure *is* the file, thus the file may not be portable to other computers. Note that executable files are automatically created as resource files.

- *You can create your own BFile-derived class.* What you do in your class to "specialize" your files is up to you. To help in the effort, BFile provides a **FileCreated()** hook function that's automatically invoked when you create a new file as an instance of your class (specifically, it's invoked as part of BDirectory's **Create()** and BFile's **CopyTo()** functions).

- *You can create your own table to which your BFiles' records conform.* The function that creates wholly new files (BDirectory's **Create()**) lets you set the name the table that's used to create the file's record. You would then supply a "store creation hook" that modifies the fields that you've defined as new files are created. The store creation hook, which was explained in the BStore class, is a call-back function (it *isn't* a class hook function) that you pass as an argument to BDirectory's **Create()**, BStore's **MoveTo()**, and BFile's **CopyTo()** functions.

- *You can add "extra" entries to the BFile's record.* This is performed through BRecord's **SetExtra()** function. The advantage of an extra entry is that it doesn't have to be part of the definition of the table to which the record conforms—in other words, extra entries can be added (and removed) dynamically without re-defining the record's table. A single record can hold any number of extra entries.

- *Use the* **SetTypeAndApp()** *function.* If all you want to do is be able to identify the "type" or "creator" of a file, you can use BFile's **SetTypeAndApp()** and **GetTypeAndApp()** functions (where the "App" in the function name means the same as the traditional "creator"). The advantage of this approach is that you can avoid everything described heretofore: You don't need to force the file's data into a non-portable structure, and you don't have fuss with the file's record.

Locating and Creating Files

Most of the functions that locate, create, and otherwise "externally" manipulate files are defined by the BStore and BDirectory classes. The most important of these are:

Defined in BStore:

- **SetRef()** is the fundamental function that establishes a "link" between a file and a BFile object. BFile augments this function (and so it's listed among the "Member Functions" section, below), but the primary documentation for it is in the BStore class.

- **MoveTo()** moves a file from one directory to another.

Defined in BDirectory

- **GetFile()** locates a file by name or index (into a directory) and refers a BFile to it.
- **Create()** creates a new file in the file system, and refers a BFile to it.
- **Remove()** removes a file from the file system.

The BFile class itself adds two whole-cloth file manipulation functions:

- **CopyTo()** creates a new file as a copy of the receiving BFile.
- **SwitchWith()** takes two files (the receiving BFile and a BFile that you pass as an argument) and switches their contents. This function is provided as an efficient way for an application to make back-up copies of the files that it's writing.

Opening and Closing Files

Before examining or manipulating a file, you have to open the BFile that refers to it by calling the **Open()** function. The object remains open until the **Close()** function is called.

The **Open()** function takes a single argument that you use to specify the file's "open mode". The constants that represent these modes are:

- **B_READ_ONLY**. In this mode, your BFile can read the file's contents, but it can't write into the file. Other BFile objects are allowed to open the file while your BFile has it open in read-only mode.
- **B_READ_WRITE** lets your object read and write the file. Again, other objects can also open the file.
- **B_EXCLUSIVE** gives you exclusive access (for reading and writing) to the file. No other BFile can open the file while your object has it open in this mode.

Reading and Writing Files

BFile's **Read()** and **Write()** functions are the means by which you examine and modify the data that lies in a file. They operate much as you would expect: For example, the BFile must be open in the appropriate mode, they read or write some number of bytes of data, and successive **Read()** or **Write()** calls read or write contiguous sections of the file.

An important point with regard to **Read()** and **Write()** is that they're not virtual. If you create a BFile-derived class because, for example, you want to read in units of **long**s rather than bytes, you have to create your own reading function (which might invoke **Read()**) and give it a different name. (This is what the Media Kit's BSoundFile class does: It reads "frames" of sound through the **ReadFrames()** function).

Hook Functions

FileCreated() Invoked when a new file is created. You implement this function in a BFile-derived class to perform class-specific initialization. This initialization can include modification of the new file's BRecord.

Constructor and Destructor

BFile()

BFile(void**)**

The BFile constructor creates a new, unreferenced object, and returns a pointer to it. The object won't correspond to an actual file until its record ref is set. You can set the ref directly by calling the **SetRef()** function, or you can allow the ref to be set as a side effect by passing your BFile object as an argument to any of these functions:

- **BFile::CopyTo()**
- **BDirectory::Create()**
- **BDirectory::GetFile()**

~BFile()

virtual **~BFile(**void**)**

Destroys the BFile object; this *doesn't* remove the file that the object corresponds to (to remove a file, use BDirectory's **Remove()** function). The object is automatically closed (through a call to **Close()**) before the object is destroyed.

See also: **Close()**

Member Functions

Close()

virtual long **Close(**void**)**

Closes the BFile. The object's BRecord is automatically committed to the database when you call this function.

You should be aware that **Close()** is called automatically by the BFile destructor, and by BDirectory's **Remove()** function.

The BFile must previously have been opened through an **Open()** call. If the object isn't open (or, more broadly, if the BFile's ref hasn't been set), **Close()** returns **B_ERROR**; otherwise, **B_NO_ERROR** is returned.

See also: **Open()**

CopyTo()

long **CopyTo(**BDirectory ***toDir**,
const char **newName**,
BFile **newFile**,
store_creation_hook **createHook** = NULL,
void **createData** = NULL,
copy_status_hook **copyHook** = NULL**)**

Makes a copy of the BFile's file, moves the copy into the directory given by *toDir*, names it *newName*, and returns a new BFile object (by reference in *newFile*) that refers to the new file.

The *newName* argument *must* be supplied—if you want to copy the file but retain the same name as the original file, pass *this_object***->Name()** as the argument's value. You can also copy a file into the same directory (by passing *this_object***->Parent()** as the *toDir* argument); in this case, however, you must supply a different name for the copied file.

The BRecord that's created for the new BFile will conform to the same table as the BRecord of the original BFile (by default, this is the Kit-defined "File" table). Furthermore, the values in the new BRecord are copied from the original file's BRecord (with some obvious changes, such as the file's name, its parent, and so on). The new BRecord is committed just before **CopyTo()** returns. The **CopyTo()** function automatically commits the original object's BRecord as well.

If the new BRecord conforms to a custom table, you may want to modify the new BRecord before it's committed. The two "create" arguments provide this ability:

- *createHook* is a pointer to a "store creation hook" function. The function is called after the new BFile has been created and its BRecord's values set, but before the BRecord is committed. The new BFile is passed as the first argument to *createHook*. The value returned by *createHook* is significant: If it returns **B_ERROR**, the copy operation is aborted; **B_NO_ERROR** lets it continue.
- *createData* is a buffer of data that's passed as the second (and final) argument to the store creation hook function.

For more information on the use of the store creation hook mechanism, see "The Store Creation Hook" on page 73.

The final argument, *copyHook* is a "copy status hook" function. This function, if supplied, is invoked periodically as the copy operation progresses. The protocol for the hook is

long ***copy_status_hook_name*(**record_ref *ref*, int *size_delta*, void **no_op***)*

The *ref* argument is the ref of the file that's being copied from; *size_delta* is the amount of data that's been copied from the source file into the destination file since the last time the hook function was called; the final argument is currently unused. If the hook function returns a value other than **B_NO_ERROR**, the copy operation is halted, but the data that's already been copied isn't erased.

The rules governing the ability to copy a file into a specific directory are the same as those that apply to creating a file in that directory. Again, see the **BDirectory::Create()** function for more information.

The target BFile must be closed for the **CopyTo()** function to work. If the BFile couldn't be copied (for whatever reason) **B_ERROR** is returned; otherwise, **B_NO_ERROR** is returned.

See also: **SwitchWith()**, **BDirectory::Create()**, **BStore::MoveTo()**

FileCreated()

virtual long **FileCreated(**void**)**

This is a hook function that's automatically invoked when a new file is created. Specifically, it's invoked by BDirectory's **Create()** function and BFile's **CopyTo()** function. You can implement this function in a derived class to perform file-initialization operations. The file that's being created, in the context of the implementation, is referred to by the **this** pointer. The store creation hook that was passed to **Create()** or **CopyTo()** will already have been called and the file's record will have been committed by the time this function is invoked.

There are no restrictions on the operations that this function may perform; for example, you can implement **FileCreated()** to open and write the file, or modify and commit the file's record. Keep in mind, however, that the file's record will already have been committed for the first time just before this function is invoked.

You can stop the file from being created by implementing the function to return a value other than **B_NO_ERROR**.

GetTypeAndApp() *see* SetTypeAndApp()

Open(), OpenMode(), IsOpen()

virtual long **Open(**long *mode***)**
long **OpenMode(**void**)**
bool **IsOpen(**void**)**

The **Open()** function opens the BFile so its file's data can be read or written (or both). The file remains open until **Close()** is called.

The operations you can perform on an open file depend on the *mode* argument:

- If *mode* is **B_READ_ONLY**, you'll be able to read the file, but not write it.
- If it's **B_READ_WRITE**, you can read and write the file.
- If it's **B_EXCLUSIVE**, you can read and write the file *and* no other BFile object will be able to open the file until you call **Close()**. (The other two modes don't prevent the file from being opened by other objects.)

Note that the **B_EXCLUSIVE** mode doesn't prevent changes to the file that can be performed while the file is closed. For example, some other actor can delete the file (through the command line, Browser, or BDirectory's **Remove()** function) while your BFile holds the file open in exclusive mode.

If the BFile's ref hasn't been set, if some other BFile has the file open in **B_EXCLUSIVE** mode, if the mode argument isn't one of the values listed here, or if, for any other reason, the file couldn't be opened, **Open()** returns **B_ERROR**. Upon success, it returns **B_NO_ERROR**.

OpenMode() returns the mode that the file was opened with. In addition to the three modes listed above, the function can also return **B_FILE_NOT_OPEN** if the BFile isn't open.

IsOpen() returns **TRUE** if the BFile is open, and **FALSE** if not.

See also: **Close()**, **Read()**, **Write()**, **Seek()**

Read()

long **Read(**void **data*, long *dataLength***)**

Copies (at most) *dataLength* bytes of data from the file into the *data* buffer. The function returns the actual number of bytes that were read—this may be less than the amount requested if, for example, you asked for more data than the file actually holds.

The BFile's data pointer is moved forward by the amount that was read such that a subsequent **Read()** would begin at the following "unread" byte. Freshly opened, the pointer is set to the first byte in the file; you can reposition the pointer prior to a **Read()** call through the **Seek()** function. Keep in mind that the same data pointer is used for reading *and* writing data.

For this function to work, the BFile must already be open. If the object isn't open, or if, for any other reason, the file couldn't be read, the function returns **B_ERROR**.

See also: **Open()**, **Seek()**, **Write()**

Seek

long **Seek(**long *byteOffset*, long *relativeTo***)**

Relocates the BFile's data pointer. The location that you want the pointer to assume is given as a certain number of bytes (*byteOffset*) relative to one of three positions in the data. These three positions are represented by the following constants (which you pass as the value of *relativeTo*):

- **B_SEEK_TOP** represents the beginning of the file.
- **B_SEEK_MIDDLE** represents the pointer's current location.
- **B_SEEK_BOTTOM** represents the end of the file.

For example, the following moves the pointer five bytes forward from its present position:

```
aFile->Seek(5, B_SEEK_MIDDLE)
```

If *byteOffset* is negative, the pointer moves backwards. Here, the pointer is set to five bytes from the end of the file:

```
aFile->Seek(-5, B_SEEK_BOTTOM)
```

If you seek to a position beyond the end of a file, the file is padded with uninitialized data to make up the difference. For example, the following code doubles the size of *aFile*:

```
aFile->Seek( aFile->Size() * 2, B_SEEK_TOP)
```

Keep in mind that the padding is uninitialized; if you want to pad the file with **NULL**s (for example), you have to write them yourself.

The function returns the pointer's new location, in bytes, reckoned from the beginning of the file. You can use this fact to get the pointer's current position in the file:

```
/* The inquisitive, no-op seek. */
long currentPosition = aFile->Seek(0, B_SEEK_MIDDLE);
```

Seek() is normally followed by a **Read()** or **Write()** call. Note that both of these functions move the pointer by the amount that was read or written.

For the function to succeed, the BFile must already be open; **B_ERROR** is returned if the object isn't open.

Warning: Currently, seeking before the beginning of a file *isn't* illegal. Doing so doesn't affect the size or content of the file, but it does move the pointer to the requested (negative) location. The **Seek()** function will return this location as a negative number. A subsequent read or write on that location will cause trouble.

See also: **Open()**, **Read()**, **Write()**

SetRef()

virtual long **SetRef(**record_ref *ref***)**
virtual long **SetRef(**BVolume **volume*, record_id *recID***)**

Sets the BFile's ref. The BStore class defines the basic operations of these functions. These versions add a BFile-specific wrinkle: They close the object before setting the ref.

See also: **BStore::SetRef()**

SetTypeAndApp(), GetTypeAndApp()

long **SetTypeAndApp(**ulong *type*, ulong *app***)**
long **GetTypeAndApp(**ulong **type*, ulong **app***)**

These functions set and return, respectively, constants that represent the file's contents (its "type"), and the application that created the file. The Browser uses these constants to display an icon for the file, and to launch the appropriate application when the file is opened.

If the application that you're designing creates new files, you should set the type and app for these files through **SetTypeAndApp()** (this information *isn't* set automatically). The *app* value must be an application signature. You can retrieve your application's signature through **BApplication::GetAppInfo()**.

When the Browser tells an application to open a file, the app can use **GetTypeAndApp()** to look at the file's type constant to determine how the file should be opened. You can use one of the data type values declared in **app/AppDefs.h** as the *type* value, but understand that *type* needn't be globally declared (as constrasted with *app*): The type that you set can be privately meaningful to the application.

If you want to set a file's type so the Browser will take it to be an application, use the value 'BAPP'. The *app* argument, in this case, is ignored (by the Browser, at least).

With regard to icons: The Icon World application lets you create the correspondence between an application and its icon, as well as between the file types that the application recognizes and the icon that's displayed for each type. See "Notes on Developing a Be Application" for more information on Icon World.

Note: In contrast to most of BFile's other functions, **SetTypeAndApp()** and **GetTypeAndApp()** operate properly if the BFile is closed. Moreover, the functions are actually more reliable if the object *is* closed.

Both functions return **B_ERROR** if they fail, **B_NO_ERROR** otherwise. Note that the *app* value (for **SetTypeAndApp()**) isn't checked to make sure that it identifies a recognized application.

Size()

long **Size(**void**)**
long **SetSize(**long *newSize***)**

Size() returns the size of the file's data, in bytes. The BFile needn't be open.

SetSize() sets the size of the file, in bytes. The BFile must be open and writable.

The functions return **B_ERROR** if the BFile's ref hasn't been set, or if the BFile's record has disappeared. In addition, **SetSize()** returns **B_ERROR** if the file isn't open in the proper mode; otherwise it returns **B_NO_ERROR**.

SwitchWith()

long **SwitchWith(**BFile **otherFile***)**

Causes the receiving BFile and the argument object to trade data. The files' records are *not* switched. Both objects must be closed.

SwitchWith() is provided as an efficient way to create a back-up file for files that your application is writing. Here's how you're supposed to use it:

Let's say you've written an application that can open, read, and write files. The user uses your app to open a file called "MyText". You application creates and opens a BFile (**MyTextFile**) that refers to the file. It then allocates a buffer to hold the file's data, and copies the file's data (or as much as it thinks it will need) into the buffer. It also creates a second BFile (**tmpFile**) as a copy of the original (through **CopyTo()**) called "tmp". As the user works, your application occasionally writes the current state of the buffer to the **tmpFile**. When the user tells the application to save, the app closes both files and invokes **SwitchWith()**:

```
MyTextFile->SwitchWith(tmpFile)
```

Your app then re-opens **tmpFile** (which now holds the previously saved version that it just got from **MyTextFile**) and brings it back up to date.

You could get the same result by calling **CopyTo()** (copying from the "tmp" file to the original file) every time the user saves, but the **SwitchWith()** function is much faster.

See also: **CopyTo()**

Write()

long **Write(**const void **data*, long *length***)**

Copies *length* bytes from the *data* buffer into the object's file. The data is copied starting at the data pointer's current position; the existing data at that position (and extending for *length* bytes) is overwritten. The size of the file is increased, if necessary, to accommodate

the new data. When this function returns, the data pointer will point to the first byte that follows the newly copied data.

The function returns the number of bytes that were actually written; except in extremely unusual situations, the returned value shouldn't vary from the value you passed as *length*.

The object must already be open for this function to succeed. If it isn't open, or if, for any other reason, the data couldn't be written, **B_ERROR** is returned.

See also: **Open()**, **Seek()**, **Read()**

BQuery

Derived from: public BObject

Declared in: <storage/Query.h>

Overview

The BQuery class defines functions that let you search for records that satisfy certain criteria. Querying is the primary means for retrieving, or "fetching," records from a database.

Defining a Query

To define a query, you construct a BQuery object and supply it with the criteria upon which its record search will be based. This criteria consists of tables and a predicate:

- The set of tables that you specify restricts the range of candidate records: Only those records that conform to one of the specified tables are considered in the search. The **AddTable()** and **AddTree()** functions tell a BQuery which tables to consider.
- The predicate is a logical test that (typically) compares the value of a particular field (in a record) to a constant value. You can also compare one field's value to another field's value. A predicate is constructed by "pushing" fields, constants, and operators on the BQuery's "predicate stack" (using "reverse Polish notation," as explained in a later section). The predicate is optional.

Let's say you want to find all records in the "People" table that have "age" values greater than 12. The BQuery definition would look like this:

```
/* We'll assume that myDb is a valid BDatabase object. */
BQuery *teenOrMore = new BQuery();
BTable *people = myDb->FindTable("People");

/* Add the table to the BQuery. */
teenOrMore->AddTable(people);

/* Create the predicate. */
teenOrMore->PushField("age");
teenOrMore->PushLong(12);
teenOrMore->PushOp(B_GT);
```

The Table List

A single BQuery, during a single fetch, can search in more than one table. When you call **AddTable()**, the previously added table (if any) isn't bumped out of the table list; instead, the tables accumulate to widen the range of candidate records. However, all BTables that you pass as arguments to **AddTable()** (for a single BQuery) must belong to the same BDatabase object.

Another way to add multiple tables to a query is to use the **AddTree()** function. **AddTree()** adds the table represented by the argument and all tables that inherit from it. Table inheritance is explained in the BTable class specification.

You can't selectively remove tables from a BQuery's table list. If you feel the need to remove tables, you have two choices: You can remove *all* tables (and the predicate) through the **Clear()** function, or you can throw the BQuery object away and start from scratch with a new one.

The Predicate

As mentioned earlier, the BQuery predicate is constructed using "reverse Polish notation" (or "RPN"). In this construction, operators are "post-fixed"; in other words, the operands to an operation are pushed first, followed by the operator that acts upon them. That's why the predicate used in the example, "age > 12", was created by pushing the elements in the order shown:

```
/* Predicate construction for "age > 12" */
teenOrMore->PushField("age");
teenOrMore->PushLong(12);
teenOrMore->PushOp(B_GT);
```

The query operators that you can use are represented by the following constants:

Constant	Meaning
B_EQ	equal
B_NE	not equal
B_GT	greater than
B_GE	greater than or equal to
B_LT	less than
B_LE	less than or equal to
B_AND	logical AND
B_OR	logical OR
B_NOT	negation
B_ALL	wildcard (matches all records)

Except for **B_ALL**, the query operators expect to operate on two previously pushed operands. **B_ALL**, which is used to retrieve all the records in the target tables, should be pushed all by itself (through **PushOp()**).

Complex Predicates

You can create complex predicates by using the conjunction operators **B_AND** and **B_OR**. As with comparison operators, a conjunction operator is pushed after its operands; but with the conjunctions, the two operands are the results of the two previous comparisons (or previous complex predicates).

For example, let's say you want to find the records for people that are between 12 and 36 years old. The programmatic representation of this notion, and its reverse Polish notation, looks like this:

Programmatic expression: ("age" > 12) && ("age" < 36)

Reverse Polish Notation: "age" 12 B_ GT "age" 36 B_LT B_AND

The RPN version prescribes the order of the BQuery function calls:

```
/* Predicate construction for "(age > 12) and (age < 36)" */
teenOrMore->PushField("age");
teenOrMore->PushLong(12);
teenOrMore->PushOp(B_GT);

teenOrMore->PushField("age");
teenOrMore->PushLong(36);
teenOrMore->PushOp(B_LT);

teenOrMore->PushOp(B_AND);
```

Predicates can be arbitrarily deep; the complex predicate shown above can be conjoined with other predicates (simple or complex), and so on.

Fetching

Once you've defined your BQuery, you tell it to perform its search by calling the **Fetch()** function:

```
if (teenOrMore->Fetch() != B_NO_ERROR)
    /* the fetch failed */
```

When it's told to fetch, a BQuery object sends the table and predicate information to the Storage Server and asks it to find the satisfactory records. The winning records (identified by their record IDs) are returned to the BQuery and placed in the BQuery's record ID list, which you can then step through using **CountRecordIDs()** and **RecordIDAt()**:

```
long num_recs = teenOrMore->CountRecordIDs();
record_id this_rec;

for (int i = 0; i < num_recs; i++)
    this_rec = teenOrMore->RecordIDAt(i);
```

To turn the BQuery's record IDs into BRecord objects, you pass the IDs to the BRecord constructor:

```
BList *teens = new BList();
long num_recs = teenOrMore->CountRecordIDs();
record_id this_rec;
BRecord *teen_rec;

for (int i = 0; i < num_recs; i++)
{
    this_rec = teenOrMore->RecordIDAt(i);
    teen_rec = BRecord new(people->Database(), this_rec);
    teens->AddItem(teen_rec);
}
```

Live Queries

By default, a BQuery performs a "one-shot" fetch: Each **Fetch()** call retrieves record IDs, sets them in the BQuery's record ID list, and that's the end of it. Alternatively, you can declare a BQuery to keep working—you can declare it to be "live"—by passing **TRUE** as the argument to the constructor:

```
BQuery *live_q = new BQuery(TRUE);
```

When you tell a live BQuery to fetch, it searches for and retrieves record ID values, just as in the default version, but then the Storage Server continues to monitor the database for you, noting changes to records that would affect your BQuery's results. If the data in a record is modified such that the record now passes the predicate whereas before it didn't, or now doesn't pass but used to, the Server automatically sends messages that will, ultimately, update your BQuery's record list to reflect the change. In short, a live BQuery's record list is always synchronized with the state of the database. But you have to do some work first.

Preparing your Application for a Live Query

It was mentioned above that the Storage Server sends messages to update a live BQuery. The receiver of these messages (BMessage objects) is your application object. In order to get the update messages from your application over to your BQuery, you have to subclass BApplication's **MessageReceived()** function to recognize the Server's messages. Below are listed the messages (as they're identified by the BMessage **what** field) that the function needs to recognize:

what Value	Meaning
B_RECORD_ADDED	A record ID needs to be added to the record list.
B_RECORD_REMOVED	An ID needs to be removed from the list.
B_RECORD_MODIFIED	Data has changed in a record currently in the list.

The only thing your **MessageReceived()** function needs to do to properly respond to a Storage Server message is pass the message along in a call to the Storage Kit's global **update_query()** function, as shown below:

```
#include <Query.h>

void MyApp::MessageReceived(BMessage *a_message)
{
    switch(a_message->what) {
        case B_RECORD_ADDED  :
        case B_RECORD_REMOVED :
        case B_RECORD_MODIFIED :
            update_query(a_message);
            break;
        /* Other app-defined messages go here */
        ...
        default:
            BApplication::MessageReceived(a_message);
            break;
    }
}
```

update_query() finds the appropriate BQuery object and calls its **MessageReceived()** function. The default BQuery **MessageReceived()** implementation handles the **B_RECORD_ADDED** and **B_RECORD_REMOVED** messages by manipulating the record list appropriately. In the case of a **B_RECORD_MODIFIED** message, the BQuery does nothing.

If you want to handle modified records in your application, you can create your own BQuery-derived class and re-implement **MessageReceived()**. To get the identity of the record, you retrieve, from the BMessage, the **long** data named “rec_id”. The following code demonstrates the general look of such a function:

```
/* Re-implementation of MessageReceived() for MyQuery,
 * a BQuery-derived class
 */
void MyQuery::MessageReceived(BMessage *a_message)
{
    record_id rec;

    rec = a_message->FindLong("rec_id");

    switch(a_message->what) {
        case B_RECORD_MODIFIED :
            /* do something with the record */
            break;
        case B_RECORD_ADDED:
        case B_RECORD_REMOVED:
            /* Pass the other two message types to BQuery. */
            BQuery::MessageReceived(a_message);
            break;
    }
...
```

Keep in mind that you don’t have to derive your own class to take advantage of the live query mechanism. Simply getting to the **update_query()** step is enough to keep the your BQuery’s record list up-to-date.

Hook Functions

MessageReceived() Can be overridden to handle live BQuery notifications.

Constructor and Destructor

BQuery()

BQuery(bool *live* = FALSE**)**

Creates a new BQuery object and returns it to you. If *live* is **TRUE**, the BQuery's record list is kept in sync with the state of the database (after the object performs its first fetch). If it's **FALSE**, the database isn't monitored.

See the class description for more information on live BQuery objects.

~BRecord()

~BRecord(void**)**

Frees the memory allocated for the object's record list. If this is a live BQuery, the Storage Server is informed of the object's imminent destruction (so it won't send back any more database-changed notifications).

Member Functions

AddRecordID()

void **AddRecordID(**record_id *id***)**

Tells the BQuery to consider the argument record to be a winner, whether it passes the predicate or not. You call this function *before* you fetch; after the fetch, you'll find that *id* has been added to the record list (and will be monitored, if this is a live query). You can call this function any number of times and so add multiple "predicate-exempt" records, but you can add each specific record only once (duplicate entries are automatically squished to a single representative).

The set of exempt records isn't forgotten after the BQuery performs a fetch. For example, in the following sequence of calls...

```
query->AddRecordID(MyRecord);
query->Fetch();
query->Fetch();
```

... you don't have to "re-prime" the second fetch by re-adding **MyRecord**.

Conversely, **AddRecordID()** doesn't *instantly* add the record to the BQuery's record list: The records that you add through **AddRecordID()** aren't put in the record list until you call **Fetch()**. For example, in this sequence:

```
query->AddRecordID(MyRecord);
query->Fetch();
query->AddRecordID(YourRecord);
```

... **MyRecord** is in **query**'s record list, but **YourRecord** isn't.

Although this isn't the normal way to add records to the list—normally, you define the BQuery's predicate and then fetch records—it can be useful if you want to "fine-tune" the record list. For example, if you want to monitor a particular record through a live query regardless of whether that record passes the BQuery's predicate, you can add it through this function.

Important: Currently, the **AddRecordID()** function is slightly flawed: The records that you add through this function *must* conform to one of the BQuery's tables.

AddTable(), AddTree

void **AddTable(**BTable *a_table**)**
void **AddTree(**BTable *a_table**)**

Adds one or more BTable objects to the BQuery's table list. The first version adds just the BTable identified by the argument. The second adds the argument and all BTables that inherit from it (where "inheritance" is meant as it's defined by the BTable class).

You can add as many BTables as you want; invocations of these functions augment the table list. However, any BTable that you attempt to add must belong to the same BDatabase object.

There's no way to remove BTables from the table list. If you tire of a BTable, you throw the BQuery away and start over.

See also: **CountTables()**, **TableAt()**

Clear()

void **Clear(**void**)**

Erases the BQuery's predicate (the table list and record lists are kept intact). Although this function can be convenient in some cases, it usually better to create a new BQuery for each distinct predicate that you want to test.

CountRecordIDs()

long **CountRecordIDs(**void**)**

Returns the number of records in the BQuery's record list. If the object isn't live, the value returned by this function will remain constant between fetches; if it's live, it may change at any time.

See also: **RecordIDAt()**

CountTables()

long **CountTables(**void**)**

Returns the number of BTables in the BQuery's table list.

See also: **TableAt()**

Fetch(), FetchOne()

long **Fetch(**void**)**
long **FetchOne(**void**)**

Tests the BQuery's predicate against the records in the designated tables (in the database), and fills the record list with the record ID numbers of the records that pass the test:

- **Fetch()** tests all candidate records.
- **FetchOne()** stops after it finds the first winner. This is a convenient function if all you want to do is verify that there is *any* record that fulfills the predicate, or if you know that there's only one.

Note: Currently, **FetchOne()** doesn't—it simply invokes **Fetch()**. Single record fetching will be added in a subsequent release.

The object's record list is cleared before the winning records are added to it.

If the BQuery is live, **Fetch()** turns on the Storage Server's database monitoring; **FetchOne()** doesn't.

Fetching is performed in the thread in which the **Fetch()** function is called; the function doesn't return until all the necessary records have been tested. The on-going monitoring requested by a live query is performed in the Storage Server's thread.

Both functions return **B_NO_ERROR** if the fetch was successfully executed (even if no records were found that pass the predicate); **B_ERROR** is returned if the fetch couldn't be performed.

See also: **RunOn()**

FieldAt()

char ***FieldAt(**long *index***)**

Returns a pointer to the *index*'th field name that you pushed onto the predicate stacked. The pointed-to string belongs to the query—you shouldn't modify or free it. The string itself is a copy of the string that you used to push the field; in other words, the names that are returned by **FieldAt()** are the same names that you used as arguments in previous **PushField()** calls. If *index* is out of bounds, the function returns NULL.

Field names are kept in the order that they were pushed. **FieldAt(**0**)**, for example returns the first field name that you pushed on the stack.

This function is provided, mainly, as an aid to interface design. It's not meant as a diagnostic tool.

FromFlat() *see* ToFlat()

HasRecordID()

bool **HasRecordID(**record_id *id***)**

Returns **TRUE** if the argument is present in the object's record list. Otherwise it returns **FALSE.**

See also: **RecordIDAt()**, **CountRecordIDs()**

IsLive()

bool **IsLive(**void**)**

Returns **TRUE** if the BQuery is live. You declare a BQuery to be live (or not) when you construct it. You can't change its persuasion thereafter.

MessageReceived()

virtual void **MessageReceived(**BMessage **a_message***)*

Invoked automatically by the **update_query()** function, as discussed in "Live Queries" on page 40. You never call this function directly, but you can override it in a BQuery-derived class to change its behavior. The messages it can receive (as defined by their **what** fields) are these:

what Value	Meaning
B_RECORD_ADDED	A record ID needs to be added to the record list.
B_RECORD_REMOVED	A record ID needs to be removed from the list.
B_RECORD_MODIFIED	Data has changed in a record in the list.

The default responses to the first two messages do the right thing with regard to the record list: The specified record ID is added to or removed from the BQuery's record list. The default response to the modified message, however, is to do nothing.

The record that has been added, removed, or modified is identified by its record ID in the BMessage's "rec_id" slot:

```
record_id rec = a_message->FindLong("rec_id");
```

PrintToStream()

void **PrintToStream(**void**)**

Prints the BQuery's predicate to standard output in the following format:

```
arg count = count
    element_type element_value
    element_type element_value
    element_type element_value
    ...
```

element_type is one of "longarg", "strarg", "field", or "op". *element_value* gives the element's value as declared when it was pushed. The order in which the elements are printed is the order in which they were pushed onto the stack.

PushLong(), PushDouble(), PushString(), PushField(), PushOp()

void **PushLong(**long *value***)**
void **PushDouble(**double *value***)**
void **PushString(**const char **string***)**
void **PushField(**const char **field_name***)**
void **PushOp(**query_op *operator***)**

These functions push elements onto the BQuery's predicate stack. The first four push values (or, in the case of **PushField()**, potential values), that are operated on by the operators that are pushed through **PushOp()**.

The **query_op** constants are:

Constant	Meaning
B_EQ	equal
B_NE	not equal
B_GT	greater than
B_GE	greater than or equal to
B_LT	less than
B_LE	less than or equal to
B_AND	logical AND
B_OR	logical OR

B_NOT	negation
B_ALL	wildcard (matches all records)

Predicate construction is explained in "The Predicate" on page 38. Briefly, it's based on the "reverse Polish notation" convention in which the two operands to an operation are pushed first, followed by the operator. The result of an operation can be used as one of the operands in a subsequent operation.

See also: **FieldAt()**

RecordIDAt()

record_id **RecordIDAt(**long *index***)**

Returns the *index*'th record ID in the object's record list. The record list is empty until the object performs a fetch.

See also: **CountRecordIDs()**

RunOn()

bool **RunOn(**record_id *record***)**

Tests the record identified by the argument against the BQuery's predicate. If the record passes, the function returns **TRUE**, otherwise it returns **FALSE**. The record ID *isn't* added to the record list, even if it passes. You use this function to quickly and platonically test records—it isn't as serious as fetching.

See also: **Fetch()**

SetDatabase()

void **SetDatabase(**Database **db***)**

Sets the BQuery's database to the argument. You use this function after you've called **FromFlat()** to tell the BQuery which database it should fetch from when next it fetches. As explained in the **FromFlat()** description, a flattened query doesn't remember the identity of its database.

TableAt()

BTable ***TableAt(**long *index***)**

Returns the *index*'th BTable in the object's table list.

See also: **CountTables()**

ToFlat(), FromFlat()

char ***ToFlat(**long ****size*****)**
void **FromFlat(**char ****flatQuery*****)**

These functions "flatten" and "unflatten" a BQuery's query. **ToFlat()** flattens the query: It transforms the BQuery's table and predicate information into a string. The flattened string is returned directly by **ToFlat()**; the length of the flattened string is returned by reference in the *size* argument.

FromFlat() sets the object's query as specified by the *flatQuery* argument. The argument, unsurprisingly, should have been created through a previous call to **ToFlat()**. Any query information that already resides in the calling object is wiped out.

The one piece of information that isn't translated through a flattened query is the identity of the database upon which the query is based. For flattening and unflattening to work properly, the database of the BQuery that calls **FromFlat()** must match that of the BQuery that flattened the query. You can use the **SetDatabase()** function after calling **FromFlat()** to set the object's database.

You use these functions to store your favorite queries, or to transmit query information between BQuery objects in separate applications.

See also: **SetDatabase()**

BRecord

Derived from: public BObject

Declared in: <storage/Record.h>

Overview

A BRecord represents a *record* in a database. A record is a collection of values that, considered together, describe a single, multi-faceted "thing." The thing that a record describes depends on the *table* to which the record conforms. For example, each record that conforms to the "File" table would describe different attributes of a specific file: its name, size, the directory it's contained in, and so on. A single record can store as much as 32 kilobytes of data (but, to be safe, you should try to keep your records a wee bit smaller than that).

A BRecord object lets you examine and modify the values that are collected in a record. But first, you have to associate the BRecord object with the record that you want to inspect or alter. How you make this association depends on whether you're creating a new record that you wish to add to the database, or retrieving an existing record from the database. These topics are discussed separately in the following sections.

Creating a New Record

You create a new record in reference to a specific table (within a particular database). In your application, you create this reference by passing a BTable object to the BRecord constructor. For example, the following code constructs a BRecord object that conforms to the "Employee" table (the table was created in an example in the BTable class description):

```
/* We'll assume the existence of the a_db BDatabase object. */
BTable *employee_table = a_db->FindTable("Employee");
BRecord *employee_record = new BRecord(employee_table);
```

By conforming to a BTable, a BRecord is given appropriately-sized "slots" that will hold data for each of the *fields* defined by the table. For example, the "Employee" table (as defined in an example in the BTable class description) has three fields:

- The **char *** field "name" names a specific employee.
- The **long** field "extension" identifies the employee's telephone extension.

- The **record_id** field "manager" identifies some other record (possibly in another table) that contains information about the employee's manager (this explained at length later in this class description).

The **employee_record** object, therefore, can accommodate values for these three fields. In a freshly created BRecord, the value for each field is **NULL** (cast as appropriate for the data type of the field).

Important: You must explicitly delete the BRecord objects that you construct in your application. Some of the operations that a BRecord performs (such as committing or removing) might lead you to think that you've "given" the object to the Storage Server, and that you're absolved from the responsibility of destruction. You haven't; you're not.

Setting Data in the BRecord

To put data in a BRecord object, you use its **Set...()** functions; these functions are named for the type of data that they implant:

- **SetLong()** places a long value in the BRecord.
- **SetDouble()** places a double value.
- **SetString()** copies a string.
- **SetRecordID()** places a a **record_id** value.
- **SetTime()** places a double that measures time since January 1, 1970.
- **SetRaw()** copies an arbitrarily long buffer of "raw" data (type **void ***).

Each of these functions designates, as its first argument, the table field that's used to refer to the data. There are two ways to make this designation: by a field's name, or by its *field key* (as defined by the BTable class).

Continuing our employee record example, we begin to put data in the new BRecord by setting data for the "name" and "extension" fields:

```
/* We'll designate the "extension" field by the field's name.
*/
employee_record->SetLong("extension", 123);
```

For variety, we'll set the "name" field by its field key:

```
field_key name_key = employee_table->FieldKey("name");
employee_record->SetString(name_key, "Mingo, Lon");
```

In most cases, there's no difference between the two methods of designating a field (by name or field key); you can use which ever is more convenient. The one instance in which there is a distinction is if you have a table with similarly named fields that are typed differently, in which case, the fields will *only* be distinguishable by field key. For example, if your table has a long field named "info" and a string field that's also named "info", you would distinguish between the fields by using their keys.

Committing a BRecord

The data that you set in a BRecord isn't seen by the database (and so can't be seen by other applications) until you *commit* the data through BRecord's **Commit()** function:

```
record_id mingo_id = employee_record->Commit();
```

The function sends the object's data back to the Storage Server, which places it in the database; the Server creates a new record to hold the data if necessary. The **record_id** value that the function returns uniquely identifies the record within its database (as explained in the next section).

Important: Notice that the BRecord in the example was committed with an "empty" field: The "manager" field hasn't yet been set. Because this is a new record, the value at this field is, by default, **NULL**. Unfortunately, there's no way to distinguish between a default **NULL** and a legitimate **NULL**. For example, if our "Employee" table included a **long** "vacation days" field, the value (for that field) could legitimately be 0—it would look the same as **NULL**. You wouldn't be able to tell if the value was accurate, or if the field hadn't yet been filled in.

Record ID Numbers

A record is identified, within its database, by a record ID number (type **record_id**): Every record in a given database has a different record ID. A BRecord knows the record ID of the record it represents (you can get it through the **ID()** function). But keep in mind that a record ID identifies a record, not a BRecord; thus:

- Before you commit a new BRecord (more accurately, before you commit it for the first time), the object won't have a record ID because it doesn't yet represent a real record.
- More than one BRecord object can point to the same record: They can have the same record ID value, even if the objects are in different applications. Because of this, a record ID number can be passed between applications—in a BMessage, typically—the number will have the same meaning (it will represent the same record) in the other application as it does in yours.

Record ID Fields

One of the features of the **record_id** type is that it can be used to define a table field. Just as you can declare a table field to accept **long** or string data, you can declare a field to take record ID values (through BTable's **AddRecordIDField()** function). Through the use of a record ID field, one record can point to another record. Although the two records must reside in the same database, the two records needn't conform to the same table. In fact, you can't designate, in the field definition, the table that the pointed-to record conforms to.

Returning to the example, the "manager" field in the "Employee" table is typed as a **record_id** field. To set the value for this field in our employee record, we need to find the

record ID of Lon Mingo's manager. This is a job for a BQuery object, as explained in that class.

The Record Ref Structure

The **record_ref** structure is similar to the **record_id** number: It identifies a record in a database. The difference between these two entities is that the **record_ref** structure encodes the record ID *and* the database ID (the ID of the database in which the record resides); the structure's definition is

```
struct record_ref {
    record_id    record;
    database_id  database;
}
```

A record ref (or, simply, "ref") is, therefore, more exacting in its identification of a record than is the record ID. So why would you use a record ID if a ref is more precise?

- Generally speaking, refs are meant to be used in applications that want to access the database but that don't want to worry about the details of tables, queries, and so on. More specifically, refs are used to identify and retrieve items from the file system.
- Record ID's, on the other hand, are the common coin of "real" database applications. For example, the BTable class defines a **SetRecordIDField()**—it doesn't have a function that sets a field that takes a ref. Similarly, BQuery objects retrieve record ID numbers—they don't retrieve refs. If you're using BTables and BQueries, you know which database you're talking to, so you don't need to encode its identity in a cumbersome structure.

Comparing Refs

The **record_ref** structure defines the == and != comparison operators. This allows you to compare two refs as values. For example, the following is legal:

```
record_ref a = a_record->Ref();
record_ref b = b_record->Ref();

if (a == b)
    ...
```

For two refs to be equal, their **database** fields must be the same and their **record** fields must be the same.

Retrieving an Existing Record

In addition to creating new (potential) records for you, the BRecord constructor can retrieve an existing record from a database. To do this, you pass a BDatabase object and record ID to the constructor:

BRecord(BDatabase ***a_database*****,** record_id *record***)**

Typically, you fetch the record ID numbers that BQuery object and tell it which records to fetch. The object retrieves record ID numbers which you then use here to actually get records. (See the BQuery class for information on fetching.)

Data Examination

To examine the data in a BRecord, you ask for the value of a specific field (as defined by the object's BTable). This is accomplished by functions that take this form:

Find*Type***(**field_key *key***)**
Find*Type***(**char ****field_name*****)**

where *Type* is one of the six data types that a field can take (*ergo* **FindLong()**, **FindDouble(), FindRaw()**, **FindRecordID()**, **FindString()**, and **FindTime()**). Each function has two versions so you can designate the field by field key or by name. The functions return the field's data directly. The two pointer-returning functions (**FindRaw()** and **FindString()**) return pointers to data that's owned by the BRecord. You shouldn't try to modify BRecord data by writing to these pointers.

Updating a BRecord

Keep in mind that when you examine a BRecord's data, you're looking at the object's copy of the data that exists in the actual record in the database. If the actual record changes—if a field's value is modified, or if the record itself disappears—such changes are *not* automatically reflected in the BRecord objects that point to the record. ("Live" queries, as explained in the BQuery class, help in this regard, as they inform your application when a change is made.)

If you want to be sure you have the most recent data in your BRecord before you examine it, you should call the **Update()** function. **Update()** re-copies the record's data into your BRecord object. Note, however, that any uncommitted changes that you've made to the BRecord are lost when you update.

Data Modification

Modifying data in a BRecord is also done in reference to specific fields. The suite of modification functions mirrors those for examination, but with an additional argument that specifies the value you want to set:

Set*Type***(**field_key *key***,** *data_type value***)**
Set*Type***(**char **field_name***,** *data_type value***)**

For example, the functions that set **long** data are:

SetLong(field_key *key***,** long *value***)**
SetLong(char **field_name***,** long *value***)**

The changes that you make to the object's data aren't sent back to the database until you call **Commit().** The one exception to this is if you remove the record altogether (through the **Remove()** function). You don't have to call **Commit()** after you call **Remove().**

Extra Fields

In addition to the fields that are defined by its table, a record can contain "extra" fields. Extra fields are created and removed dynamically (through a BRecord object) without affecting the record's table definition. Extra fields are identified by name only, and the data they contain is always untyped (it's considered to be raw).

To add an extra field to a BRecord, you use the **SetExtra()** function. The function takes three arguments: The name of the field that you're adding, the data that you want the field to contain, and the length of the data. For example, here we add two extra fields to the employee record:

```
employee_record->SetExtra("motto",
                          (const void *)"I Am Mingo",
                          strlen("I Am Mingo"));

long age = 35;
employee_record->SetExtra("age",
                          (const void *)&age,
                          sizeof(long));
employee_record->Commit();
```

To find the data in an extra field, you pass the name of the field to **FindExtra()**. The function returns a pointer to the data as it lies in the BRecord object—it doesn't copy the data. The function also returns the amount of data (in bytes) that the extra field is storing:

```
char *m_ptr, *motto;
long *a_ptr, age, size;

if ((m_ptr = (char *)FindExtra("motto", &size)) == NULL)
    ...
else {
    motto = malloc(size+1); /* Add 1 for the NULL */
    strncpy(motto, m_ptr, size);
    motto[size] = '\0';
}

if ((a_ptr = (long *)FindExtra("age", &size) == NULL)
    ...
```

```
else
    age = *a_ptr;
```

BRecord supplies the function **GetExtraInfo()** so you can discover the names and sizes of a record's extra fields. The function takes an index (*n*) as its initial argument and returns a pointer to the *n*'th extra field's name and the size of the field in its second and third arguments. It returns **B_ERROR** if the index is out-of-bounds. Here, we use the function to iterate over all the extra fields in a record:

```
char *name;
long size;
long i = 0;

while (employee_record->GetExtraInfo(i++, &name, &size)
            != B_ERROR)
    printf("Extra Field Name: %s; Size %d\n", name, size);
```

A single record can contain any number of extra fields; the only restoration is that they all must have different names. If you call **SetExtra()** using a name that already exists, the existing data is removed and the new data is installed. On the other hand, an extra field *can* have the same name as a field in the record's table: The BRecord object keeps the two sets of fields separate, so it won't get confused.

Since extra fields aren't part of a table definition, you can't declare them to be indexed (as the term is used in the BTable class), and you can't use them in a query predicate (see the BQuery class).

Constructor and Destructor

BRecord()

BRecord(BDatabase *****database***, record_id *id***)**
BRecord(record_ref *ref***)**
BRecord(BTable *****table***)**
BRecord(BRecord *****record***)**

Creates a new BRecord object and returns it to you.

The first version of the constructor (the BDatabase and **record_id** version) is used to acquire the record with the given ID from the specified database. The second version does the same, but encodes the database and record identities as a single **record_ref** value.

The second version (BTable) constructs a BRecord that can accommodate values for the fields that are declared in its BTable argument.

The third version copies the data from the argument BRecord into the new BRecord (including the ref value).

You should follow a call to the constructor with a call to **Error()** to make sure the specified record was found or created; the function returns **B_ERROR** for failure and **B_NO_ERROR** for success.

See also: **Error()**

~BRecord()

~BRecord(void**)**

Frees the memory allocated for the object's copy of the database data. The object is *not* automatically committed by the destructor; if there are uncommitted changes, you must explicitly commit them or they'll be lost.

Note that you are responsible for deleting the BRecords that you've constructed. When you commit or remove a record (when you call **Commit()** or **Remove()**), you're *not* giving the object to the Server.

Member Functions

Commit()

record_id **Commit(**void**)**

Sends the BRecord's data back to the database. The function returns the **record_ref** of the record that the object represents. It does this as a convenience for new records, which will be receiving fresh ref numbers; "old" records (records that were previously retrieved from the database) don't change ref values when they're committed.

You should call **Error()** immediately after calling **Commit()** to see if the operation was successful (**B_NO_ERROR**). It will fail (**B_ERROR**) if the ref isn't valid, if the record has been locked by some other object, or if some other obstacle bars the path of ingress.

See also: **Lock()**, **Update()**

Database()

BDatabase ***Database(**void**)**

Returns the BDatabase object that represents the database that owns the table that defines the record that killed the cat that ate the rat that's represented by this BRecord.

Error()

long **Error(**void**)**

Returns an error code that symbolizes the success of the previous call to certain other functions. The following functions set the code that's returned here:

- the BRecord constructor
- **Commit()**
- **Update()**
- **FindLong(), FindString(), ...**
- **SetLong(), SetString(), ...**
- **Remove()**

In all cases, a return from **Error()** of **B_NO_ERROR** means that the previous call was successful; **B_ERROR** means it failed.

After **Error()** returns the error code is automatically reset to **B_NO_ERROR.**

FindDouble() FindLong(), FindRaw(), FindRecordID(), FindString(), FindTime()

double **FindDouble(**char ***field_name****)**
double **FindDouble(**field_key *key****)**

long **FindLong(**char ***field_name****)**
long **FindLong(**field_key *key****)**

void ***FindRaw(**char ***field_name**,** long ***size****)**
void ***FindRaw(**field_key *key**,** long ***size****)**

record_id **FindRecordID(**char ***field_name****)**
record_id **FindRecordID(**field_key *key****)**

const char ***FindString(**char ***field_name****)**
const char ***FindString(**field_key *key****)**

double **FindTime(**char ***field_name****)**
double **FindTime(**field_key *key****)**

These functions return the value of the designated field in the BRecord. None of these functions check to make sure you're returning the value in an appropriate data type, nor do they perform any type conversion.

FindRaw() and **FindString()** return pointers to data that's owned by the object. If you want to manipulate or store the data, you must copy it before deleting the object. The **FindRaw()** functions also return, by reference in *size*, the amount of data that it points to.

You should always check **Error()** after calling one of these functions to make sure the call was successful. The usual culprit, in a failure, is an illegitimate field specification. Asking for the value of a non-existing field, for example, will fail.

There is a subtle difference between the field name and field key versions of these functions: If you ask for a value by field name, the data type given by the selected function is used to locate the correct field. For example, if the "age" field stores **long** data but you ask for its value as a string ...

```
char *ageString = FindString("age");
```

... the function won't be able to find a string-valued "age" field and so will fail (**Error()** will return **B_ERROR**). The analogous request by field key:

```
char *ageString = FindString(a_table->FieldKey("age"));
```

won't appear to fail (**Error()** returns **B_NO_ERROR**), even though it will return something awful.

See also: **SetDouble()**

FindExtra() *see* SetExtra()

GetExtraInfo() see SetExtra()

IsNew()

bool **IsNew(**void**)**

Returns **TRUE** if the object was constructed to represent a new record, and hasn't yet been committed.

See also: the BRecord constructor

Ref()

record_ref **Ref(**void**)**

Returns the **record_ref** structure of the BRecord's record. This structure uniquely identifies the record across all databases. This function always returns a **record_ref** value, even if the BRecord has never been committed (in which case the structure's **record** field will be -1).

Remove()

void **Remove(**void**)**

Removes the BRecord's record from the database. The success of the removal is reported in the value returned by **Error()** (**B_NO_ERROR** if the record was removed, **B_ERROR** if it wasn't).

RemoveExtra() *see* SetExtra()

SetDouble(), SetLong(), SetRaw(), SetRecordID(), SetString(), SetTime()

void **SetDouble(**char ***field_name*, double *value***)**
void **SetDouble(**field_key *key*, double *value***)**

void **SetLong(**char ***field_name*, long *value***)**
void **SetLong(**field_key *key*, long *value***)**

void **SetRaw(**char ***field_name*, void ***ptr*, long *size***)**
void **SetRaw(**field_key *key*, void ***ptr*, long *size***)**

void **SetRecordID(**char ***field_name*, record_id *value***)**
void **SetRecordID(**field_key *key*, record_id *value***)**

void **SetString(**char ***field_name*, char ***ptr***)**
void **SetString(**field_key *key*, char ***ptr***)**

void **SetTime(**char ***field_name*, double *value***)**
void **SetTime(**field_key *key*, double *value***)**

Sets the value of the designated field to the value given by *value*. These functions don't perform type checking or type conversion. (See **FindDouble()** for more information on fields and types; the rules described there apply here.)

SetRaw() and **SetString()** copy the data that's pointed to by their *ptr* arguments. The **SetString()** pointer must point to a **NULL**-terminated string. You specify amount of data (in bytes) that you want **SetRaw()** to copy through the function's *size* argument. Keep in mind that you can only store 32 kilobytes of data in a single record (in all its fields combined).

To gauge the success of the modification, check the value returned by **Error()**. If the field's value was successfully set, **Error()** returns **B_NO_ERROR**; otherwise it returns **B_ERROR**.

The value-setting functions don't affect the actual record that the BRecord represents: When you call a **Set***Type***()** function, you're modifying the BRecord's copy of the data, not the actual data that lives in the database. This means that you're able to successfully call these function if the record is locked, and if the BRecord doesn't (yet) have a ref (conditions under which many other functions fail). To write your change to the database, you call BRecord's **Commit()** function.

Keep in mind that a subsequent **Lock()** call will wipe out the (uncommitted) changes that you've made through these functions. This is an important point since many applications will want to lock before committing. If you plan on locking, you should do so *before* using these functions. In other words:

```
/* Lock, modify, commit, unlock. */
a_record->Lock();

a_record->SetLong("age", 9);
a_record->SetString("name", "Emma");
```

```
...
a_record->Commit();
a_record->Unlock();
```

See also: **FindLong()**

SetExtra(), FindExtra(), RemoveExtra(), GetExtraInfo()

void **SetExtra(**const char **name*, const void **data*, long *dataLength***)**

void ***FindExtra(**const char **name*, long ***dataLength***)**

void **RemoveExtra(**const char **name***)**

long **GetExtraInfo(**long *index*, char ***name*, long ***dataLength***)**

These functions add, query, and remove the BRecord's "extra" fields. Extra fields can be added and removed dynamically; they aren't part of the definition of the table to which the record conforms. Extra fields are identified by names and can hold an arbitrary amount of untyped data. The names of a record's extra fields must be unique among themselves, but can be the same as the record's "normal" (table-defined) fields. For examples of these functions, see "Extra Fields" on page 52.

SetExtra() creates a new extra field named name, or replaces the existing so-named field. The data that the field holds is copied from *data*; the amount of data to copy is given by *dataLength*. The extra data that you add through this function must be committed (through the **Commit()** function) just like "normal" data.

FindExtra() finds the field named *name* and returns a pointer to its data (directly). The length of the data is passed back by reference through the *dataLength* argument. Keep in mind that the function gives you a pointer to data that's owned by the BRecord. You shouldn't modify or free the pointer. If **FindExtra()** can't find the named field, it returns **NULL**.

RemoveExtra() removes the named field.

GetExtraInfo() retrieves information about the *index*'th extra field: A pointer to the field's name is returned in **name*, and the length of the field's data is returned in **dataLength*. You shouldn't allocate **name* before passing it in—the pointer that's passed back points into the BRecord itself. By the same token you mustn't free or modify **name*. If *index* is out of bounds, **GetExtraInfo()** returns **B_ERROR**, and sets **name* to point to **NULL**. Otherwise, it returns **B_NO_ERROR**.

Table()

BTable ***Table(**void**)**

Returns the BTable to which the BRecord conforms.

Update()

void **Update(**void**)**

Copies the record's data from the database into the BRecord. Any uncommitted changes you have made to the data that's currently held by the BRecord will be lost. The success of the update is reported by the value returned by the **Error()** function (**B_NO_ERROR** means success; **B_ERROR** indicates failure).

BResourceFile

Derived from: public BFile

Declared in: <storage/ResourceFile.h>

Overview

The BResourceFile class defines structured files that contain a collection of data entries, or *resources*. A single resource file can hold an unlimited number of resources; a single resource within a resource file can contain an unlimited amount of data.

Creating a Resource File

A resource file (as it lies on the disk) is tagged with an identifying header that distinguishes it (the file) from "plain" files. The distinction between a resource file and a plain file is important: Although you can (inadvertently, one assumes) refer a BResourceFile object to a plain file, you won't be able to use the object to open the file. Simply referring a BResourceFile object to an existing plain file will *not* transform the file into a resource file.

To create a new resource file—to create a file that's given a resource header—you pass a pointer to an allocated BResourceFile object to BDirectory's **Create()** function:

```
BResourceFile rFile;
aDirectory->Create("NewFile", &rFile);
```

You can also create a new resource file by copying an existing resource file through BFile's **CopyTo()** function.

The only files that are automatically created (by the system) as resource files are executables: All applications and programs have the capacity to store resources.

Accessing Resource Data

After you've created (or otherwise obtained) a resource file, you open the BResourceFile object that refers to it through the **Open()** function (inherited from BFile), and then use the *Manipulate***Resource()** functions (**AddResource()**, **RemoveResource()**, and so on) defined by the BResourceFile class to examine and manipulate the file's contents. Each of the resource-affecting functions performs its magic on one resource at a time.

BResourceFile doesn't disqualify BFile's **Read()** and **Write()** functions—but you shouldn't use them. These functions will read and write the resource file as flat data, as if it were a plain file. It's your file, but this probably isn't what you want. (To be a bit less prohibitive, reading a resource file is safe and might be slightly informative).

When you've had enough of manipulating resources (and not **Write()**-ing them), you should close the resource file, through the inherited **Close()** function.

Identifying a Resource within a Resource File

A single resource within a resource file is tagged with a data type, an ID, and a name:

- The data type is one of the Application Kit-defined types (**B_LONG_TYPE**, **B_STRING_TYPE**, and so on) that characterize different types of data. The data type that you assign to a resource doesn't restrict the type of data that the resource can contain, it simply serves as a way to label the type of data that you're putting into the resource so you'll know how to cast it when you retrieve it.
- The ID is an arbitrary integer that you invent yourself. It need only be meaningful to the application that uses the resource file.
- The name is optional, but can be useful: You can look up a resource by its name, if it has one.

Taken singly, none of these tags needs to be unique: Any number of resources (within the same file) can have the same data type, ID, or name. It's the *combination* of the data type constant and the ID that uniquely identifies a resource within a file. The name, on the other hand, is more of a convenience; it never needs to be unique when combined with the data type or with the ID.

Data Format

All resource data is assumed to be "raw": If you want to store a **NULL**-terminated string in a resource, for example, you have to write the **NULL** as part of the string data, or the application that reads the resource from the resource must apply the **NULL** itself. Put more generally, the data in a resource doesn't assume any particular structure or format, it's simply a vector of bytes.

Data Ownership

The resource-manipulating functions cause data to be read from or written to the resource file directly and immediately. In other words, the BResourceFile object doesn't create its own "resource cache" that acts as an intermediary between your application and the resource file. This has a couple of implications:

- Resource data that you retrieve from or write to a BResourceFile object belongs to your application. For example, the data that's pointed to by the **FindResource()**

function is allocated by the object for you—it's your responsibility to free the data when your finished with it. Similarly, the data that you pass to **AddResource()** (to be added as a resource in the file) must be freed by your application after the function returns.

- The individual changes that you make to the resources are visible to other BResourceFiles (that are open on the same file) immediately as they are made. You can't, for example, bundle up a bunch of changes and then "commit" them all at the same time.

Constructor and Destructor

BResourceFile()

BResourceFile(void**)**
BResourceFile(record_ref *ref***)**

The BResourceFile constructor creates a new object and returns a pointer to it. You can set the object's ref by passing it as an argument here; without the argument, the object won't refer to a file—it will be essentially useless—until the ref is set. The methods by which you set (or re-set) an unreferenced BResourceFile's ref are the same as for a BFile:

- **BStore::SetRef()**
- **BFile::CopyTo()**
- **BDirectory::Create()**
- **BDirectory::GetFile()**

You can refer a BResourceFile object to any file; that is, you're *allowed* to do so. However, only those BResourceFile objects that refer to actual resource files are allowed to be opened—the **Open()** function will fail if the BResourceFile refers to a plain file.

Simply pointing the ref to a random file will not convert the file so that it can hold resources. Resource files can only be created by passing a BResourceFile object to BDirectory's **Create()** function, or by copying an existing resource file through **CopyTo()**.

~BResourceFile()

virtual **~BResourceFile(**void**)**

Destroys the BResourceFile object; this *doesn't* remove the file that the object corresponds to (to remove a file, use BDirectory's **Remove()** function). The object is automatically closed (through a call to **Close()**) before the object is destroyed.

Member Functions

AddResource()

long **AddResource(**ulong *type*,
long *id*,
void **data*,
long *dataLength***,**
const char **name* = NULL**)**

Adds a new resource to the file. For this function to have an effect, the file must be open for writing. The arguments are:

- *type* is one of the data type constants defined by the Application Kit (**B_LONG_TYPE**, **B_STRING_TYPE**, and so on).
- *id* is the ID number that you want to assign to the resource. The value of the ID has no meaning other than that which you application imbues it with; the only restriction on the ID is that the combination of it and the data type constant must be unique across all resources in this resource file.
- *data* is a pointer to the data that you want the resource to hold.
- *dataLength* is the length of the *data* buffer, in bytes.
- *name* is optional, and needn't be unique. Or even interesting.

Ownership of the *data* pointer isn't assigned to the BResourceFile object by this function; after **AddResrouce()** returns, your application can free or otherwise manipulate the buffer that *data* points to without affecting the data that was written to the file.

If the combination of *type* and *id* is already being used by a resource in this BResourceFile, or if, for any other reason, the resource data couldn't be written to the file, the function returns **B_ERROR**. Otherwise, it returns **B_NO_ERROR**.

Warning: Currently, **AddResource()** *will* write over an existing resource. In this case, the function returns a positive integer (specifically, it returns the number of bytes that it just wrote), but it *doesn't* change the name of the resource. For now, you should call **RemoveResource()** just before calling **AddResource()**, passing the same *type* and *id* arguments to both functions.

See also: **WriteResource()**

FileCreated()

virtual long **FileCreated(**void**)**

FileCreated() is a hook function, defined by BFile, that's called when a new file is created. BResourceFile implements **FileCreated()** to put a magic number at the top of the resource file. If you derive a class from BResourceFile and implement your own version of

FileCreated(), you should call BResourceFile's version of the function before performing your own initializations.

FindResource()

void ***FindResource(**ulong *type*,
long *id*,
void **dataLength***)*

void ***FindResource(**ulong *type*,
const char **name*,
void **lengthFound***)*

Finds the resource identified by the first two arguments, and returns a pointer to a copy of the resource's data. The size of the data, in bytes, is returned by reference in **lengthFound.*

It's the caller's responsibility to free the pointed-to data.

If the first two arguments don't identify an existing resource, **NULL** is returned.

See also: **ReadResource()**

GetResourceInfo()

bool **GetResourceInfo(**long *byIndex*,
ulong **typeFound*,
long **idFound*,
char ***nameFound*,
long **lengthFound***)*

bool **GetResourceInfo(**ulong *byType*,
long *andIndex*,
long **idFound*,
char ***nameFound*,
long **lengthFound***)*

bool **GetResourceInfo(**ulong *byType*,
long *andId*,
char ***nameFound*,
long **lengthFound***)*

bool **GetResourceInfo(**ulong *byType*,
char **andName*,
long **idFound*,
long **lengthFound***)*

These functions return information about a specific resource, as identified by the first one or two arguments:

- The first version (*byIndex*) searches for the n'th resource in the file.

- The second (*byType* plus *andIndex*) searches for the n'th resource that has the given type.
- The third (*byType* plus *andId*) looks for the resource with the unique combination of type and ID.
- The third (*byType* plus *andName*) looks for the first resource that has the given type and name.

The other arguments return the other statistics about the resource (if found). The pointer that's returned in **foundName* belongs to the BResourceFile. Don't free it.

The functions return **TRUE** if a resource was found, and **FALSE** otherwise.

HasResource()

bool **HasResource(**ulong *type*, long *id***)**

bool **HasResource(**const char **name*, ulong *type***)**

Returns **TRUE** if the resource file contains a resource as identified by the arguments, otherwise it returns **NOPE**.

Keep in mind that there may be more than one resource in the file with the same *name* and *type* combination. The *type* and *id* combo, on the other hand, is unique.

ReadResource()

long **ReadResource(**ulong *type*,
long *id*,
void **data*,
long *offset***,**
long *dataLength***)**

Reads data from an existing resource (identified by *type* and *id*) and places it in the *data* buffer. *offset* gives the location (measured in bytes from the start of the resource data) from which the read commences, and *dataLength* is the number of bytes you want to read. The *data* buffer must already be allocated and should be at least *dataLength* bytes long.

You can ask for more data than the resource contains; in this case, the buffer is filled with as much resource data as exists (or from *offset* to the end of the resource). However, note well that the function *doesn't* tell you how much data it actually read.

The function returns **B_ERROR** if the buffer is only partially filled, or if the resource wasn't found. Otherwise, it returns **B_NO_ERROR**.

See also: **FindResource(), WriteResource()**

RemoveResource()

long **RemoveResource(**ulong *type*, long *id***)**

Removes the resource identified by the arguments. The function returns **B_NO_ERROR** if the resource was successfully removed, and **B_ERROR** otherwise.

WriteResource()

long **WriteResource(**ulong *type*,
long *id*,
void **data*,
long *offset***,**
long *dataLength***)**

Writes data into an existing resource, possibly overwriting the data that the resource currently contains. The *type* and *id* arguments identify the target resource; this resource must already be present in the file—**WriteResource()** doesn't create a new resource if the *type*/*id* combination fails to identify with a winner.

data is a pointer to the new data that you want to place in the resource; *dataLength* is the length of the data buffer. *offset* gives the location at which you want the new data to be written; the offset is taken as the number of bytes from the beginning of the existing resource data. If the new data is placed such that it exceeds the size of the current resource data, the resource grows to accommodate the new data.

You can't use this function to "shrink" a resource. To remove a portion of data from a resource, you have to remove the resource and then re-add it.

If *type* and *id* don't identify an existing resource, of if the data couldn't be written, for whatever reason, this function return **B_ERROR**. Otherwise, it returns **B_NO_ERROR**.

See also: **AddResource()**

BStore

Derived from: public BObject

Declared in: <storage/Store.h>

Overview

BStore is an abstract class that defines common functionality for its two subclasses, BFile and BDirectory. You never construct direct instances of BStore, nor does the Storage Kit "deliver" such BStore instances to your application. The BStore objects that you work with will always be instances of BFile or BDirectory (or from a class derived from these).

Furthermore, you shouldn't derive your own classes directly from BStore. If you want to create your own file class, you should derive your class from BFile (or, possibly, BResourceFile). Y

Note: Throughout this class description, the terms "file" and "item" are used generically to mean an actual item in a file system. The characteristics ascribed to files (in the following) apply to directories as well.

Files, Records, and BStores

Every file in the file system has a database record associated with it. The record contains information about the file, such as its name, when it was created, the directory it lives in, and so on. All file system activities are performed on the basis of these "file records." For example, if you want to locate a file, you have to locate the file's record; passing the record (albeit indirectly, as described below) to a BStore causes the object to "refer to" the file on disk. Until the object is referred to a file, it's abstract and useless.

A BStore's record is established through a *record ref*. A record ref (or, simply, *ref*) is a structure of type **record_ref** that uniquely identifies a record across all databases by listing the record's ID as well as the ID of its database:

```
struct record_ref {
    record_id record;
    database_id database;
}
```

The nicety of the ref is that it bundles up all the database information that a BStore needs, allowing your application to ignore the details of database organization.

Note: Record refs aren't used only to identify records that describe files. A record ref is simply a means for a identifying a record, regardless of what that record signifies.

How to Set a Ref

BStore's **SetRef()** function sets the calling object's ref directly. This function is often used in an implementation of BApplication's **RefsReceived()** hook function. **RefsReceived()** is invoked automatically when a ref is sent to your application in a BMessage. For example, when the user drops a file icon on your application, your application receives the ref of the file through a **RefsReceived()** notification.

In a typical implementation of **RefsReceived()**, you would ask the ref if it represents a file or directory, allocate a BFile or BDirectory accordingly, and then pass the ref to the object in an invocation of **SetRef()**. An example of this is given in the description of the **does_ref_conform()** function, in the section "Global Functions, Constants, and Defined Types" on page 99.

SetRef() isn't the only way to refer an object to a file. The most important of the other functions that perform this feat are listed below:

- BDirectory's **GetFile()** sets the ref for its BFile argument. The function refers the object to a file based on the file's name, or index within the directory. **GetDirectory()** performs an analogous reference for a BDirectory argument.
- BStore's **GetParent()** sets the argument BDirectory to refer to the calling object's "parent" directory. This is the directory that contains the file that the object refers to.
- BVolume's **GetRootDirectory()** refers its BDirectory argument to the BVolume's *root directory.* This is the "starting-point" directory in the volume's file system.

Using these functions, you can traverse an entire file system: Given a BVolume object, you can descend the file system by calling **GetRootDirectory()**, and then iteratively and recursively calling **GetFile()** and **GetDirectory()**. Given a BFile or BDirectory, you can ascend the hierarchy through recursive calls to **GetParent()**.

An example of file system browsing, and a discussion of the file system hierarchy is given in the description of the BDirectory class.

Altering the File System

Continuing the list of ref-setting functions, the following group of Storage Kit functions set refs as side-effects of altering the structure of the file system:

- BDirectory's **Create()** adds a new file to the file system. The BFile (or BDirectory) that you pass to the function is referred to the new file (or directory).

- **Remove()**, also defined by BDirectory, removes, from the file system, the file referred to by the argument. This effectively "unsets" the argument object's ref.
- BStore's **MoveTo()** moves the calling object's file to a new parent directory.
- BFile's **CopyTo()** copies the calling object's file and sets the ref of the argument BFile to refer to the copy. Note that you can only copy files—you can't copy directories.

Passing Files to Other Threads

A file's ref acts as a system-wide identifier for the file. If you want to "send" a file to some other application, or to another thread in your own application—in other words, if you want more than one process to operate asynchronously on the same file—you should communicate the identity of the file by sending its ref. The thread that receives the ref would construct its own BStore object and call **SetRef()**, in the manner of the **RefsReceived()** function, described earlier.

You can't retrieve a BStore's ref directly from the object. Instead, you retrieve the object's record (through the **Record()** function) and then retrieve the ref from the record (through BRecord's **Ref()** function). The example below demonstrates this as it prepares a BMessage to hold a ref that's sent another application:

```
/* 'zapp' is the signature of the app that we want to send the
 * ref to.
 */
BMessenger *msngr = new BMessenger('zapp');

/* By declaring the BMessage to be a B_REFS_RECEIVED command,
 * the message will automatically show up (when sent) in the
 * other app's RefsReceived() function.
 */
BMessage *msg = new BMessage(B_REFS_RECEIVED);

/* Retrieve the ref from aFile (which is assumed to be
 * an extant BFile object).
 */
record_ref fileRef = aFile->Record()->Ref();

/* Add the ref to the BMessage and send it.  */
msg->AddRef("refs", fileRef);
msngr->SendMessage(msg);
```

Custom Files

It's possible to "customize" your files by, providing them with "custom" records. To do this you need to understand a little bit about the database side of the Storage Kit. Before continuing here, you should be familiar with the BRecord and BTable classes.

When you create a new file, a record that represents the file is automatically created and added to the database. The table to which this record conforms depends on whether the file is, literally, a file (as opposed to a directory): If it's a file, the record conforms to the "File" table; if it's a directory, it conforms to "Folder." (Resource files, as described in the BResourceFile class, also conform to "File".)

The **Create()** function, defined by BDirectory, lets you declare (by name) a table of your own design as the table to which the new file's record will conform. The only restriction on the table is that it should inherit (in the table-inheritance sense) from either "File" or "Folder" as the item that you're creating is a file or a directory.

By creating and using your own "file tables," you can augment the amount and type of information that's kept in a file's record. In the example shown below, a "Image File" table is defined and used to create a new file:

```
/* The BDatabase object aDB is assumed to exist. */
BTable *ImageTable = aDB->CreateTable("Image Table", "File");

SoundTable->AddLongField("Height");
SoundTable->AddLongField("Width");
SoundTable->AddStringField("Description");

/* Create a new "image file."  The BDirectory object aDir
 * is assumed to exist.
 */
BFile myImageFile;
aDir->Create("Bug.image", &myImageFile, "Image Table");
```

Tables, remember, are defined for specific databases; the **ImageTable** definition shown here is defined for the **aDB** database. Similarly, a directory is part of a specific file system. If you designate a table when creating a new file, the table's database and the directory's file system must belong to the same volume. Put programmatically, the database and directory objects used above must be related thus:

```
aDB->Volume() == aDir->Volume()
```

Adding Data to a File Record

To add data to a file's record, you get the record through BStore's **Record()** function, and then call BRecord's data-adding functions. For example:

```
BRecord *myImageRec = myIageFile->Record();

myImageRec->SetLong("Height", 256);
myImageRec->SetLong("Width", 512);
myImageRec->SetString("Description", "Bug squish");
myImageRec->Commit();
```

The **Commit()** call at the end of the example is essential: If you change a file's record directly, you must commit the changes yourself (but see "The Store Creation Hook" on page 73 for an exception to this rule).

File Record Caveats

If you create and use your own file records, heed the following:

- *You may only change those fields that were added through your table.* Because of table-inheritance, your file records will contain a number of fields that were defined by the "File" or "Folder" tables. Don't touch these fields. They don't belong to you.
- *Don't mix BRecord function calls with BStore function calls.* Almost all the BStore (and BFile and BDirectory) functions update the file's record (they call BRecord's **Update()**). If you're in the middle of altering the BRecord and then call a seemingly innocuous function—**GetName()**, for example—you'll lose the BRecord changes that you've made. You must call BRecord's **Commit()** after making BRecord changes and before you make subsequent BStore calls.

The Store Creation Hook

In some cases, you may want to change a new file's record before the file becomes "public." Normally, when you call BDirectory's **Create()** function, the system creates a record for the file, fills in the fields that it knows about (in other words, it fills in the fields that belong to the "File" or "Folder" table), commits the record, and then returns the new BFile (or BDirectory) to you. (This would be the natural order of things in the example shown above.)

The important point here is that the record is committed *before* you get a chance to touch the fields that you're interested in. If some application has a *live query* running (as defined by the BQuery class), the incompletely filled-in record—which will be a candidate for the query from the time that it's committed by the system—may inappropriately pass the query.

To give you access to the record before it's committed, **Create()** lets you pass a *store creation hook* function as an optional (fourth) argument. Such a function assumes the following protocol:

long ***store_creation_hook_name*(**BStore **item*, void **hookData***)**

Note that this is a global function; the store creation hook can't be declared as part of a class. Also, although **store_creation_hook** is declared (in **Store.h**) as a **typedef,** the declaration is intended to be seen for its protocol only: You can't declare a function as a **store_creation_hook** type.

The store creation hook is called just after the file's record is created, but before it's committed. The first argument is a BStore object that represents the new file. The record changes shown in the previous example would be performed in a store creation hook thus:

```
/* Define a store creation hook function. */
long imageFileHook(BStore *item, void *hookData)
{
    BRecord *myImageRec = item->Record();
```

```
    myImageRec->SetLong("Height", 256);
    myImageRec->SetLong("Width", 512);
    myImageRec->SetString("Description", "Bug squish");
    return B_NO_ERROR;
}
```

Note that you *don't* commit record changes that you make in a store creation hook. They'll be committed for you after the function returns. If the hook function returns a value other than **B_NO_ERROR**, the store creation is aborted (by the **Create()** function).

The **Create()** call with this hook function would look like this:

```
aDir->Create("Bug.image", &myImageFile, "Image Table",
             imageFileHook);
```

Other Hook Providers

All Storage Kit functions that create files provide a store creation hook mechanism. These are:

- **BFile::CopyTo()**
- **BDirectory::Create()**
- **BStore::MoveTo()**

The details of the mechanism as demonstrated by the **Create()** examples shown here apply without modification to the other functions as well.

Hook Data

You can pass additional data to your hook function by supplying a buffer of **void *** data as the **Create()** function's final argument. This "hook data" is passed as the second argument to the hook function. Here, we redefine the hook function used above to accept an image description string as hook data:

```
/* Define a store creation hook function. */
bool imageFileHook(BStore *item, void *hookData)
{
    BRecord *myImageRec = item->Record();

    myImageRec->SetLong("Height", 256);
    myImageRec->SetLong("Width", 512);
    myImageRec->SetString("Description", (char *)hookData);
    return TRUE;
}
```

And here we call **Create()**, passing it some hook data:

```
aDir->Create("Bug.image", &myImageFile, "Image Table",
             iamgeFileHook, (void *)"Bug squish");
```

Hook Function Rules

The rules that govern the use and implementation of a store creation hook are similar to those you follow when, in general, you modify a BStore's record.

- The store creation hook mechanism is provided *exclusively* so you can get to your own table fields in a new file's record. You mustn't use it for any other purpose—you mustn't set fields that you didn't define, or alter the new BStore in any way.
- Within the implementation of a store creation hook function, the *only* BStore function that you can call is **Record().**

Constructor and Destructor

BStore()

protected:

BStore(void**)**

The BStore constructor is protected to prevent you from creating direct instances of the class.

~BStore()

virtual **~BStore(**void**)**

Although the BStore is public, you can't actually use it. Since you can't construct a BStore object, you'll never have the opportunity to destroy one.

Member Functions

CreationTime(), ModificationTime(), SetModificationTime()

long **CreationTime(**void**)**
long **ModificationTime(**void**)**
long **SetModificationTime(**const long *time***)**

The first two functions return the time the referred-to item was created and last modified, measured in seconds since January 1, 1970. To convert the time value to a string, you can use standard-C function **strftime()** or **ctime()** (as declared in **time.h**). If the object doesn't refer to a file (or directory), the functions return **B_ERROR**.

SetModificationTime() lets you set the modification time for the item. The function returns **B_ERROR** if the object's ref isn't set, if the item lives in a read-only file system, or if the modification time couldn't otherwise be set.

And a very special note to all you BFile users: These three functions work regardless of the open state of the target object.

Error()

int **Error(**void**)**

Returns an error code that indicates the success of the previous BStore function call. The possible codes are:

- **B_ERROR**; the requested operation couldn't be performed, typically because the object isn't valid.
- **B_NAME_IN_USE**; this code is returned if, in an immediately preceding **SetName()** call, you attempted to set the item's name to one that identifies an existing item.
- **B_NO_ERROR**; the previous call succeeded.

The **Error()** function *doesn't* record the success of the BStore operators.

GetName()

long **GetName(**char ***name***)**

Copies the BStore's name into *name*. You must allocate the argument before you pass it in. File names are never longer than the constant **B_FILE_NAME_LENGTH**; to be safe, *name* should be at least that long. It's the caller's responsibility to free the *name* buffer.

If the BStore doesn't refer to a file, this returns **NULL** and sets the **Error()** code to **B_ERROR**.

See also: **SetName()**

GetParent()

long **GetParent(**BDirectory ***parent***)**

Sets the argument's ref to the directory that contains this BStore. You must allocate the argument before you pass it to the function; it's the caller's responsibility to delete the argument object.

If this BStore represents a volume's root directory (for which there is no parent), or if the object is invalid, this function returns **B_ERROR**; otherwise, it returns **B_NO_ERROR**.

GetPath()

long **GetPath(**char **buffer*, long *bufferSize***)**

Constructs the full path name to this object and copies the name into *buffer*. You must allocate the buffer before you pass it in; you pass the size of the buffer (in bytes) through the *bufferSize* argument.

The path name is absolute and includes the volume name as its first element. You could, for example, cache the name and then use it later as the argument to the global **get_ref_for_path()** function. As long as the file system hasn't changed, the latter function would return the ref of the original item.

If the object doesn't refer to a file system item, or if the buffer isn't long enough to accommodate the name, **B_ERROR** is returned and nothing is copied into the buffer. Otherwise, **B_NO_ERROR** is returned.

See also: **get_ref_for_path(), BDirectory::GetRefForPath()**

MoveTo()

long **MoveTo(**BDirectory **dir*,
const char **newName* = **NULL**,
store_creation_hook **hookFunc* = **NULL**,
void **hookData* = **NULL)**

Removes the item from its present directory, and moves it to the directory represented by *dir*. You can, optionally, rename the item at the same time by providing a value for the *newName* argument.

The *hookFunc* and *hookData* arguments let you alter the file's record before it's committed. This is exhaustively explained in the section "The Store Creation Hook" on page 73 of the introduction to this class.

See also: **SetName()**, **BFile::CopyTo()**, **BDirectory::Create()**

Record()

BRecord ***Record(**void**)**

Returns a BRecord object that represents the record in the database that holds information for the file system item that this BStore refers to. You can examine the values in the BRecord (through functions defined by the BRecord class), but you should only set and modify those fields that you've defined yourself (if any).

Any changes that you make to the BRecord must be explicitly committed by calling BRecord's **Commit()** function. Furthermore, you must commit your changes *before* calling other BStore functions, even those that are seemingly innocuous.

More information on the use and meaning of a BStore's record is given in the section "Custom Files" on page 71 of the introduction to this class.

SetName()

long **SetName(**const char **name***)*

Sets the name of the item to *name*. If the item is the root directory for its volume, the name of the volume is set to the argument as well.

Every item within a directory must have a different name; if *name* conflicts with an existing item in the same directory, the function fails and returns **B_NAME_IN_USE**. Also, you can't change the name of a file that's currently open; **SetName()** will return **B_ERROR** in this case. **B_ERROR** is also returned if, for any other reason, the name couldn't be changed. Success is indicated by a return of **B_NO_ERROR**.

See also: **GetName()**, **MoveTo()**

SetRef()

virtual long **SetRef(**record_ref *ref***)**
virtual long **SetRef(**BVolume **volume*, record_id *id***)**

Sets the object's record ref. By setting an BStore's ref, you cause the object to refer to a file in the file system.

The first version of the function sets the ref to the argument that you pass. This version of the function is typically called in response to a ref being received by your application.

The second version induces the ref from the BVolume (which implies a specific database) and record ID arguments. This version is useful if you're finding files through a database query.

More information on a BStore's ref is given in the section "Files, Records, and BStores" on page 69 of the introduction to this class.

VolumeID()

long **VolumeID(**void**)**

Returns the ID of the volume in which this item is stored. To turn the ID into a BVolume object, pass it to BVolume's **SetID()** function.

See also: **BVolume::SetID()**

Operators

= (assignment)

inline BStore& **operator=(**const BStore&**)**

Sets the ref of the left operand object to be the same as that of the right operand object.

== (equality)

bool **operator==(**BStore**)** const

Compares the two objects based on their refs. If the refs are the same, the objects are judged to be the same.

!= (inequality)

bool **operator!=(**BStore**)** const

Compares the two objects based on their refs. If the refs are not the same, the objects are judged to be not the same.

BTable

Derived from: public BObject

Declared in: <storage/Table.h>

Overview

The BTable class defines objects that represent *tables* in a database.

A table is a template for a *record*, where a record is a collection of data that describes various aspects of a "thing." As a template, the table characterizes the individual datums that a record can contain. Each such characterization, which consists of a name and a data type, is called a *field* of the table. To make an analogy, a table is like a class definition, its fields are like data members, and records are instances of the class.

A table's definition—the make-up of its fields—is persistent: The definition is stored in a particular database. Within a database, tables are identified by name; the BDatabase class provides a function, **FindTable()**, that lets you retrieve a table based on a name (more accurately, the function returns a BTable object that represents the table that's stored in the database). To create a new table, you use BDatabase's **CreateTable()**, passing the name by which you want the table to be known (an example is given in the next section). The reliance on BDatabase to find and create tables enforces two important BTable tenets:

- *A table can only exist in reference to a particular database*. You can't, for example, create a table and *then* add or otherwise "apply" it to a database. The BDatabase object that you use as the target of a **CreateTable()** invocation represents the database that will own the newly created table.
- *The Storage Kit manages the construction and freeing of BTables for you*. You *obtain* BTable objects—through BDatabase's **FindTable()** and **CreateTable()** (among others)—rather than construct them yourself.

A subtler point regarding tables is that they don't actually contain the records that they describe. For example, every file in the Be file system is represented by a record in the database. File records contain information such as the file's name, its size, when it was created, and so on. These categories of information (in other words, the "name," "size," "creation data,") are enumerated as fields in the "File" table. But the "File" table doesn't contain the records themselves—it's simply the template that's used to create file records.

Creating a Table

As mentioned above, you create a new table (and retrieve the BTable that's constructed to represent it) through BDatabase's **CreateTable()** function. The function takes two arguments:

- The first argument (a **char ***) supplies a name for the table. Unfortunately, the Storage Kit doesn't force table names to be unique. Before you create a new table, you should make sure your proposed name won't collide with an existing table (as demonstrated in the example below).
- The second argument is optional; it identifies a table—by name or by BTable object—that will act as the new table's "parent." If you designate a parent, the new table will automatically contain the parent's field definitions (as well as its grandparent's, and so on).

In the following example, a new table named "Employee" is created; the example assumes the existence and validity of the **a_db** BDatabase object:

```
BTable *employee_table;

/* It's a good idea to synchronize the BDatabase before
 * creating a new table.  This refreshes the object's table
 * list.
 */
a_db->Sync();

/* Make sure the database doesn't already have an
 * "Employee" table.
 */
if (a_db->FindTable("Employee") != NULL)
    return; /* or whatever */
else
    /* Create the table. */
     employee_table = a_db->CreateTable("Employee");
```

The table name that you choose should, naturally enough, fit the "things" that the table describes. By convention, table names are singular, not plural.

Adding Fields to a Table

Having created a table, you'll want to add fields to it by calling BTable's field-adding functions. A field has two properties: a name and a data type. You pass the name as an argument to a field-creating function; the data type is implied by the function name:

- **AddStringField()** adds a field that represents **(char *)** data.
- **AddLongField()** does the same for **long** data.
- **AddRawField()** is for buffers of unspecified data type (**void ***).

- **AddTimeField()** adds fields that hold **double** values. Despite the function's name, you use this for *any* double value, not just time values.

- **AddRecordIDField()** adds a record ID field. This is one of the trickier BTable notions, and is fully explained in the BRecord class description. Briefly, the value that a record ID field represents is an integer that uniquely identifies a specific record in the database. By adding a record ID field to a BTable, you allow records to point to each other. (Using database parlance, the field lets you "join" records.)

Typically, you add fields only when you're creating a new table; however, you're not prevented from adding them to existing tables.

Here we add three fields to the "Employee" table; the first field gives the employee's name, the second gives the employee's telephone extension, and the third identifies the record that represents the employee's manager (this is further explained in the BRecord class description):

```
field_key name_key =
     employee_table->AddStringField("name", B_INDEXED_FIELD);

field_key extension_key =
     employee_table->AddLongField("extension");

field_key manager_key =
     employee_table->AddRecordIDField("manager");
```

Notice that the **Add...Field()** functions don't return objects. That's because fields aren't represented by objects; instead, they're identified by name or by *field key*, as explained in the next section (a subsequent section explains the meaning of the B_**INDEXED_FIELD** argument used in the example).

You can retrieve information about a field through BTable's **GetFieldInfo()** functions.

Field Keys

A field key is an integer that identifies a field within its table. Field key values have the data type **field_key**, and are returned by the **Add**...**Field()** functions. (You can also get a field's key through the **FieldKey()** function, passing the field's name as an argument.) Field keys are used, primarily, when you add and retrieve BRecord data; this is taken up in the BRecord class description.

Field keys are *not* unique across the entire database—a field key value doesn't encode the identity of the field's table. Furthermore, a field's key value is computed on the basis of the field's name and data type. If you add, to a table, two fields that have the same name and data type (which you aren't prevented from doing), the fields will have the same field key value.

Field Flags

The optional second argument to the **Add...Field()** functions is a list of flags that you want to apply to the field. Currently, there's only one flag (**B_INDEXED_FIELD**), so the second argument is either that or it's excluded.

The presence of the **B_INDEXED_FIELD** flag means that the field will be considered when the database generates its index (which it does automatically). Indexing makes data-retrieval somewhat faster, but it also makes data-addition somewhat slower; the more fields that are indexed, the greater the difference on either side. In general, you should only index fields that you think will be most frequently used when data is retrieved (or *fetched*).

In the example, the "name" field is indexed; this implies the predication that employee data will most likely be fetched on the basis of the employee's name. (See the BQuery class for examples of how data is fetched.)

Table Inheritance

A table can inherit fields from another table. For example, let's say you want to create a "Temp" table that inherits from "Employee". To the "Temp" table you add fields named "agency" and "termination" (date):

```
BTable *temp_table;

a_db->Sync();

/* This time, we perform the name-collision check AND test
 * to ensure that the parent exists.
 */
if (a_db->FindTable("Temp") != NULL ||
    a_db->FindTable("Employee") == NULL)
    return;

/* Now create the table... */
temp_table = a_db->CreateTable("Temp", "Employee");

/* ... and add the new fields.  First we check to make sure
 * we didn't inherit these fields from "Employee". The checks
 * allow similarly named fields with different types, but not
 * fields that are identical in name -and- type.  You can
 * tighten the check to disallow fields with identical names
 * by omitting the FieldType() check.
 */
if ( temp_table->FieldKey("agency") != B_ERROR)
    if ( temp_table->FieldType("agency") != B_STRING_TYPE)
        field_key agency_key =
             temp_table->AddStringField("agency");

if ( temp_table->FieldKey("termination") != B_ERROR)
    if ( temp_table->FieldType("termination") != B_TIME_TYPE)
```

```
field_key term_key =
     temp_table->AddTimeField("termination");
```

The checks that accompany the field additions in the example are, perhaps, a bit overly-scrupulous, but they can be important in some situations, such as if you're letting a user define tables through manipulation of the user interface.

A table hierarchy can be arbitrarily deep. However, all tables within a particular hierarchy must belong to the same database—table inheritance can't cross databases. Also, there's no "multiple inheritance" for tables.

If you want your tables to show up in a Browser query window, the table must inherit, however remotely, from "BrowserItem". Furthermore, only those fields that start with a capital letter are displayed in the letter. Uncapitalized field names are considered private.

Note: Table hierarchies have nothing to do with the C++ class hierarchy. You can't manufacture a table hierarchy by deriving classes based on BTable, for example.

Type and App

When the user double-clicks an icon that's displayed by the Browser, the Browser launches (or otherwise finds) a particular app and then sends the clicked icon's record to the app. How does the Browser know which app to launch? If the icon represents a file, then the Browser can simply ask the file for the app's signature through the representative BFile object's **GetTypeAndApp()** message.

However, if the icon doesn't represent a file—if it represents a "pure" database record—then the Browser asks the record's table for *its* app, through BTable's **GetTypeAndApp()** function. When you create a new table, you set the type and app through **SetTypeAndApp()**. The "type" information for a table means the same thing as the "type" of a file: It's an application-specific identifier that describes the content of some data.

The type and app information for a table doesn't belong to the Browser. Any application can set and query this information.

Using a BTable

BTable objects are used in the definitions and operations of BRecord and BQuery objects. These topics are examined fully in the descriptions of those classes, and are summarized here.

BTables and BRecords

A table defines a structure for data, but it doesn't, by itself, supply or contain the actual data. To add data to a database, you must create and add one or more records. Records are created in reference to a particular table; specifically, the amount and types of data

that a record can hold is determined by the fields of the table through which it's created. The record is said to "conform" to the table.

In your application, you create a record for a particular table by passing the representative BTable object to the BRecord constructor:

```
/* Create a record for the "Employee" table. */
BRecord *an_employee = new BRecord(employee_table);
```

So constructed, the **an_employee** object will accept data for the fields that are contained in the **employee_table** object. Adding data to a BRecord, and examining the data that it contains, is performed through BRecord's **Set...()** and **Find...()** functions; the set of these functions complements BTable's **Add...Field()** set.

BTable and BQuery

A BQuery object represents a request to fetch records from the database. The definition of a BQuery includes references to one or more BTable objects. To add a BTable reference to a BQuery, you use the BQuery **AddTable()** or **AddTree()** function. The former adds a single BTable (passed as an argument), the latter includes the argument BTable and all its descendants.

When the BQuery performs a fetch, it only considers records that conform to its BTables' tables. You can further restrict the domain of candidate records as described in the BQuery class description. Anticipating that description, here's what you do to fetch all the records that confrom to a particular table:

```
/* Fetch all Employee records. */
BQuery *employee_query = new BQuery();

employee_query->AddTable(employee_table);
employee_query->PushOp(B_ALL);
employee_query->Fetch();
```

To fetch all "Employee" records—including those that conform to "Temp" as well as to any other table that descends from "Employee"—we add the "Employee" table as a tree:

```
employee_query->AddTree(employee_table);
employee_query->PushOp(B_ALL);
employee_query->Fetch();
```

Constructor and Destructor

The BTable class doesn't declare a constructor or destructor. You never explicitly create or destroy BTable objects; you use, primarily, a BDatabase object to find such objects for you. See the BDatabase class description.

Member Functions

AddLongField(), AddRawField(), AddRecordIDField(), AddStringField(), AddTimeField()

field_key **AddLongField(**char **field_name*, long *flags* = 0**)**
field_key **AddRawField(**char **field_name*, long *flags* = 0**)**
field_key **AddRecordIDField(**char **field_name*, long *flags* = 0**)**
field_key **AddStringField(**char **field_name*, long *flags* = 0**)**
field_key **AddTimeField(**char **field_name*, long *flags* = 0**)**

Adds a new field to the BTable and returns the **field_key** value that identifies it. You supply a name for the field through the *field_name* argument. The *flags* argument gives additional information about the field; currently, the only flag value that the functions recognize is **B_INDEXED_FIELD**. See the section "Field Keys" on page 85 for more information about indexing.

You declare the type of data that the field will hold by selecting the appropriate function:

- **AddRawField()** declares untyped data (**void ***).
- **AddLongField()** declares **long** data.
- **AddRecordIDField()** declares **record_id** values.
- **AddStringField()** declares (**char ***) data.
- **AddTimeField()** declares double data.

Note: You use **AddTimeField()** to add any double-bearing field, not just fields that will hold time values. The names will be fixed in a subsequent release.

Note that the functions don't force fields names to be unique within a BTable; you can add any number of fields with the same name. Furthermore (and slightly more concerning), you aren't prevented from adding fields that have identical names *and* types. Since field keys are based on a combination of name and type, this means that any number of fields in a table can have the same field key value.

See also: **GetFieldInfo()**

ChildAt()

BTable ***ChildAt(**long *index***)**

Returns the BTable that sits in the *index*'th slot of the target BTable's "child table" list. Only those BTables that are direct descendants of the target are considered; in other words, a BTable doesn't know about its grandchildren. The function returns **NULL** if the index is out-of-bounds.

See also: **CountChildren()**

CountChildren()

long **CountChildren(**void**)**

Returns the number of BTables that directly inherit from this BTable.

See also: **ChildAt()**

CountFields()

long **CountFields(**void**)**

Returns the number of fields in the BTable; the count includes inherited fields.

See also: **GetFieldInfo()**

Database()

BDatabase ***Database(**void**)**

Returns the BDatabase object that represents the database that owns the table that's represented by this BTable. This is the object that was the target of the **FindTable()** or **CreateTable()** function that manufactured this BTable object.

See also: **BDatabase::FindTable(), BDatabase::CreateTable()**

FieldKey()

field_key **FieldKey(**char **name***)**
field_key **FieldKey(**char **name***, long *type***)**

Returns the field key for the named field. The second version of the function is in case you have two fields with the same name, but different types (two fields with the same name *and* type can't be distinguished). The type argument must be one of the following constants:

B_LONG_TYPE
B_RAW_TYPE
B_RECORD_TYPE
B_STRING_TYPE
B_TIME_TYPE

Note: You use the **B_TIME_TYPE** for all **double**-field searches.

If the named field isn't found, **B_ERROR** is returned.

See also: **FieldType(), GetFieldInfo()**

FieldType()

long **FieldType(**field_key *key***)**
long **FieldType(**char **name***)*

Returns a constant that represents the type of data that the designated field holds. The possible return values are:

B_RAW_TYPE
B_LONG_TYPE
B_RECORD_TYPE
B_STRING_TYPE
B_TIME_TYPE

Note: The **B_TIME_TYPE** is used for all **double**-bearing fields.

If the field isn't found, **B_ERROR** is returned.

See also: **FieldKey(), GetFieldInfo()**

GetFieldInfo()

bool **GetFieldInfo(**long *index***,**
char **name*,
field_key **key*,
long **type*,
long **flags***)*
bool **GetFieldInfo(**char **name*,
field_key **key*,
long **type*,
long **flags***)*
bool **GetFieldInfo(**field_key *key***,**
char **name*,
long **type*,
long **flags***)*

Finds the field designated by the first argument and returns, in the other arguments, information about it. The first version identifies the field by index into the BTable's list of fields, the second by its name, and the third by its field key.

The value returned in the *type* argument is one of the following constants:

- **B_LONG_TYPE**
- **B_RAW_TYPE**
- **B_RECORD_TYPE**
- **B_TRING_TYPE**
- **B_TIME_TYPE**

Note: The **B_TIME_TYPE** is used for all **double**-bearing fields.

The *flags* value will either be **B_INDEXED_FIELD** or 0. (See "Field Keys" on page 85 for more information about field flags.)

If the field isn't found, the functions returns **FALSE**; otherwise they return **TRUE**.

See also: **AddLongField()...**

HasAncestor()

bool **HasAncestor(**BTable **a_table***)**

Returns **TRUE** if the target BTable inherits (however remotely) from *a_table*. Otherwise returns **FALSE**.

See also: **BDatabase::Parent()**, **BDatabase::CreateTable()**

Name()

char ***Name(**void**)**

Returns the table's name. The name is set when the table is created.

See also: **BDatabase::CreateTable()**

Parent()

BTable ***Parent(**void**)**

Returns the table's parent, or **NULL** if none. A table's parent is designated when the table is created.

See also: **BDatabase::CreateTable()**

BVolume

Derived from: public BObject

Declared in: <storage/Volume.h>

Overview

A BVolume object represents a *volume*, an entity that contains a single, hierarchical file system and a single database. The data in a volume (the file system and database) is persistent: It's stored on a medium such as a hard disk, floppy disk, CD-ROM, or other storage device.

When a volume's existence is made known to the computer—when the volume is *mounted*—the system automatically constructs a BVolume (for your application) to represent it. When the volume is unmounted, the representative object is automatically destroyed. You can retrieve these BVolume objects directly through global functions, or construct your own BVolume objects that point to the objects that are created by the Kit. This is described in the next section.

Through a BVolume object you can retrieve information such as the volume's name, its storage capacity, how much of the volume is available, and so on. None of the BVolume functions manipulate or alter the volume—for example, you can't unmount a volume by calling a BVolume function (and rightly so, mounting and unmounting isn't an activity that's expected of an application).

Retrieving a BVolume

There are three ways to retrieve BVolume objects:

- *Retrieve the "boot volume" directly.* The boot volume contains the executables for the kernel and servers that are running on your machine. To retrieve the BVolume that corresponds to the boot volume, call the global **boot_volume()** function:

```
BVolume myBootVol = boot_volume();
```

- *Step through your application's list of BVolume objects.* You do this through the global **volume_at()** function. The function takes an index argument (a **long**), and returns the BVolume object at that position in the list. The first BVolume is at index 0; others (if any) follow at monotonically increasing index numbers. To test the success of the function, you invoke **Error()** upon the returned object. The following example demonstrates this:

```
/* Print the name of every mounted volume. */
void VolumeNamePrinter()
{
   BVolume this_vol;
   char vol_name[B_OS_NAME_LENGTH];
   long counter = 0;
   while((this_vol = volume_at(counter++)))
   {
      if (this_vol.Error() != B_NO_ERROR)
         break;
      this_vol.GetName(vol_name);
      printf("Volume %s is available\n", vol_name);
   }
}
```

- *Construct an object based on a volume ID.* A volume is identified globally by a unique integer (a **long**). By passing a valid volume identifier as the argument to the BVolume constructor, you can retrieve a BVolume object that corresponds to the volume. As explained in the next section, volume ID numbers are passed to your application through BApplication hook functions that are called when volumes are mounted and unmounted. (Also, see the **ID()** function for more information on volume ID numbers.)

- *Retrieve a BVolume from a BDatabase*. As mentioned earlier, every volume contains a single database. Given a BDatabase object (which represents a specific database) you can retrieve the corresponding BVolume by passing the BDatabase object to the global **volume_for_database()** function.

Mounting and Unmounting

As mentioned above, BVolume objects are automatically constructed as volumes are mounted. Similarly, the system frees the BVolume object for a volume that's been unmounted. The system informs your application of these events through BApplication's **VolumeMounted()** and **VolumeUnmounted()** hook functions. Both functions provide a BMessage as an argument; in the "volume_id" field of the BMessage you'll find the volume ID of the affected volume. To turn the volume ID into a BVolume object, you pass it as an argument to the BVolume constructor .

In the following example implementation of these functions, information is printed as volumes are mounted and unmounted:

```
void MyApp::VolumeMounted(BMessage *msg)
{
    BVolume *new_vol;
    char vol_name[B_OS_NAME_LENGTH];

    /* Get the volume ID and turn it into an object. */
    new_vol = new BVolume(msg->FindLong("volume_id"));
    new_vol->GetName(vol_name);

    /* Print information about the volume. */
```

```
    printf("Volume %s mounted; %f bytes available.\n",
        vol_name, new_vol->FreeBytes());
}

void MyApp::VolumeUnmounted(BMessage *msg)
{
    BVolume *old_vol;
    char vol_name[B_OS_NAME_LENGTH];

    new_vol = new BVolume(msg->FindLong("volume_id"));
    new_vol->GetName(vol_name);

    /* Print information about the volume. */
    printf("Volume %s unmounted.\n", vol_name);
}
```

As implied by the example, **VolumeMounted()** is called after the BVolume is constructed; **VolumeUnmounted()** is called before the object is destroyed. Thus, within the implementations of these functions, you can assume that the BVolume object is still valid.

Important: If you want your application's volume list to be updated as volumes are mounted and unmounted, you *must* have a running **be_app** object. This is so even if you don't implement **VolumeMounted()** and **VolumeUnmounted()**. Furthermore, your application mustn't be an "Argv Only" app.

The File System

Every volume encapsulates the hierarchy of directories and files for a single file system. The "bridge" between a volume and the file system hierarchy is the volume's *root directory.* As its name implies, a root directory stands at the root of a file hierarchy such that all files (and directories) in the hierarchy can be traced back to it.

Every volume has a single root directory; to retrieve a volume's root directory (in the form of a BDirectory object), you pass an allocated BDirectory to BVolume's **GetRootDirectory()** function:

```
/* Get the root directory for the first mounted volume. */
BVolume *first_vol;
BDirectory root_dir;

first_vol = volume_at(0);
new_vol->GetRootDirectory(&root_dir);
```

The **GetRootDirectory()** "fills in" the BDirectory that you pass so that it refers to the root directory.

Volumes in Path Names

The Storage Kit's implementation of the file system obviates the need for path names. Specific files aren't identified by a concatenation of slash-separated subdirectories, but

by objects. However, path names are still displayed in terminal windows, and are used by command-line programs. To identify a volume in a path name, you use this format:

/volumeName/directoryName/directoryName/...

The volume name itself *doesn't* include the surrounding slashes. In other words, a volume name might be "fido" but not "/fido/" (nor "/fido" nor "fido/").

You can't set a volume's name directly—BVolume doesn't have a name-setting function. A volume takes its name from that of its root directory. To change a volume's name, you have to retrieve the root directory and change *its* name (by invoking **SetName()** on the BDirectory).

The Database

You can retrieve a volume's database through the BVolume **Database()** function. The function returns the BDatabase object that represents the database. As described in the BDatabase class description, BDatabase objects are created for you in much the same way as are BVolume objects: As volumes are mounted and unmounted, BDatabase objects that represent the contained databases are constructed and destroyed.

In general, you only need to access a volume's database if you're creating an application that performs database activities (as opposed to an application that uses the Storage Kit simply to access the file system).

Constructor and Destructor

BVolume()

BVolume(void**)**
BVolume(long *volume_id***)**

The first version of the constructor creates an "abstract" object that doesn't correspond to an actual volume. To create this correspondence, you invoke the **SetID()** function.

The second version creates a BVolume that corresponds to the volume identified by the argument.

~BVolume()

virtual **~BVolume(**void**)**

Destroys the object.

Member Functions

Capacity()

double **Capacity(**void**)**

Returns the number of bytes of data that the volume can hold. This is the total of used and unused data—for an assessment of available storage, use the **FreeBytes()** function.

See also: **FreeBytes()**

Database()

BDatabase ***Database(**void**)**

Returns the BDatabase object that represents the volume's database. Every volume contains exactly one database (and each database is contained in exactly one volume).

See also: **BDatabase::Volume()**

FreeBytes()

double **FreeBytes(**void**)**

Returns a measure, in bytes, of the available storage in the volume.

See also: **Capacity()**

GetName()

long **GetName(**char **name***)**

Copies the volume's name into the argument. The argument should be at least **B_OS_NAME_LENGTH** bytes long. The name returned here is that which, for example, shows up in the Browser's "volume window."

Setting the name is typically (and most politely) the user's responsibility (a task that's performed, most easily, through the Browser). If you really want to set the name of the volume programmatically, you do so by renaming the volume's root directory.

Currently, this function always returns **B_NO_ERROR**.

See also: **GetRootDirectory()**

GetRootDirectory()

long **GetRootDirectory(**BDirectory **dir***)**

Returns, in *dir*, a BDirectory object that's set to the volume's *root directory.* This is the directory that lies at the root of the volume's file system**,** and from which all other files and directories descend.

You have to allocate the argument that you pass to this function; for example:

```
BDirectory root_dir;

a_volume->GetRootDirectory(&root_dir);
```

Some of the BDirectory (and, through inheritance, BStore) functions are treated specially for the root directory:

- **SetName()** not only sets the name of the root directory, it also sets the name of the volume.
- **Remove()** and **MoveTo()** always fail for a root directory—you're not allowed to remove or move a root directory.
- **Parent()** returns **B_ERROR**. By definition, root directories don't have parents. (Admittedly, the error code returned by **Parent()** is less than helpful; you can't tell the difference between an asked-for-the-root's-parent **B_ERROR**, and a something-is-terribly-wrong **B_ERROR**.)

Currently, this function always returns **B_NO_ERROR**.

GetDevice()

long **GetDevice(**char **deviceName***)**

Copies the name of the device upon which the volume is mounted into *deviceName*. The argument should be allocated to hold at least **B_OS_NAME_LENGTH** characters. If the BVolume corresponds to an actual volume (if its ID is set), this returns **B_NO_ERROR**. Otherwise, it returns **B_ERROR**.

ID()

long **ID(**void**)**

Returns the volume's identification number. This number is unique among all volumes that are *currently* mounted, and is only valid for as long as the volume is mounted.

The value returned by this function is used, primarily, when you're communicating the identity of a volume to some other application.

See also: **volume_at()** in "Global Functions"

IsReadOnly()

bool **IsReadOnly(**void**)**

Returns **TRUE** if the volume is declared to be read-only.

IsRemovable()

bool **IsRemovable(**void**)**

Returns **TRUE** if the volume's media is removable (if it's a floppy disk).

Global Functions

The following functions are declared as global functions (in **storage/Volume.h**). Since they're global, they don't rightfully belong in the BVolume class specification. But since they pertain specifically to volumes, their place, here, is justified.

boot_volume()

BVolume **boot_volume(**void**)**

Returns the BVolume object that represents the "boot volume." This is the volume that contains the kernel and other system resources.

volume_at()

BVolume **volume_at(**long *index***)**

Returns the *index*'th BVolume in your application's volume list (counting from 0). The list is created and administered for you by the Storage Kit. See the class description, above, for an example of how the function is used.

If *index* is out-of-bounds, the function sets the returned object's **Error()** code to **B_ERROR**.

volume_for_database()

BVolume **volume_for_database(**BDatabase **db***)**

Returns the BVolume that corresponds to the volume that contains the database identified by the argument.

If *db* is invalid, the function sets the returned object's **Error()** code to **B_ERROR**.

Global Functions, Constants, and Defined Types

This section lists parts of the Storage Kit that aren't contained in classes.

Global Functions

boot_volume()

<storage/Volume.h>

BVolume **boot_volume(**void**)**

Returns the BVolume object that represents the machine's "boot" volume. This is the volume that contains the exectuables for the kernel, app server, net server, and so on, that are currently running.

See also: "BVolume" on page 91

database_for()

<storage/Database.h>

BDatabase ***database_for(**long *databaseID***)**

Returns the BDatabase object that represents the database that's identified by *databaseID*. Database ID numbers are unique and persistent (within a practical estimation of eternity).

If *databaseID* is invalid—if it doesn't identify an available database—the function returns **NULL**.

See also: "BDatabase" on page 7

does_ref_conform()

<storage/Record.h>

bool **does_ref_conform(**record_ref *ref*, const char **tableName***)*

Returns **TRUE** if the record referred to by *ref* conforms to the table identified by *tableName*, either directly or through table-inheritance; otherwise returns **FALSE**. Although you can use this function anywhere, it's particularly useful when testing refs that you are passed to your application in a BMessage object. Most commonly, you test to see if the refs you

have received represent files, directories, or either. The table names that you use for each of these is listed below:

- The “File” table is used for files.
- The “Folder” table is used for directories.
- The “FSItem” table is used for file system items (files and directories).

The Be software defines a number of other tables that you can use in the **does_ref_confrom()** test (the names listed above are by far the most useful). The complete list of Be-defined table names can be found in the section “System Tables” on page 107.

get_ref_for_path()

<storage/Store.h>

long **get_ref_for_path(**const char ***pathName*, record_ref ***ref***)**

This function finds the file (or directory) that’s designated by *pathName*, and returns the file’s ref by reference in *ref*. The path name should be absolute, and should include the volume name as its first element. (Although **get_ref_for_path()** will try to interpret a relative pathname as branching from the current working directory, you shouldn’t rely on this; the identity of the current working directory isn’t guaranteed.)

Note that BDirectory provides a **GetRefForPath()** member function that accepts absolute *or* relative path names.

update_query()

<storage/Query.h>

void **update_query(**BMessage ***aMessage***)**

Used to forward messages from the Storage Server to a live BQuery object. You use this function as part of a derived-class implementation of BApplication’s **MessageReceived()** function; you never call it elsewhere in your application.

See also: “BQuery” on page 35

volume_at()

<storage/Volume.h>

BVolume **volume_at(**long *index***)**

Returns the *index*’th BVolume in your application’s volume list (counting from 0). The list is created and administered for you by the Storage Kit.

If *index* is out-of-bounds, the function sets the returned object’s **Error()** code to **B_ERROR**.

See also: “BVolume” on page 91

volume_for_database()

<storage/Volume.h>

BVolume **volume_for_database(**BDatabase ***db*****)**

Returns the BVolume object that corresponds to the argument database (as represented by a BDatabase object).

If *db* is invalid—if it doesn't identify a database—the function sets the returned object's **Error()** code to **B_ERROR**.

Constants

File Open Modes

<storage/StorageDefs.h>

Constant	Meaning
B_READ_ONLY	The file is open for reading only.
B_READ_WRITE	The file is open for reading and writing.
B_EXCLUSIVE	The file is open for reading and writing, and no one else can open the file until its closed.
B_FILE_NOT_OPEN	The file isn't open.

The first three constants are used by BFile's **Open()** function to describe the mode in which the object should open its file. Add the fourth and you have the set of value that can be returned by BFile's **OpenMode()** function.

See also: **BFile::Open()**

File Seek Constants

<storage/StorageDefs.h>

Constant	Meaning
B_FILE_TOP	Seek from the first byte in the file.
B_FILE_MIDDLE	Seek from the currently-pointed to position.
B_FILE_BOTTOM	Seek from the last byte in the file.

These constants are used as arguments to BFile's **Seek()** function to describe where a file seek should start from.

See also: **BFile::Seek()**

Live Query Messages

<storage/Query.h>

Constant	Meaning
B_RECORD_ADDED	A record ref needs to be added to the BQuery's ref list.
B_RECORD_REMOVED	A ref needs to be removed from the list.
D_RECORD_MODIFIED	Data has changed in a record referred to by one of the refs in the ref list.

These constants are the potential **what** values of a BMessage that's sent from the Storage Server to your application.

See also: **BQuery::MessageReceived()**

query_op Constants

<storage/Query.h>

Constant	Meaning
B_EQ	equal
B_NE	not equal
B_GT	greater than
B_GE	greater than or equal to
B_LT	less than or equal to
B_LE	less than or equal to
B_AND	logically AND the previous two elements
B_OR	logically OR the previous two elements
B_NOT	negate the previous element
B_ALL	wildcard; matches all records

These **query_op** constants are the operator values that can be used in the construction of a BQuery's predicate.

See also: **PushOp()** in the BQuery class

Table Field Flags

<storage/Table.h>

Constant	Meaning
B_INDEXED_FIELD	Create an index based on the values taken by this field.

Each field that you add to a BTable takes a set of flags. Currently, the only flag that is recognized is **B_INDEXED_FIELD**.

See also: **BTable::AddLongField()**

Defined Types

database_id

<storage/StorageDefs.h>

typedef long **database_id**

The **database_id** type represents values that uniquely identify individual databases.

See also: **record_id**, the BDatabase class description

field_key

<storage/StorageDefs.h>

typedef long **field_key**

The **field_key** type represents fields in a BTable.

See also: the BTable class description

query_op

<storage/StorageDefs.h>

typedef long enum {...}**query_op**

The **record_ref** type represents a set of constants that can be used in a BQuery's predicate.

See also: **Query Operator Constants**

record_id

<storage/StorageDefs.h>

typedef long **record_id**

The **record_id** type represents values that uniquely identify records in a known database.

See also: **record_ref**, the BRecord class description

record_ref

<storage/StorageDefs.h>

typedef struct {
    record_id **record**;

```
        database_id database;
} record_ref
```

The **record_ref** type is a structure that uniquely identifies a particular record among all records in all currently available databases. The structure also defines the == and != operators, thus allowing **record_ref** structures to be compared as values.

See also: the BRecord class description

System Tables and Resources

System Tables

This section lists the names of the tables that are defined by the Storage Kit, as well as the names (and types) of the tables' fields. You should never need to use these tables, except to create other tables that inherit from them—you certainly shouldn't take advantage of the field definitions presented here in order to set record values yourself. They're listed, primarily, so you can avoid name collisions. Note that none of these names (whether of the tables or their fields) are defined as constants, nor are they published in any of the header files.

If you want your tables to show up in a Browser query window, the table must inherit, however remotely, from "BrowserItem". Furthermore, only those fields that start with a capital letter are displayed in the letter. Uncapitalized field names are considered private.

"Icon"

Parent table: (none)

Field Name	Field Type
"creator"	LONG_TYPE
"type"	LONG_TYPE
"largeBits"	RAW_TYPE
"smallBits"	RAW_TYPE

"Dock"

Parent table: (none)

Field Name	Field Type
"dbType"	LONG_TYPE
"dock_mode"	LONG_TYPE
"big_origin"	RAW_TYPE
"mini_origin"	RAW_TYPE

"BrowserItem"

Parent table: (none)

Field Name	Field Type
"Name"	STRING_TYPE
"Size"	LONG_TYPE
"Created"	TIME_TYPE
"Modified"	TIME_TYPE
"parentID"	LONG_TYPE
"dbType"	LONG_TYPE
"fsType"	LONG_TYPE
"fsCreator"	LONG_TYPE
"parentRef"	RECORD_TYPE
"flags"	LONG_TYPE
"xLoc"	LONG_TYPE
"yLoc"	LONG_TYPE
"iconRef"	RECORD_TYPE
"dock_index"	LONG_TYPE
"openOnMount"	LONG_TYPE
"inited"	LONG_TYPE
"invisible"	LONG_TYPE

"FSItem"

Parent table: "BrowserItem"

Field Name	Field Type
"appFlags"	LONG_TYPE
"version"	LONG_TYPE

"File"

Parent table: "FSItem"

Field Name	Field Type
"Project"	STRING_TYPE
"Description"	STRING_TYPE

"Folder"

Parent table: "FSItem"

Field Name	Field Type
"sortProperty"	LONG_TYPE
"sortReverse"	LONG_TYPE
"dirID"	LONG_TYPE
"viewMode"	LONG_TYPE
"lastIconMode"	LONG_TYPE
"numProperties"	LONG_TYPE
"propertyList"	RAW_TYPE
"windRect"	RAW_TYPE
"iconOrigin"	RAW_TYPE
"listOrigin"	RAW_TYPE

"Proxy"

Parent table: "BrowserItem"

Field Name	Field Type
"realItem"	RECORD_TYPE

"Volume"

Parent table: "Folder"

Field Name	Field Type
"Volume Size"	LONG_TYPE
"isLocal"	LONG_TYPE

"Machine"

Parent table: "Folder"

Field Name	Field Type
(none)	

"Query"

Parent table: "Folder"

Field Name	Field Type
"QueryString"	STRING_TYPE
"flatQuery"	RAW_TYPE
"database_id"	LONG_TYPE

"Person"

Parent table: "BrowserItem"

Field Name	Field Type
"Company"	STRING_TYPE
"Address"	STRING_TYPE
"Phone"	STRING_TYPE
"City"	STRING_TYPE
"State"	STRING_TYPE
"Zip"	STRING_TYPE
"E-mail"	STRING_TYPE
"Fax"	STRING_TYPE
"Comments"	STRING_TYPE

"E-Mail"

Parent table: "BrowserItem"

Field Name	Field Type
"Status"	STRING_TYPE
"Priority"	LONG_TYPE
"From"	STRING_TYPE
"Subject"	STRING_TYPE
"Reply"	STRING_TYPE
"When"	DOUBLE_TYPE
"Enclosures"	LONG_TYPE
"header"	RAW_TYPE
"content"	RAW_TYPE
"content_file"	RECORD_TYPE
"enclosures"	RAW_TYPE
"mail_flags"	LONG_TYPE

"Message"

Parent table: "BrowserItem"

Field Name	Field Type
"Status"	LONG_TYPE
"Kind"	LONG_TYPE
"From"	STRING_TYPE
"When"	TIME_TYPE
"Length"	LONG_TYPE
"dataFile"	STRING_TYPE
"At"	STRING_TYPE
"outbound"	LONG_TYPE
"Forum"	STRING_TYPE

"Preference"

Parent table: "BrowserItem"

Field Name	Field Type
"appSignature"	LONG_TYPE
"version"	LONG_TYPE
"User Name"	STRING_TYPE

System Resources

This section lists the resource types that the Be software uses. To be specific, the Icon World application adds resources of the following types to the applications that you create; the Browser looks for and recognizes these resource types when it displays file information and icons.

As with the table listings, above, the following is provided primarily so you can avoid unintentional collisions—in general, you shouldn't add resources by the types listed below. However, it isn't inconceivable that someone might try adding an 'ICON' resource directly (for example).

'APPI'

The resource that's identified by the type 'APPI' stores information about the application. The data in the resource is a single **app_info** structure. This structure is described in Chapter 2, "The Application Kit." The name of the 'APPI' resource is "app info".

'ICON'

The 'ICON'-type resource holds data that creates the application's large icons. The data for the resource is a 32x32 pixel bitmap in **COLOR_8_BIT** color space. For the exact representation of such data, see the BBitmap class in the Interface Kit.

There can be more than one 'ICON'-typed resource:

- The 'ICON' resource that's named "BAPP" holds the icon that's displayed for the application.
- The 'ICON' that takes, as a name, the application's signature converted to a string holds the data that's displayed for documents created by the application.

'MICN'

The ‘MICN’ type resource holds “mini-icon” data. The details are the same as the ‘ICON’ type described above, except that a mini-icon is a 16x16 pixel bitmap.