The Be™ Book

The Software Development Environment for the BeBox™
Developer Release 8

The Be Book: The Software Development Environment for the BeBox
reference documentation for the Be operating system, Developer Release 8
revised August 1996

For developer support, email: **devsupport@be.com**

Be, Inc.
800 El Camino Real
Suite 300
Menlo Park, CA 94025
http://www.be.com

1 Introduction

1 Introduction

The BeBox™ is an integrated package of hardware and software. The hardware supports the innovative design of the software, and the software exploits the extraordinary capabilities of the hardware. Among other things, the BeBox offers:

- Parallel processing on two high-performance CPUs.
- An operating system designed for efficient multithreading. It automatically splits assignments between the CPUs and will give priority to threads that need uninterrupted service.
- An architecture that supports the real-time processing of data for audio and video applications.
- An interface that lets applications and users view everything that's stored on-disk as if it were in a relational database.
- Dynamically loaded device drivers, built-in networking, interapplication messaging, shared libraries, protected and shared address spaces, an application framework that implicitly assigns a separate thread of execution to each window, and many other features.

Be system software is designed to make the features of the BeBox readily and efficiently available to all applications. The application programming interface (API) is written in the C++ language and takes advantage of the opportunities C++ offers for object-oriented programming. It includes numerous class definitions from which you can take much of the framework for your application.

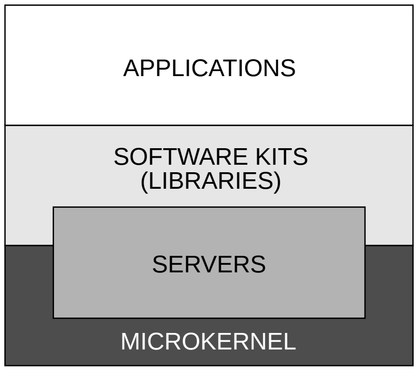
Software Overview

System software on the BeBox lies in three "layers":

- A microkernel that works directly with the hardware and device drivers.
- Several servers that can attend to the needs of any number of running applications. The servers take over much of the low-level work that would normally have to be done by each application.

- Dynamically linked libraries that provide an interface to the servers and encapsulate facilities for building Be applications.

Applications are built on top of these layers, as illustrated below:



The API for all system software is organized into several “kits.” Each software kit has a distinct domain—there’s a kit that contains the basic software you’ll need to run an application on the BeBox, a kit for putting together a user interface, one for organizing data stored on-disk, another for networking, and so on.

With the exception of the Kernel Kit and much of the Network Kit, which have ANSI C interfaces, all the kits are written in the C++ language and make extensive use of class definitions. Each kit defines an integrated set of classes that work together to structure a framework for applications within its domain.

By incorporating kit classes in your application—directly creating instances of them, deriving your own classes from them, and inventing your own classes to work with them—you’ll be able to make use of all the facilities built into the BeBox. And you’ll find that a good deal of the work of programming a Be application has already been done for you by the engineers at Be.

Servers

Standing behind many of the software kits are servers—separate processes that run in the background and carry out basic tasks for client applications. Servers serve Be applications, not users; they have a programming interface (through the various kits) but no user interface. They typically can serve any number of running applications at the same time. A server can be viewed either as an extension of the kernel or as an adjunct to an application. It’s really a little of both.

If you look inside the **/system** directory on the BeBox, you’ll see a number of servers listed. The main ones that you should know about are the Storage Server and the Application Server.

- The *Storage Server* coordinates access to “persistent data”—data that lives on long-term storage media, such as a hard disk or floppy diskette. The Server keeps track of data by enumerating its qualities (its type, size, where it’s located, and so on) in an entry called a *record*. In some cases, the record can hold the data itself; it can be a way of retaining data, not just of recording information about it. A *database* contains a collection of records; each medium (each hard or floppy disk) has its own database.

The Storage Server also manages the file system. A file, like any other distinct piece of persistent data, is represented by a record in a database. Although you gain access to files by referring to the records that represent them, Be software is designed to make file access as “database-free” as possible.

An application can create new records, add them to a database, query a database and access records, open files to read and write, traverse directories, and carry out other storage and retrieval tasks through the classes defined in the Storage Kit. The Kit is the programming interface to the Server.

- The *Application Server* handles most of the low-level user interface work. It provides applications with windows, manages the interactions among windows, renders images in windows on instructions from the application, and monitors what the user does on the keyboard and mouse. It’s the application’s conduit for both drawing output and event input. In addition to being a “window provider,” the Server also maintains the global environment shared by all applications.

  An application connects itself to the Server when it constructs a BApplication object (as defined in the Application Kit). This should be one of the first things that every application does. Every BWindow object (defined in the Interface Kit) also makes a connection to the Server when it’s constructed. Each window runs independently of other windows—in its own thread and with its own connection to the Server.

In addition, the *Print Server* manages the printer and printing tasks much as the Application Server manages rendering on the screen. Various media servers take care of the distribution of data to and from media devices. For example, the *Audio Server* manages sound data that arrives through the microphone and line-in jacks, and sends sound data to the speaker and line-out jacks. These servers will turn up in later chapters as they discuss the architecture of system software. Most other servers will remain in the background.

Kits

Some of the software kits will be used by all applications, others only by applications that are concerned with the specific kinds of problems the kit addresses. Most applications will need to open files and put windows on-screen, for example; fewer will want to process audio data.

The kits currently available are summarized below:

- The *Application Kit* is a small amount of software that is nevertheless essential for all applications. It gives an application the ability to communicate with other applications, to become known to the Browser, and to use software in the other kits. It defines a messaging service that the system uses to report events to applications, and that an application can use to organize activity among its threads.

  The Kit's principal class is BApplication; every application must have one (and only one) BApplication object to act as its global representative. Begin with this kit before programming with any of the others.

- The *Storage Kit* is an interface for storing data on-disk, retrieving it, and keeping abreast of changes that are made to it. It's the client interface to the Storage Server. Information can be stored with various attached properties, so that it can be retrieved, accessed, and organized according to those properties, not just according to a file designation in a hierarchical directory structure.

  The Storage kit has two parts: One set of classes (BDatabase, BTable, BRecord, and BQuery) provides typical database access to stored information. Another set (BVolume, BDirectory, and BFile) provides an interface to the file system. The file-system classes are built on top of the database classes—files and directories have records in the database—but can be used with a minimum of "database" overhead. In the easiest (and most typical) case, an application doesn't need to know anything about database techniques to read and write files.

- The *Interface Kit* is used to build and run a graphical and interactive user interface. It structures the twin tasks of drawing in windows and handling the messages that report user actions (like clicks and keystrokes) directed at what was drawn. Its BWindow class encapsulates an interface to windows. Its BView class embodies a complete graphics environment for drawing.

  Each window on (and off) the screen is represented by a separate BWindow object and is served by a separate thread. A BWindow has a hierarchy of associated BView objects; each BView draws one portion of what's displayed in the window and responds to user actions prompted by the display. The Interface Kit defines a number of specific BViews, such as BListView, BButton, BScrollBar, and BTextView—as well as various supporting classes, such as BRegion, BBitmap, and BPicture.

  Every application that puts a window on-screen will need to make use of this kit.

- The *Media Kit* defines an architecture for the real-time processing of data—especially audio and video data. It gives applications the ability to generate, examine, manipulate, and realize (or "render") medium-specific data in real time. Applications can, for example, synchronize the transmission of data to different media devices, so they can easily incorporate and coordinate audio and video output.

- The *Midi Kit* is designed specifically for processing music data in MIDI (Musical Instrument Digital Interface) format.

- The *3D Kit* brings an object-oriented implementation of 3D concepts into the Be software environment. Its goal is to make fairly sophisticated three-dimensional representations available to ordinary applications—and to do so simply and efficiently. With this Kit, an application can define three-dimensional objects (or "models"), place the models in a three-dimensional setting, animate them in the setting, and have the user interact them. < In addition to the 3D Kit, a future release will provide an optimized implementation of the OpenGL library and tools for developers who require a high-end 3D engine. >

- The *Kernel Kit* is the one kit that's not object-oriented. It defines an interface for creating threads (the basic units of scheduling and execution on the CPUs) and the attendant facilities that regulate threads and coordinate their interaction (such as ports, priorities, and semaphores). It also defines a system of memory management, including reserved and shared areas of memory. Applications that rely on the higher-level kits won't need to use much of the kernel interface.

- The *Device Kit* has two parts. One part provides programming interfaces for the various connectors on the BeBox; it currently consists of classes that represent the serial ports, the joystick ports, and the GeekPort™. The other part of the Kit is the API for creating loadable device drivers. Drivers for graphics cards run as extensions of the Application Server; printer drivers run in the Print Server. All other drivers are loaded by the kernel.

- The *Game Kit* is a collection of software that's especially useful for developing games, though it can be used by any application. It currently consists of just one class—BWindowScreen—which gives an application direct access to the graphics card driver for the screen. With that access, the application can set up a game-specific graphics environment on the card, take direct charge of the frame buffer, and call driver functions for accelerated drawing.

- The *Network Kit* contains global C functions that let you identify remote machines that are connected to the network, and communicate with those machines through the TCP and UDP message protocols. The Kit also contains API (including the BMailMessage class) that enables applications to talk to the Be mail daemon, and send and receives SMTP and POP mail messages.

- The *Support Kit* is a collection of various defined types, error codes, and other facilities that support Be application development and the work of the other kits. It includes basic type definitions, the BList class for organizing ordered collections of data, and a system for having objects retain class information that they can reveal at run time. You can pick and choose the parts of this kit that you want to adopt for your application.

Contents

This manual documents system software for which a public API is currently available. The present version covers the eleven kits summarized above. Later releases will document more software as the API is codified.

After the introductory chapter you're now reading, there's a chapter for each kit, followed by two appendices. The table of contents is:

We may, from time to time, issue updated versions of one chapter or another, as well as add new chapters for new kits. So that page numbers won't become totally confusing as new documentation arrives, each chapter numbers its pages independently of the others. Each chapter begins on page 1 and has its own table of contents.

Where it can, the documentation tries to let you know what might be changing. It encloses temporary comments in angle brackets, <such as this>. Bracketed information is sometimes speculative, anticipating planned changes to the software that have yet to be implemented. Angle brackets sometimes also enclose information that's true about the present release, but is scheduled to change. Hopefully, language and context are enough to distinguish the two cases.

Just as the software tries to simplify the work of programming an application for the BeBox, this documentation tries to make it easy for you to understand the software. Your comments on it, as on the software, are appreciated. Suggestions, bug reports, and notes on what you found helpful or unhelpful, clear or unclear, are all welcome.

Class Descriptions

Since most Be software is organized into classes, much of the documentation you'll be reading in this manual will be about classes and their member functions. Each class description is divided into the following sections:

Overview	An introductory description of the class. The overview is usually brief, but for the main architectural classes, it can be lengthy. Start here to learn about the class.
Data Members	A list of the public and protected data members declared by the class, if there are any. If this section is missing, the class declares only private data members, or doesn't declare any data members at all. Most data members are private, so this section is usually absent.
Hook Functions	A list of the virtual functions that you're invited to override (re-implement) in a derived class. Hook functions are called by the kit at critical junctures; they "hook" application-specific code into the generic workings of the kit. Looking through the list will give you an idea of how to adapt the kit class to the needs of your application.
Constructor and Destructor	The class constructor and destructor. Only documented constructors produce valid members of a class. Don't rely on the default constructors promised by the C++ compiler.
Member Functions	A full description of all public and protected member functions, including hook functions.
Operators	A description of any operators that are overloaded to handle the class type.

If a section isn't relevant for a particular class—if the class doesn't define any hook functions or overload any operators, for example—that section is omitted.

Rely only on the documented API. You may occasionally find a public function declared in a header file but not documented in the class description. The reason it's not documented is probably because it's not supported and not safe; don't use it.

Programming Conventions

The software kits were designed with some conventions in mind. Knowing a few of these conventions will help you write efficient code and avoid unexpected pitfalls. The conventions for memory allocation, object creation, and virtual functions are described below.

Responsibility for Allocated Memory

The general rule is that whoever allocates memory is responsible for freeing it:

- If your application allocates memory, it should free it.
- If a kit allocates memory and passes your application a pointer to it, the kit retains responsibility for freeing it.

For example, a **Text()** function like this one,

```
char *text = someObject->Text();
```

would return a pointer to a string of characters residing in memory that belongs to the object that allocated it. The object will free the string; you shouldn't free it.

You should also not expect the string pointer to be valid for long. It will stay valid as long as you hold a lock that prevents others from changing the string or deleting the object. But once you release the lock that protects the data, something may happen to modify it, change its location in memory, or free it at any time. If your application needs continued access to the string, it should make a copy for itself or call **Text()** each time the string is needed.

In contrast, a **GetText()** function would copy the string into memory that your application provides:

```
char *text = (char *)malloc(someObject->TextLength() + 1);
someObject->GetText(text);
```

Your application is responsible for the copy.

In some cases, you're asked to allocate an object that kit functions fill in with data:

```
BPicture *picture = new BPicture;
someViewObject->BeginPicture(picture);
. . .
someViewObject->EndPicture();
```

Because your application allocated the object, it's responsible for freeing it.

Be system software tries always to keep allocation and deallocation paired in the same body of code—if you allocated the memory, free it; if you didn't, don't.

This general rule is followed wherever possible, but there are some exceptions to it. BMessage objects (in the Application Kit) are a prominent exception. Messages are like packages you put together and then mail to someone else. Although you create the package, once you mail it, it no longer belongs to you.

Another exception is **FindResource()** in the BResourceFile class of the Storage Kit. This function allocates memory on the caller's behalf and copies resource data to it; it then passes responsibility for the memory to the caller:

```
long numBytes;
void *res = someFile.FindResource(B_RAW_TYPE, "name", &numBytes);
```

The BResourceFile object allocates the memory in this case because it knows better than the caller how much resource data there is and, therefore, how much memory to allocate.

Exceptions like this are rare and are clearly stated in the documentation.

Object Allocation

All objects can be dynamically allocated (using the **new** operator). Some, but not all, can also be statically allocated (put on the stack). Static allocation is appropriate for certain kinds of objects, especially those that serve as temporary containers for transient data.

However, many objects may not work correctly unless they're allocated in dynamic memory. The general rule is this:

> *If you assign one object to another (as, for example, a child BView in the Interface Kit is assigned to its parent BView or a BMessage is assigned to a BMessenger), you should dynamically allocate the assigned object.*

This is because there may be circumstances which would cause the other object to get rid of the object you assigned it. For example, a parent BView deletes its children when it is itself deleted. In the Be software kits, all such deletions are done with the **delete** operator. Therefore, the original allocation should always be done with **new**.

Virtual Functions

The software kits declare functions virtual for a variety of reasons. Most of the reasons simply boil down to this: Declaring a function virtual lets you reuse its name in a derived class. You can, for example, implement a special version of a function for one kind of object and give it the same name as the version defined in the kit for other objects. Or, if you always take certain steps when you call a particular kit function, you can reimplement the function to include those steps. You don't have to package your additions under a different name.

However, there's another, more important reason why some functions are declared virtual. These functions reverse the usual pattern for library functions: Instead of being implemented in the kit and called by the application, they're called by the kit and implemented in the application. They're "hooks" where you can hang your own code and introduce it into the on-going operations of the kit.

Hook functions are called at critical junctures as the application runs. They serve to notify the application that something has happened, or is about to happen, and they give the application a chance to respond.

For example, the BApplication class (in the Application Kit) declares a **ReadyToRun()** function that's called as the application is getting ready to run after being launched. It can be implemented to finish configuring the application before it starts responding to the user. The BWindow class (in the Interface Kit) declares a **WindowActivated()** function that can be implemented to make any necessary changes when the window becomes the active window. By implementing these functions, you fit application-specific code into the generic framework of the kit.

It's possible to divide hook functions into three groups:

- Most hook functions are empty. As implemented by the declaring class, they don't do anything. It's up to derived classes to give them substance. Like **WindowActivated()** and **ReadyToRun()**, these functions are named for what they announce—for what led to the function call—rather than for what they might be implemented to do. They can be implemented to do almost anything you want.

- Some hook functions are given default implementations to cover the general case. Like the functions in the first group, these functions are also named for the occurrence that prompts the function call—for example, **ScreenChanged()** and **QuitRequested()**. If you decide to implement your own version of the function, you can choose either to *replace* the kit's default version or to *augment* it, as discussed below.

- A few hook functions are implemented to perform a particular task. You can call these functions just as you would any ordinary nonhook function, but they're also called at pivotal points within the framework of the kits. They therefore do double duty: They serve both as functions that you might call and as hooks that are called for you. These functions are generally named for what they do—like **MakeFocus()** or **SetValue()**. If you override any of them, you should always augment the original version, never replace it.

If you override a hook function that has been implemented—either by the class that declares it or by a derived class—it's generally best to preserve what the function already does by incorporating the old version in the new. For example:

```
void MyWindow::ScreenChanged(BRect grid, color_space mode)
{
    . . .
    BWindow::ScreenChanged(grid, mode);
    . . .
}
```

In this way, the new function augments the inherited version, rather than replaces it. It builds on what has already been implemented. In some cases, each class in a branch of the inheritance hierarchy will contribute a bit of code to a function. Because each version incorporates the inherited version, the function has its implementation spread vertically throughout the inheritance hierarchy.

Multiple Threads

A Be application is inherently multithreaded; it runs as a *team* of separately scheduled threads of execution that share a common address space. In addition to the *main thread* in which the application starts up and in which its **main()** function executes, each window is provided with its own thread. An application becomes multithreaded simply by creating a window.

Applications might explicitly create other threads for a variety of reasons—a thread might monitor a data channel, for example, or some less important processes might be put in a thread with a low priority to keep the user interface responsive. In addition, some kits (such as the Media Kit) have architectures that invite you to use multiple threads, and some spawn threads that work behind the scenes (like the thread that keeps live queries alive).

Each thread runs independently of the others, but the main thread has a special status. It's the first thread in the team, and it should also be the last. All other threads should be killed before the main thread and the application team are laid to rest.

The following sections discuss some considerations that come up when programming in a multithreaded environment. You may want to defer reading them until you see how the Be operating system defines and makes use of threads.

Protecting Data

Because all threads in a team live in the same address space, more than one of them might try to access the same data at the same time. If a data structure is static, unchanging, and certain to remain in place until the application quits, this won't be much of a problem. But that's generally not the case. If it's possible for one thread to alter some shared data, or delete it, while another thread is reading it (or worse, while the other thread is also altering the data, but in a different way), obvious problems result. Data could be left in an internally inconsistent state, pointers could be invalid, and so on.

There are various ways to avoid these problems—to keep critical data "multithread-safe". One maneuver is to put a single thread in charge of a data structure (or object). From the point of view of the data, the application isn't multithreaded; only one thread can read, alter, or delete it. Functions that deal with the data could simply return an error if the calling thread lacks authorized access.

The Be operating system provides two additional mechanisms that you can use to keep data multithread-safe:

- You can institute a locking procedure for the data. Locks are based on semaphores, which the Kernel Kit provides. Threads, in effect, wait in line to acquire a semaphore that gives them permission to access the data. When one thread releases the semaphore, the next thread can acquire it.

  Classes in some kits define **Lock()** and **Unlock()** functions that utilize semaphores, so it makes sense to talk about "locking" and "unlocking" an object—a window, for example.

  As long as all parties abide by the rules, semaphores and locks guarantee that only one thread at a time will be admitted to the data. Without this mechanism, one thread could not safely access data controlled by another thread.

- You can use the high-level messaging system, which the Application Kit defines. Messages asynchronously transfer control from one thread to another. They can make sure that a particular thread deals with particular data. For example, instead of locking an object and directly modifying its state, you can post a message to the thread that's associated with the object, and have that thread make the modification. If window *A* posts a message that concerns window *B*, window *B* will receive and respond to the message in its own thread.

  Using messages to communicate between threads ensures that each thread operates on just its "own" data. For example, a window thread might accept messages that affect the window data structure (really a BWindow object) and other objects associated with the window. As long as other threads post messages rather than try for direct access, these objects will be accessed only from one thread.

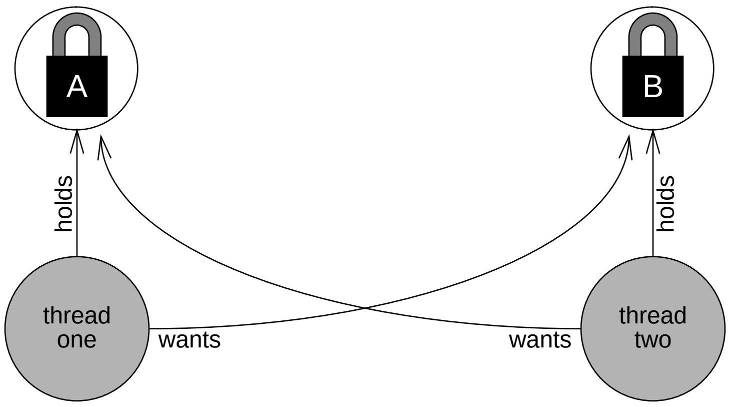
In the Be operating system, locks and messages are bound together in one important respect: When a thread receives a message, it automatically locks the object associated with the thread. For example, when a window thread gets a message, it locks the window data structure (the BWindow object). The lock remains in place until the thread is finished responding to the message.

This makes it possible for locks and messages to be used in combination in a multithreaded world. The choice of which to use depends on the situation and the design of your application.

The locking and messaging mechanisms are themselves multithread-safe on the BeBox. The system handles all the tricky cases—such as a destination thread disappearing while a message is being posted to it or a data structure being deleted while it's being locked. The functions that acquire a semaphore or a lock and those that post messages are designed to fail gracefully and return an error if the objects of their attention have been destroyed.

Avoiding Deadlocks

A deadlock occurs when one thread tries to acquire a lock that another thread holds, while the other thread tries to acquire a lock that the first thread holds. This is diagrammed below:



Each thread blocks waiting to acquire the lock that the other thread holds. Neither will succeed because neither will release the lock it holds while it waits for the other thread to release its lock. They both wait forever—a deadlock. (Deadlocks can also involve a combination of three or more threads, but two are sufficient. The essential ingredient is that each thread holds a lock while it waits for another lock.)

As the diagram above indicates, there are two necessary conditions for a deadlock to occur. A deadlock can't happen unless:

- A thread that holds a lock tries for another one. If threads hold only one lock at a time, deadlocks can't occur.
- Two or more threads must try to acquire the same locks, but in a different sequence. In the illustration, thread one acquires lock *A* first, then tries for *B*, while thread two works in the opposite order. It acquires *B* first, then tries for *A*. If all threads always acquire locks in the same order, deadlocks can't occur.

If you structure your code to avoid either or both of these conditions, you won't experience deadlocks.

As mentioned earlier, when a thread receives a message it locks the object that owns the thread. Therefore, as a thread responds to a message it implicitly holds one lock. If it tries for another one, it will meet the first condition for a deadlock stated above.

At times this may be unavoidable. When it is, it's important to structure the code so that all threads try for the locks in the same order. For example, if window *X* and window *Y* need to share data, and window *X* can lock window *Y* and window *Y* can lock window *X*, there's a distinct possibility that a deadlock will sometime occur. If the information that each window needs from the other is moved to some third object under the supervision of another lock, a deadlock could be avoided. If more than one additional object is needed and more than one lock, both windows could acquire the external locks in the same order, avoiding a deadlock.

Sometimes the solution to a deadlock is to avoid locking and rely on messages instead. The two windows in the example above might send each other messages rather than use locks to access the data directly.

Naming Conventions

As Be continues to develop system software and the API grows, there's a chance that the names of some new classes, constants, types, or functions added in future releases will clash with names you're already using in the code you've written.

To minimize the possibility of such clashes, we've adopted some strict naming conventions that will guide all future additions to the Be API. By stating these conventions here, we hope to give you a way of avoiding namespace conflicts in the future.

Most Be data structures and functions are defined as members of C++ classes, so class names will be quite prominent in application code. All our class names begin with the prefix "B"; the prefix marks the class as one that Be provides. The rest of the name is in mixed case—the body of the name is lowercase, but an uppercase letter marks the beginning of each separate word that's joined to form the name. For example:

BTextView	BFile
BRecord	BMessageQueue
BScrollBar	BList
BAudioSubscriber	BDatabase

The simplest thing you can do to prevent namespace clashes is to refrain from putting the "B" prefix on names you invent. Choose another prefix for your own classes, or use no prefix at all.

Other names associated with a class—the names of data members and member functions—are also in mixed case. (The names of member functions begin with an uppercase letter—for example, **AddResource()** and **UpdateIfNeeded()**. The names of data members begin with a lowercase letter—**what** and **bottom**, for example.) Member names are in a protected namespace and won't clash with the names you assign in your own code; they therefore don't have—or need—a "B" prefix.

All other names in the Be API are single case—either all uppercase or all lowercase—and use underbars to mark where separate words are joined into a single name.

The names of constants are all uppercase and begin with the prefix "B_". For example:

B_NAME_NOT_FOUND	**B_BACKSPACE**
B_OP_OVER	**B_LONG_TYPE**
B_BAD_THREAD_ID	**B_FOLLOW_TOP_BOTTOM**
B_REAL_TIME_PRIORITY	**B_PULSE**

It doesn't matter whether the constant is defined by a preprocessor directive (**#define**), in an enumeration (**enum**), or with the **const** qualifier. They're all uniformly uppercase, and all have a prefix. The only exceptions are common constants not specific to the Be operating system. For example, these four don't have a "B_" prefix:

TRUE	**NIL**
FALSE	**NULL**

Other names of whatever stripe—global variables, macros, nonmember functions, members of structures, and defined types—are all lowercase. Global variables generally begin with "be_",

be_app
be_roster
be_clipboard

but other names lack a prefix. They're distinguished only by being lowercase. For example:

rgb_color	**pattern**
system_time()	**acquire_sem()**
does_ref_conform()	**bytes_per_row**
app_info	**get_screen_size()**

There are few such names in the API. The software will grow mainly by adding classes and member functions, and the necessary constants to support those functions.

To briefly summarize:

Category	Prefix	Spelling
Class names	B	Mixed case
Member functions	*none*	Mixed case, beginning with an uppercase letter
Data members	*none*	Mixed case, beginning with a lowercase letter
Constants	**B_**	All uppercase
Global variables	**be_**	All lowercase
Everything else	*none*	All lowercase

If you adopt other conventions for your own code—perhaps mixed-case names, or possibly a prefix other than "B"—your names shouldn't conflict with any new ones we add in the future.

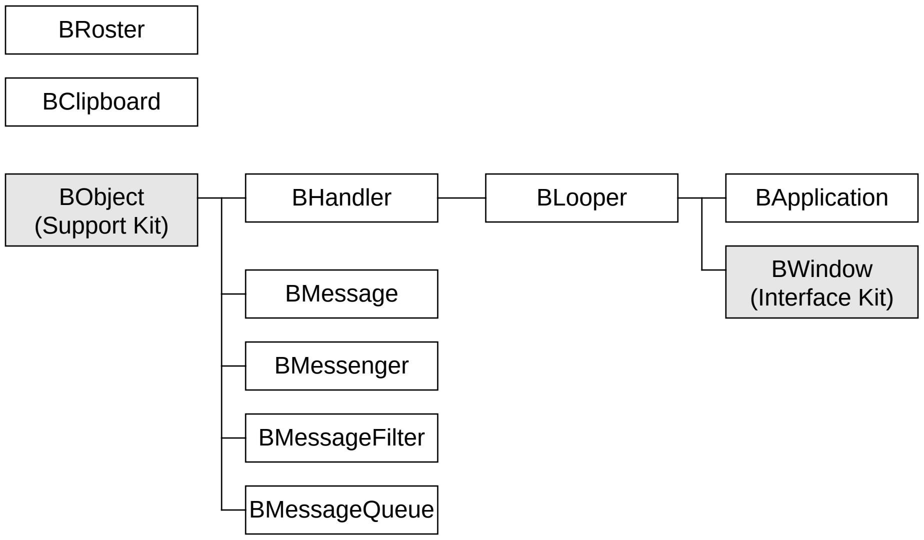
In addition, you can rely on our continuing to follow the lexical conventions established in the current API. For example, we never abbreviate "point" or "message," but always abbreviate "rectangle" as "rect" and "information" as "info." We use "begin" and "end," never "start" or "finish," in function names, and so on.

Occasionally, private names are visible in public header files. These names are marked with both pre- and postfixed underbars—for example, **_entry_** and **_remove_volume_()**. Don't rely on these names in the code you write. They're neither documented nor supported, and may change or disappear without comment in the next release.

Pre- and postfixed underbars are also used for kit-internal names that may intrude on an application's namespace, even though they don't show up in a header file. For example, the kits use some behind-the-scenes threads and give them names like "_pulse_task_" and they may put kit-internal data in public messages under names like "_button_". If you were to assign the same names to your threads and data entries, they might conflict with kit code. Since you can't anticipate every name used internally by the kits, it's best to avoid all names that begin and end in underbars.

2 The Application Kit

Application Kit Inheritance Hierarchy



2 The Application Kit

The Application Kit is the starting point for all applications. Its classes establish an application as an identifiable entity—one that can cooperate and communicate with other applications (including the Browser). It lays a foundation for the other kits. Before designing and building your application, you should secure a breathing familiarity with this basic Kit.

There are four parts to the Application Kit:

- *Messaging*. The Kit sets up a mechanism through which an application can easily make itself multithreaded, and a messaging service that permits the threads to talk to each other. This same service also delivers messages from one application to another—it's used for both inter- and intra-application communication.

  The messaging mechanism is implemented by a set of collaborating classes: BMessage objects bundle information so that it can be posted to a thread within the same application or sent to a thread in another application. BLooper objects run message loops in threads, getting messages as they arrive and dispatching them to BHandler objects. The BHandler's job is to respond to the message.

  The system employs the messaging mechanism to carry basic input to applications—from the keyboard and mouse, from the Browser, and from other external sources; system messages drive what most applications do. Every application will be on the receiving end of at least some of these messages and must have handlers ready to respond to them.

  Applications can also use the mechanism to create threads with a messaging interface, arrange communication among the threads, or exchange information with and issue commands to other applications.

- *The BApplication class*. Every application must have a single instance of the BApplication class—or of a class derived from BApplication. This object provides a number of essential services. Foremost among them is that it establishes a connection to the Application Server. The Server is a background process that takes over many of the fundamental tasks common to all applications. It renders images in windows, controls the cursor, reports what the user is doing on the keyboard and mouse, and, in general, keeps track of system resources.

  The BApplication object also runs the application's main message loop, where it receives messages that concern the application as a whole. Externally, this object represents the application to other applications; internally, it's the center where application-wide services and global information can be found. Because of it's pivotal role, it's assigned to a global variable, **`be_app`**, to make it easily accessible.

Other kits—the Interface Kit in particular—won't work until a BApplication object has been constructed.

- *The BRoster class*. The BRoster object keeps track of all running applications. It can identify applications, launch them, and provide the information needed to set up communications with them.
- *The BClipboard class*. The BClipboard object provides an interface to the clipboard where cut and copied data can be stored, and from which it can be pasted.

The messaging framework and the fundamentals of setting up a Be application are described in the following sections of this introduction. The BApplication class is documented beginning on page 21. The other classes follow in alphabetical order.

Messaging

At minimum, a messaging service must provide the means for:

- Putting together a parcel of information that can be delivered to a destination. In the Be model, these parcels are BMessage objects.
- Delivering messages to a destination. This is the job of a BMessenger object—although local messages can be "posted" directly, without the aid of a messenger. BMessengers mainly represent remote destinations.
- Processing messages as they arrive. This task is entrusted to BLooper objects.
- Letting applications define their own message-handling code. A BLooper dispatches an arriving message by calling a "hook" function of a BHandler object. Each application can implement these functions as it sees fit.

Messages

BMessage objects are parcels of information that can be transferred between threads. The message source constructs a BMessage object, adds whatever information it wants to it, and then passes the parcel to a function that delivers it to a destination.

A BMessage can hold structured data of any type or amount. When you add data to a message, you assign it a name and a type code. If more than one item of data is added with the same name and type, the BMessage creates an array of data for that name. The name and an index into the array are used to retrieve the data from the message.

The object also contains a *command constant* that says what the message is about. It's stored as a public data member (called **what**). The constant may:

- Convey a request of some kind (such as **B_ZOOM** or **BEGIN_ANIMATION**),
- Announce an event (such as **RECEIPT_ACKNOWLEDGED** or **B_WINDOW_RESIZED**), or
- Label the information that's being passed (such as **PATIENT_INFO** or **NEW_COLOR**).

Not all messages have data entries, but all should have a command constant. Sometimes the constant is sufficient to convey the entire message.

Message Protocols

Both the source and the destination of a message must agree upon its format—the command constant and the names and types of data entries. They must also agree on details of the exchange—when the message can be sent, whether it requires a response, what the format of the reply should be, what it means if an expected data item is omitted, and so on.

None of this is a problem for messages that are used only within an application; the application developer can keep track of the details. However, protocols must be published for messages that communicate between applications. You're urged to publish the specifications for all messages your application is willing to accept from outside sources and for all those that it can package for delivery to other applications. The more that message protocols are shared, the easier it is for applications to cooperate with each other and take advantage of each other's special features.

The software kits define protocols for a number of messages. They're discussed in the *Message Protocols* appendix.

Message Ownership

Typically, when an application creates an object, it retains responsibility for it; it's up to the application to free the objects it allocates when they're no longer needed. However, BMessage objects are an exception to this rule. Whenever a BMessage is passed to the messaging mechanism, ownership is passed with it. It's a little like mailing a letter—once you drop it at the post office, it no longer belongs to you.

The system takes responsibility for a delivered BMessage object and will eventually delete it—after the receiver is finished responding to it. A message receiver can assert responsibility for a message—essentially replacing the system as its owner—by detaching it from the messaging mechanism (with BLooper's **DetachCurrentMessage()** function).

Message Loops

In the Be model, messages are delivered to threads running *message loops*. Arriving messages are placed in a queue, and are then taken from the queue one at a time. After getting a message from the queue, the thread decides how it should be handled and dispatches it to an object that can respond. When the response is finished, the thread deletes the message and takes the next one from the queue—or, if the queue is empty, waits until another message arrives.

The message loop therefore dominates the thread. The thread does nothing but get messages and respond to them; it's driven by message input.

BLooper objects set up these message loops. A BLooper spawns a thread and sets the loop in motion. Posting a message to the BLooper delivers it to the thread (places it in the queue). The BLooper removes messages from the queue and dispatches them to BHandler objects. BHandlers are the objects ultimately responsible for received messages. Everything that the thread does begins with a BHandler's response to a message.

Two hook functions come into play in this process—one defined in the BLooper class and one declared by BHandler:

- BLooper's **DispatchMessage()** function is called to pass responsibility for a message to a BHandler object. It's fully implemented by BLooper (and kit classes derived from BLooper) and is only rarely overridden by applications.

- **MessageReceived()** is the BHandler function that **DispatchMessage()** calls by default. It's up to applications to implement **MessageReceived()** functions to handle expected messages.

There's a close relationship between the BLooper role of running a message loop and the BHandler role of responding to messages. The BLooper class inherits from BHandler, so the same object can fill both roles. The BLooper is the default handler for the messages it receives.

To be notified of an arriving message, a BHandler must "belong" to the BLooper; it must have been added to the BLooper's list of eligible handlers. The list can contain any number of objects, but at any given time a BHandler can belong to only one BLooper.

While a thread is responding to a message, it keeps the BLooper that dispatched the message locked. The thread locks the BLooper before calling **DispatchMessage()** and unlocks it after **DispatchMessage()** returns.

System Messages

Special dispatching is provided for a subset of messages defined the system. These *system messages* are dispatched not by calling **MessageReceived()**, but by calling a BHandler hook function specific to the message.

System messages generally originate from within the Be operating system (from servers, the kits, or the Browser). They notify applications of external events, usually something the user has done—moved the mouse, pressed a key, resized a window, selected a document to open, or some other action of a similar sort. The command constant of the message names the event—for example, **B_KEY_DOWN**, **B_SCREEN_CHANGED**, or

B_REFS_RECEIVED—and the message may carry data describing the event. The receiver is free to respond to the message (or to not respond) in any way that's appropriate.

A few system messages name an action the receiver is expected to take, such as **B_ZOOM** or **B_ACTIVATE**. The message tells the receiver what must be done. Even these messages are prompted by an event of some kind—such as the user clicking the zoom button in a window tab or picking an application to activate from the list of running applications.

System messages have a defined format. The command constant and the names and types of data entries are fixed for each kind of message. For example, the system message that reports a user keystroke on the keyboard—a "key-down" event—has **B_KEY_DOWN** as the command constant, a "when" entry for the time of the event, a "key" entry for the key that was hit, a "modifiers" entry for the modifier keys that were down at the time, and so on.

Although the set of system-defined messages is small, they're the most frequent messages for most applications. For example, when the user types a sentence, the application receives a series of **B_KEY_DOWN** messages, one for each keystroke.

Specialized BLoopers

System messages aren't delivered to just any BLooper object. The software kits derive a few specialized classes from BLooper to give significant entities in the application their own message loops. These are the objects that receive system messages and define how they're dispatched. Each message is matched to the specific kind of BLooper that's concerned with the particular event it reports or the particular instruction it delivers. Each type of message is delivered to a specific class of object.

In particular, both the BApplication class in this kit and the BWindow class in the Interface Kit derive from BLooper. The BApplication object runs a message loop in the main thread and receives messages that concern the application as a whole—such as requests to quit the application or to open a document. Each BWindow object runs in its own thread and receives messages that report activity in the user interface—including notifications that the user typed a particular character on the keyboard, moved the cursor on-screen, or pressed a mouse button. Every window that the user sees is represented by a separate BWindow object.

Each of these classes is concerned with only a subset of system messages—BApplication with *application messages* (discussed on page 16 below) and BWindow objects with *interface messages* (discussed in the chapter on the Interface Kit). In addition, the generic BLooper class defines how a small number of *system management messages* are dispatched; these messages have to do with the messaging system itself (and are discussed on page 15 of this chapter). Each class arranges for special handling of the system messages it's concerned with.

Message-Specific Dispatching

Every system message is dispatched by calling a specific virtual "hook" function, one that's matched to the message. For example, when the Application Server sends a

B_KEY_DOWN message to the window where the user is typing, the BWindow determines which object is responsible for displaying typed characters and calls that object's **KeyDown()** virtual function. Similarly, a message that reports a user decision to shut down the application—a "quit-requested" event—is dispatched by calling the BApplication object's **QuitRequested()** function. Messages that report the movement of the cursor are dispatched by calling **MouseMoved()**, those that report a change in the screen configuration by calling **ScreenChanged()**, and so on.

These hook functions are declared in classes derived from BHandler and are often recognizable by their names. In the introductory chapter, it was explained that hook functions fall into three groups:

- Those that are left to the application to implement. These functions are named for what they announce—for what led to the function call rather than for what the function might be implemented to do. **KeyDown()** is an example.

- Those that have a default implementation to cover the common case. Like those in the first group, these functions also are named for the occurrence that prompted the function call. **ScreenChanged()** is an example.

- Those that are fully implemented to perform a particular task. These are functions that you can call, but they're also hooks that are called for you. Like most ordinary functions, they're named for what they do—like **Activate()**—not for what led to the function call.

The hook functions that are matched to system messages can fall into any of these three categories. Since most system messages report events, they mostly fall into the first two categories. The function is named for the message, and the message for the event it reports.

However, if a system message delivers an instruction for the application to do something in particular, its hook function falls into the third group. The function is fully implemented in system software, but can be overridden by the application. The function is named for what it does, and the message is named for the function.

Picking a Handler and a Hook Function

A BLooper picks a BHandler for a system message based on what the message is. For example, a BWindow calls upon the object that displays the current selection to handle a **B_KEY_DOWN** message. It asks the object in charge of the area where the user clicked to handle a **B_MOUSE_DOWN** message. And it handles messages that affect the window as a whole—such as, **B_WINDOW_RESIZED**—itself.

The BLooper identifies system messages by their command constants alone (their **what** data members). If a message is received and its command constant matches the constant for a system message, the BLooper will dispatch it by calling the message-specific hook function—regardless what data entries the message may have.

If the constant doesn't match a system message that the BLooper knows about, the message is dispatched by calling **MessageReceived()**. **MessageReceived()** is, therefore, reserved for application-defined messages. It's typically implemented to distribute the responsibility for received messages to other functions. That's something that's already taken care of for system messages, since each of them is mapped to its own function.

Application-Defined Messages

Although the system creates and delivers most messages, an application can create messages of its own and have them delivered to a chosen destination. There are three ways to initiate a message:

- Messages can be *posted* to a thread of the same application,
- They can be *sent* to a thread anywhere, generally one in a remote application, and
- They can be dragged and *dropped*.

Posting Messages

Messages are posted by calling a BLooper's **PostMessage()** function. **PostMessage()** inserts the message into the BLooper's queue so that it will be handled in sequence along with other messages the thread receives. Posting depends on the message source knowing the address of the destination BLooper; it therefore works only for application-internal messages.

Posting is how one thread of execution transfers control to another thread in the same application. Suppose, for example, that the main thread of an application (the BApplication object) receives a message requesting it to show something on-screen—begin displaying a video, say. It can create a window for this purpose, then post a message to the BWindow object telling it what to do. The BWindow receives the message and acts on it within the window's thread. After posting the message, the main thread is free to receive and respond to other messages while the window thread is busy with the video.

A thread might also post messages to itself, and thereby take advantage of the messaging mechanism to arrange its activity. This is what menu items and control devices do when they're invoked; they translate a message that reports a click or a keystroke into another, more specific message—one they could post anywhere, but typically deliver to the same thread.

Sending Messages

Messages can be posted only within an application—where the thread that calls **PostMessage()** and the thread that responds to the message are in the same address space (are part of the same "team") and may even be the same thread.

To send a message to another address space, it's necessary to first set up a BMessenger object as a local representative of the remote destination. BMessengers can be constructed in two ways:

- By naming a particular instance of a running application. The BRoster object can provide signatures and team identifiers for all running applications.
- By naming a particular BHandler object in your own application.

The first method constructs a BMessenger that can send messages to the main thread of the remote application, where they'll be received and handled by its BApplication object.

The second method constructs a BMessenger that's tied to a BHandler in your own application. However, you can place the BMessenger in a message and send it to a remote application. That application can then employ the BMessenger to target messages to your BHandler. The messages are delivered to whatever BLooper the BHandler belongs to; the BLooper dispatches the message to the BHandler.

Thus, a BMessenger can be seen as a local identifier for a remote BLooper/BHandler pair. Calling the object's **SendMessage()** function delivers the message to the remote destination.

(BMessengers can send messages to local destinations as well as to a remote ones. However, it's more efficient to post a local message than to send it.)

Dropping a Message

Through a service of the Interface Kit, users can drag messages from a source location and drop them on a chosen destination, perhaps in another application. The source application puts the message together and hands it over to the Application Server, which tracks where the user drags it.

When the user drops the message inside a window somewhere, the Server delivers it to the BWindow object and targets it to the BView (a kind of BHandler) that's in charge of the portion of the window where the message was dropped. The message is placed in the BWindow's queue and is dispatched like all other messages. In contrast to messages that are posted or sent in application code, only the user determines the destination of a dragged message.

A message receiver can discover whether and where a message was dropped by calling the BMessage object's **WasDropped()** and **DropPoint()** functions.

See "Drag and Drop" on page 235 in *The Interface Kit* chapter for details on how to initiate a drag-and-drop session.

Two-Way Communication

A delivered BMessage carries a return address with it < with the current exception of messages that are posted >. The message receiver can reply to the message by calling the BMessage's **SendReply()** function. Replies can be synchronous or asynchronous:

- A message sender can ask for a synchronous reply when calling the sending function. For example:

  ```
  BMessage *reply;
  myMessenger->SendMessage(message, &reply);
  if (reply->what != B_NO_REPLY ) {
      . . .
  }
  ```

  In this case, **SendMessage()** waits for the reply; it doesn't return until one is received. (In case the message receiver refuses to cooperate, a default reply is sent when the original message is deleted.)

  A message receiver can discover whether the sender is waiting for a synchronous reply by calling the BMessage's **IsSourceWaiting()** function.

- A message sender can provide for an asynchronous reply by designating a BHandler object for the return message. For example:

  ```
  myMessenger->SendMessage(message, someHandler);
  ```

  In this case, the sending function doesn't wait for the reply; the reply message will be directed to the named BHandler. An asynchronous reply is always possible. If a BHandler isn't designated for it, the BApplication object will act as the default handler.

BMessage's **SendReply()** function has the same syntax as **SendMessage()**, so it's possible to ask for a synchronous reply to a message that is itself a reply,

```
BMessage *reply;
theMessage->SendReply(message, &reply);
if (reply->what != B_NO_REPLY ) {
    . . .
}
```

or to designate a BHandler for an asynchronous reply to the reply:

```
theMessage->SendReply(message, someHandler);
```

In this way, two applications can maintain an ongoing exchange of messages.

You can also name a target BHandler for an asynchronous reply to a dragged message.
< There is currently no provision for replying to a posted message. >

Specifying the Target

All messages have target BHandlers, whether explicitly or implicitly expressed.

- When posting a message to a BLooper, you can name a target BHandler for it. The BLooper is the default target.
- Sending a message targets it to the remote BApplication object or to the particular BHandler that was used to construct the BMessenger.
- Dropped messages are targeted to the object (a BView) that owns the piece of window real estate where the cursor was located when the message was dropped.

The target is respected when the message is dispatched; the dispatcher always calls a hook function belonging to the designated BHandler. If the message matches one that the system defines and the target BHandler is the kind of object that's expected to handle that type of message, the dispatcher will call the target's message-specific hook function. However, if the designated target isn't the handler of design for the message, the BLooper will call its **MessageReceived()** function.

For example, if a **B_KEY_DOWN** message is posted to a BWindow object and a BView is named as the target, the BWindow will dispatch the message by calling the BView's **KeyDown()** function. However, if the BWindow itself is named as the target, it will dispatch the message by calling its own **MessageReceived()** function. BView objects are expected to handle keyboard messages; BWindows are not.

Preferred Handlers

By implementing a **PreferredHandler()** function, a BLooper can name the BHandler it prefers to be the target of the messages it receives. You can follow this recommendation when posting a message < but currently not when sending a message >, or you can ignore it. The preferred handler typically changes from time to time. Choosing the preferred handler is therefore a way of determining the message target at run time. For example, a BWindow's preferred handler is the object in charge of the current selection; it changes as the user changes the selection.

Message Filters

Incoming messages can be filtered before they're dispatched to a BHandler. You can arrange to have a filtering function examine the message before the BHandler's hook function is called.

The filtering function is contained in a BMessageFilter object, which also holds the criteria for when the filter should apply. The function, called **Filter()**, is defined in classes derived from BMessageFilter.

If a BMessageFilter is attached to a BHandler, it filters only messages destined for that BHandler. It it's attached as a common filter to a BLooper object, it can filter any message that the BLooper dispatches, no matter what the handler. (In addition to the list of common filters, a BLooper can, like other BHandlers, maintain a list of filters specific to its role as a target handler.)

System Messages in the Application Kit

Although the Application Kit implements the messaging mechanism and defines all the system messages, it handles only a few of them itself. Each system message has a particular import and falls within the scope of a particular kind of BLooper object. Most are associated with BWindow objects in the Interface Kit. But there are two BLooper classes in the Application Kit; each handles its own subset of system messages:

- The generic BLooper class handles *system management messages* that help run the messaging mechanism. There are just two such messages.
- The BApplication class handles *application messages* that are not the province any particular window, but concern the application as a whole. The system defines nine different application messages.

System Management Messages

The BLooper class takes care of just two system messages; both are concerned with running the messaging mechanism:

- A **B_QUIT_REQUESTED** message asks the BLooper to quit the message loop and destroy itself. Classes derived from BLooper reinterpret this message in their own way. For the BApplication object, it's a request to quit the application. For a BWindow, it's a request to close the window. However, generically, it's simply a request to get rid of a BLooper object.
- A **B_HANDLERS_REQUESTED** message asks a target BHandler to supply BMessenger objects for other BHandlers. The correct response is to send a **B_HANDLERS_INFO** message in reply—with the BMessengers installed in a "handlers" array or with an error code in an "error" entry. The BMessengers can be used to target particular objects within the responding application.

The BLooper object dispatches these messages by calling a hook function matched to the message. The following table lists the hook functions that are called to initiate a response to system management messages and the base classes where those functions are declared:

Message type	Virtual function	Class
B_QUIT_REQUESTED	**QuitRequested()**	BLooper
B_HANDLERS_REQUESTED	**HandlersRequested()**	BHandler

Although it defines how these messages are treated, nothing in the Be operating system produces the message itself. It's up to applications to create the messages and arrange for their delivery.

See "System Management Messages" in the *Message Protocols* appendix for information on the content of system management messages, particularly **B_HANDLERS_REQUESTED**.

Application Messages

The nine application messages are an assortment of various reports and requests. One message delivers an instruction:

- A **B_ACTIVATE** instruction tells the application to activate itself—to become the active application. This message permits one application (usually the Browser) to activate another.

All the other application messages report events. Two of them notify the application of a change in its status:

- A **B_READY_TO_RUN** message reports that the application has finished launching and configuring itself and its main thread is ready to respond to messages.
- A **B_APP_ACTIVATED** message is delivered when the application becomes the active application—the one that the user is currently engaged with—or when it relinquishes that status to another application.

Two of the messages are requests that the application typically makes of itself:

- A **B_QUIT_REQUESTED** message is taken by the BApplication object to be a request to shut the entire application down, not just one thread. An application that has a user interface usually interprets some user action (such as clicking a "Quit" menu item) as a request to quit and, in response, posts a **B_QUIT_REQUESTED** message to the BApplication object. An application that is the servant of other applications may get the request from a remote source.
- A **B_ABOUT_REQUESTED** message requests information about the application, usually through an "About . . ." item in the application's main menu. The application should set up this item to post a **B_ABOUT_REQUESTED** message to the BApplication object. In response, the BApplication object should display a window with general information about the application.

Other application messages report information from remote sources:

- A **B_ARGV_RECEIVED** message is delivered either on-launch or after-launch when the application receives strings of characters the user typed on the command line, or when the application is launched by another application and is passed a similar array of character strings.
- A **B_REFS_RECEIVED** message passes the application one or more references to database records. Typically, this means the user has chosen some files from the file panel, double-clicked a document icon in the Browser, or dragged the icon and dropped in on the application icon.

- A **B_PANEL_CLOSED** message is sent by the file panel when the panel is removed from the screen.

The system is the source of one repeated message:

- Periodic **B_PULSE** messages are posted at regularly spaced intervals. They can be used to arrange repeated actions when precise timing is not critical.

All application messages are received by the BApplication object in the main thread. The BApplication object dispatches them all to itself; it doesn't delegate them to any other handler. The following chart lists the hook functions that are called to initiate the application's response to system messages and the base class where each function is declared:

Message type	Virtual function	Class
B_ACTIVATE	**Activate()**	BApplication
B_READY_TO_RUN	**ReadyToRun()**	BApplication
B_APP_ACTIVATED	**AppActivated()**	BApplication
B_QUIT_REQUESTED	**QuitRequested()**	BLooper
B_ABOUT_REQUESTED	**AboutRequested()**	BApplication
B_ARGV_RECEIVED	**ArgvReceived()**	BApplication
B_REFS_RECEIVED	**RefsReceived()**	BApplication
B_PANEL_CLOSED	**FilePanelClosed()**	BApplication
B_PULSE	**Pulse()**	BApplication

QuitRequested() is first declared in the BLooper class. BApplication reinterprets it—and reimplements it—to mean a request to quit the whole application, not just one of its threads.

Only four application messages—**B_APP_ACTIVATED**, **B_ARGV_RECEIVED**, **B_REFS_RECEIVED**, and **B_PANEL_CLOSED**—contain any data; the rest are empty. See "Application Messages" in the *Message Protocols* appendix for details on the content of these messages.

Setting Up an Application

There are just a couple of things that an application must do if it's to take its place as a well-known and cooperative resident on the BeBox:

- Internally, it needs a BApplication object, and
- Externally, it needs to publicize information about itself.

The BApplication object is essential; every application must have one to handle messages from other applications, particularly the Browser. However, it's not sufficient by itself. In addition, the application must provide:

- Icons that represent the application, and represent documents and other files associated with the application.
- An identifying signature for the application.
- Information about the application's behavior, including a strategy for how it can be launched.

The icons, signature, and behavioral information are all stored in the same resources file as the executable binary. By locating them in resources, they become available even when the application isn't running.

Although these bits of information don't strictly belong to the Application Kit, they're relevant to how parts of the Kit work and, possibly, to how you design your application. They're therefore discussed here.

Use the Icon World application to set up application resources, as described in *The Be User's Guide*, published separately.

Icons

Every application needs an icon to represent it (in a Browser window, for example). It should provide a large (32 pixel × 32 pixel) version of the icon and a smaller (16 pixel × 16 pixel) version. This can be done by creating the icons in Icon World or by importing icons created elsewhere. Either way, Icon World will construct highlighted versions of both the small and large icons and install them all in resources of type 'ICON' (for the large version) and 'MICN' (for the "mini-icon").

If an application opens documents or has other associated files, it should provide large and small icons for them as well.

Application Information

An application-information resource (named "app info" and typed 'APPI') holds other information that needs to be available—especially to the Browser—whether or not the application is running. This resource advertises the application's signature and its launch behavior, and possibly other behavioral idiosyncrasies as well. You can create it in Icon World's App Info menu.

Signatures

A signature is simply a **long** integer that identifies an application. No two applications should have the same signature.

To make sure that the signature for your application is unique, you should register it with—or obtain it from—Be's Developer Support services (**devsupport@be.com** or, in a pinch, 1 (415) 462-4103). We'll try to make sure that no one else adopts the same signature.

Use Icon World's App Info menu to install the signature in the resource.

Launch Information

There are three possible launch behaviors that you can choose for your application. Each possibility is represented by a constant:

B_MULTIPLE_LAUNCH	Several instances of the application can be running at once. It can be launched any number of times from the same executable file. This is the normal behavior for most utilities, such as the compiler, **tar**, or Heap Watch. It's also appropriate for an application that can deal with only one document at a time, and therefore must be launched anew each time it's asked to handle another file.
B_SINGLE_LAUNCH	Normally, only one instance of the application can be running. However, if the user copies the executable file for the application, it can be launched once from each copy. This is the normal behavior for most applications, including applications that can deal with more than one document at a time.
B_EXCLUSIVE_LAUNCH	When the application is running, no other instance of the same application can be launched from any source. This is appropriate for applications that require exclusive ownership of a system resource, like the telephone line.

In other words, **B_EXCLUSIVE_LAUNCH** applications are restricted by signature—only one instance of an application with that particular signature can be running at any given time. **B_SINGLE_LAUNCH** applications are restricted by executable file—there can be only one instance of an application launched from that particular executable. **B_MULTIPLE_LAUNCH** applications are unrestricted.

These categories affect how the Browser launches applications and communicates with them. In the Browser, a user can launch an application by picking the application itself or by picking one of its documents. Double-clicking an application icon picks the application, and double-clicking a document icon picks the document. Dragging a document icon and dropping it on the application icon picks both.

Whenever the user picks a **B_MULTIPLE_LAUNCH** application or picks one of its documents, the Browser always launches it anew. It doesn't matter whether another instance of the application is already running or not.

However, when the user picks a **B_SINGLE_LAUNCH** application, the Browser launches it only if an application launched from the same executable file isn't already running. Otherwise, it activates the running application. Similarly, when the user picks a document for a **B_SINGLE_LAUNCH** application, the Browser matches the document to an executable file and launches it only if a running application hasn't been launched from the same file. If one has been launched from the file, the Browser merely activates it and sends it a message identifying the document.

B_EXCLUSIVE_LAUNCH is even more restrictive than **B_SINGLE_LAUNCH**. When the user picks a **B_EXCLUSIVE_LAUNCH** application, or the document for a **B_EXCLUSIVE_LAUNCH** application, the Browser launches it only if an application with the same signature isn't already running.

Most applications don't need the extreme restrictiveness of **B_EXCLUSIVE_LAUNCH** and should choose between **B_SINGLE_LAUNCH** and **B_MULTIPLE_LAUNCH**. The choice should be informed by whether the application can have more than one file open at a time, whether multiple instances of the same application would make sense to the user, whether windows belonging to one instance might be confused for windows belonging to another instance, and similar considerations.

The best place to choose a launch behavior for your application is in IconWorld's App Info menu. If a choice isn't made, IconWorld picks **B_SINGLE_LAUNCH** by default. If an application doesn't have an application information resource, it's treated as being **B_MULTIPLE_LAUNCH** by default.

Other Information

Resources can also publicize two other behaviors, similarly designated by constants:

B_ARGV_ONLY	The application doesn't participate in the messaging system. Therefore, the only information it can receive are command-line arguments, *argc* and *argv*, passed to the **main()** function. **B_ARGV_ONLY** is assumed if the application doesn't have a BApplication object.
B_BACKGROUND_APP	The application doesn't have a user interface and therefore shouldn't appear in the Browser's application list.

BApplication

Derived from:	public BLooper
Declared in:	<app/Application.h>

Overview

The BApplication class defines an object that represents and serves the entire application. Every Be application must have one (and only one) BApplication object. It's usually the first object the application constructs and the last one it deletes.

The BApplication object has these primary responsibilities:

- *It makes a connection to the Application Server.* Any application that puts a window on-screen or relies on other system services needs this connection. It's made automatically when the BApplication object is constructed.

- *It runs the application's main message loop.* The BApplication object is a kind of BLooper, but instead of spawning an independent thread, it runs a message loop in the application's main thread (the thread that the **main()** function executes in). This loop receives and processes messages that affect the entire application, including the initial messages received from remote applications. It gets several messages from the Browser (such as reports of what documents to open). Any application that's known to the Browser or that cooperates with other applications needs a main message loop.

- *It's the home for application-wide elements of the user interface.* For example, it sets up the application's main menu and runs the file panel, which permits users to navigate the file system and pick files to open. It also lets you set, hide, and show the application's cursor. The ability to define the look of the cursor is provided by BApplication's **SetCursor()** function.

  The user interface mainly centers on windows and is defined in the Interface Kit. The BApplication object merely contains the elements that are common to all windows and specific to the application.

Derived Classes

BApplication typically serves as the base class for a derived class that specializes it and extends it in ways that are appropriate for a particular application. It declares (and inherits declarations for) a number of hook functions that you can implement in a derived class to augment and fine-tune what it does.

For example, your application might implement a **RefsReceived()** function to open a document and display it in a window, or a **ReadyToRun()** function to finish initializing the application after it has been launched and has started to receive messages. These two functions, like a handful of others, are called in response to system messages that have application-wide import. Hook functions for application messages were discussed in the introduction on page 17.

If you expect your application to get messages from remote sources, or its main thread to get messages from other threads in the application, you should also implement a **MessageReceived()** function to sort through them as they arrive.

A derived class is also a good place to record the global properties of your application and to define functions that give other objects access to those properties.

Constructing the Object and Running the Message Loop

The BApplication object must be constructed before the application can begin running or put a user interface on-screen. Other objects in other kits depend on the BApplication object and its connection to the Application Server. In particular, you can't construct BWindow objects in the Interface Kit until the BApplication object is in place.

Simply constructing the BApplication object forms the connection to the Server. The connection is severed when you quit the application and delete the object.

be_app

The BApplication constructor assigns the new object to a global variable, **be_app**. This assignment is made automatically—you don't have to create the variable or set its value yourself. **be_app** is declared in **app/Application.h** and can be used throughout the code you write (or, more accurately, all code that directly or indirectly includes **Application.h**).

The **be_app** variable is typed as a pointer to an instance of the BApplication class. If you use a derived class instead—as most applications do—you have to cast the **be_app** variable when you call a function that's implemented by the derived class.

```
    ((MyApplication *)be_app)->DoSomethingSpecial();
```

Casting isn't required to call functions defined in the BApplication class (or in the BHandler and BLooper classes it inherits from), nor is it required for virtual functions defined in a derived class but declared by BApplication (or by the classes it inherits from).

main()

Because of its pivotal role, the BApplication object is one of the first objects, if not the very first object, the application creates. It's typically created in the **main()** function. The job of **main()** is to set up the application and turn over its operation to the various message loops run by particular objects, including the main message loop run by the BApplication object.

After constructing the BApplication object (and the other objects that your application initially needs), you tell it to begin running the message loop by calling its **Run()** function. Like the **Run()** function defined in the BLooper class, BApplication's **Run()** initiates a message loop and begins processing messages. However, unlike the BLooper function, it doesn't spawn a thread; rather, it takes over the main application thread. Because it runs the loop in the same thread in which it was called, **Run()** doesn't return until the application is told to quit.

At its simplest, the **main()** function of a Be application would look something like this:

```
#include <app/Application.h>

main()
{
    . . .
    new BApplication('abcd');
    . . .
    be_app->Run();
    delete be_app;
}
```

The number passed to the constructor ('abcd') sets the application's signature. This is just a precautionary measure. It's more common (and much better) to set the signature at compile time in a resource. If there is a resource, that signature is used and the one passed to the constructor is ignored.

The **main()** function shown above doesn't allow for the usual command-line arguments, *argc* and *argv*. It would be possible to have **main()** parse the *argv* array, but these arguments are also packaged in a **B_ARGV_RECEIVED** message that the application gets immediately after **Run()** is called. Instead of handling them within **main()**, applications generally implement an **ArgvReceived()** function to do the job. This function can also handle command-line arguments that are passed to the application after it has been launched; it can be called at any time while the application is running.

Configuration Messages Received on Launch

When an application is launched, it may be passed messages that affect how it configures itself. These are the first messages that the BApplication object receives after **Run()** is called.

For example, when the user double-clicks a document icon to launch an application, the Browser passes the application a **B_REFS_RECEIVED** message with information about the document. When launched from the command line, the application gets a **B_ARGV_RECEIVED** message listing the command-line arguments. When launched by the BRoster object, it might receive an arbitrary set of configuration messages.

After all the messages passed on-launch have been received and responded to, the application gets a **B_READY_TO_RUN** message and its **ReadyToRun()** hook function is called. This is the appropriate place to finish initializing the application before it begins running in earnest. It's the application's last chance to present the user with its initial user interface. For example, if a document has not already been opened in response to an on-launch **B_REFS_RECEIVED** message, **ReadyToRun()** could be implemented to place a window with an empty document on-screen.

ReadyToRun() is always called to mark the transition from the initial period when the application is being launched to the period when it's up and running—even if it's launched without any configuration messages. The **IsLaunching()** function can let you know which period the application is in.

Quitting

The main message loop terminates and **Run()** returns when **Quit()** is called. Because **Run()** doesn't spawn a thread, **Quit()** merely breaks the loop; it doesn't kill the thread or destroy the object (unlike BLooper's version of the function).

Quit() is usually called indirectly, as a byproduct of a **B_QUIT_REQUESTED** message posted to the BApplication object. The application is notified of the message through a **QuitRequested()** function call; it calls **Quit()** if **QuitRequested()** returns **TRUE**.

When **Run()** returns, the application is well down the path of terminating itself. **main()** simply deletes **be_app**, cleans up anything else that might need attention, and exits.

Aborted Run

Applications with restricted launch behavior (**B_EXCLUSIVE_LAUNCH** and **B_SINGLE_LAUNCH**) may be launched anyway in violation of those restrictions. When this happens, the **Run()** function returns abruptly without processing any messages and the application quits as it normally does when **Run()** returns. Messages that carried on-launch information for the aborted application are redirected to the instance of the application that's already running.

Applications should be prepared for their **main()** functions to be executed in this abortive manner and guard against any undesired consequences.

Locking

You sometimes have to coordinate access to the BApplication object, since a single object serves the entire application and different parts of the application (windows, in particular) will be running in other threads. Locking ensures that one thread won't change the state of the application while another thread is changing the same aspect (or even just trying to examine it).

The BApplication object is locked automatically while the main thread is responding to a message, but it may have to be explicitly locked at other times.

This class inherits the locking mechanism—the **Lock()**, **Unlock()**, and **LockOwner()** functions—from BLooper. See that class for details.

Hook Functions

AboutRequested()	Can be implemented to present the user with a window containing information about the application.
Activate()	Activates the application by making one of its windows the active window; can be reimplemented to activate the application in some other way.
AppActivated()	Can be implemented to do whatever is necessary when the application becomes the active application, or when it loses that status.
ArgvReceived()	Can be implemented to parse the array of command-line arguments (or a similar array of argument strings).
FilePanelClosed()	Can be implemented to take note when the file panel is closed.
MenusWillShow()	Can be implemented to update the menus in the application's main menu hierarchy, just before they're shown on-screen.
Pulse()	Can be implemented to do something over and over again. **Pulse()** is called repeatedly at roughly regular intervals in the absence of any other activity in the main thread.
ReadyToRun()	Can be implemented to set up the application's running environment. This function is called after all messages the application receives on-launch have been responded to.
RefsReceived()	Can be implemented to respond to a message that contains references to database records. Typically, the records are for documents that the application is being asked to open.

VolumeMounted()	Can be implemented to take note when a new volume (a floppy disk, for example) is mounted.
VolumeUnmounted()	Can be implemented to take whatever action is necessary just before a volume is unmounted.

Constructor and Destructor

BApplication()

BApplication(ulong *signature***)**

Establishes a connection to the Application Server, assigns *signature* as the application identifier if one hasn't already been set, and initializes the application-wide variable **be_app** to point to the new object.

The *signature* that's passed becomes the application identifier only if a signature hasn't been set in a resource file. It's preferable to assign the signature in a resource at compile time, since that enables the system to associate the signature with the application even when it's not running.

Every application must have one and only one BApplication object, typically an instance of a derived class. It's usually the first object that the application creates.

~BApplication()

virtual **~BApplication(**void**)**

Closes the application's windows, if it has any, without giving them a chance to disagree, kills the window threads, frees the BWindow objects and the BViews they contain, and severs the application's connection to the Application Server.

You can delete the BApplication object only after **Run()** has exited the main message loop. In the normal course of events, all the application's windows will already have been closed and freed by then.

See also: the BWindow class in the Interface Kit, **QuitRequested()**

Member Functions

AboutRequested()

virtual void **AboutRequested(**void**)**

Implemented by derived classes to put a window on-screen that provides the user with information about the application. The window typically displays copyright data, the version number, license restrictions, the names of the application's authors, a simple description of what the application is for, and similar information.

This function is called when the user operates the "About . . ." item in the main menu and a **B_ABOUT_REQUESTED** message is posted to the application as a result.

To set up the menu item, assign it a model message with **B_ABOUT_REQUESTED** as the command constant and the BApplication object as the target, as illustrated in the **SetMainMenu()** description on page 38. The default main menu includes such an item.

See also: **SetMainMenu()**, the BMenu class in the Interface Kit

Activate()

virtual void **Activate(**void**)**

Makes the application the active application by arbitrarily picking one of its windows and making it the active window. If the application doesn't have any windows, or if the chosen window happens to be hidden, the attempted activation will fail. < A surer method of activation will be provided in a future release. >

This function is called when the main thread receives a **B_ACTIVATE** message, which any application can send to any other application. The Browser uses this method to activate a running application when, for example, the user double-clicks its icon or selects it from the application menu.

However, **Activate()** is not called when the application is first launched or when the user makes one of its windows the active window. Therefore don't rely on it as a way of being notified that the application has become active. Rely on **AppActivated()** instead.

See also: **activate_app()** and **BWindow::Activate()** in the Interface Kit, **AppActivated()**

AppActivated()

virtual void **AppActivated(**bool *isActive***)**

Implemented by derived classes to take note when the application becomes—or ceases to be—the active application. The application has just attained that status if the *isActive* flag is **TRUE**, and just lost it if the flag is **FALSE**. The active application is the one that owns the current active window and whose main menu is accessible through the icon displayed at the left top corner of the screen.

< Currently, this function is called only when the change in active application is a consequence of a window being activated. It can be called while an application is being launched, provided that the application puts a window on-screen. However, it's always called after **ReadyToRun()**, not before. >

See also: **BWindow::WindowActivated()** in the Interface Kit, "**B_APP_ACTIVATED**" on page 5 in the *Message Protocols* appendix

ArgvReceived()

virtual void **ArgvReceived(**int *argc*, char ***argv***)**

Implemented by derived classes to respond to a **B_ARGV_RECEIVED** message that passes the application an array of argument strings, typically arguments typed on the command line. *argv* is a pointer to the strings and *argc* is the number of strings in the array. These parameters are identical to those traditionally associated with the **main()** function.

When an application is launched from the command line, the command-line arguments are both passed to **main()** and packaged in a **B_ARGV_RECEIVED** message that's sent to the application on-launch (before **ReadyToRun()** is called). When BRoster's **Launch()** function is passed *argc* and *argv* parameters, they're similarly bundled in an on-launch message.

An application might also get **B_ARGV_RECEIVED** messages after it's launched. For example, imagine a graphics program called "Splotch" that can handle multiple documents and is therefore restricted so that it can't be launched more than once (it's a **B_SINGLE_LAUNCH** or a **B_EXCLUSIVE_LAUNCH** application). If the user types

```
Splotch myArtwork
```

in a shell, it launches the application and passes it an on-launch **B_ARGV_RECEIVED** message with the strings "Splotch" and "myArtwork". Then, if the user types

```
Splotch yourArtwork
```

the running application is again informed with a **B_ARGV_RECEIVED** message. In both cases, the BApplication object dispatches the message by calling this function.

To open either of the artwork files, the Splotch application will need to translate the document pathname into a database reference. It can do this most easily by calling **get_ref_for_path()**, defined in the Storage Kit.

See also: **RefsReceived()**, "**B_ARGV_RECEIVED**" on page 5 in the *Message Protocols* appendix

CloseFilePanel() *see* RunFilePanel()

CountWindows()

long **CountWindows(**void**)** const

Returns the number of windows belonging to the application. The count includes only windows that the application explicitly created. It omits, for example, the private windows used by BBitmap objects.

See also: the BWindow class in the Interface Kit

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BHandler **target***)**

Augments the BLooper function to dispatch system messages by calling a specific hook function. The set of system messages that the BApplication object receives and the hook functions that it calls to respond to them are listed under “Application Messages” on page 16 of the chapter introduction.

Other messages—those defined by the application rather than the Application Kit—are forwarded to the *target* BHandler’s **MessageReceived()** function. Note that the *target* is ignored for most system messages.

DispatchMessage() locks the BApplication object and keeps it locked until the main thread has finished responding to the message.

You can override this function to dispatch your own messages differently.

See also: **BLooper::DispatchMessage()**, **BHandler::MessageReceived()**

FilePanelClosed()

virtual void **FilePanelClosed(**BMessage **message***)**

Implemented by derived classes to take note when the file panel is closed. The *message* argument contains information about how the panel was closed and its state at the time. It has **B_PANEL_CLOSED** as its **what** data member and may include entries under the names “frame” (the last frame rectangle of the panel), “directory” (the last directory it displayed), “marked” (the item that was marked in its list of filters), and “canceled” (whether the user closed the panel). Some of this information can be retained to configure the panel the next time it runs.

See also: “**B_PANEL_CLOSED**” on page 5 in the *Message Protocols* appendix, **RunFilePanel()**

GetAppInfo()

long **GetAppInfo(**app_info **theInfo***)** const

Writes information about the application into the **app_info** structure referred to by *theInfo*. The structure contains the application signature, the identifier for its main thread, a reference to its executable file in the database, and other information.

This function is the equivalent to the identically-named BRoster function—or, more accurately, to BRoster's **GetRunningAppInfo()**—except that it only provides information about the current application. The following code

```
app_info info;
if ( be_app->GetAppInfo(&info) == B_NO_ERROR )
     . . .
```

is simply a shorthand for:

```
app_info info;
if ( be_roster->GetRunningAppInfo(be_app->Team(),
                                              &info) == B_NO_ERROR )
     . . .
```

GetAppInfo() returns **B_NO_ERROR** if successful, and an error code if not.

See the BRoster function for the error codes and for a description of the information contained in an **app_info** structure.

See also: **BRoster::GetAppInfo()**

HandlersRequested()

virtual void **HandlersRequested(**BMessage **message***)**

Responds to a **B_HANDLERS_REQUESTED** *message* by sending a **B_HANDLERS_INFO** reply. The reply supplies a BMessenger object for each requested BHandler that's associated with the BApplication object. The BMessengers are placed in the reply message under the name "handlers".

This version of **HandlersRequested()** interprets the request for handlers as a request for BLoopers belonging to the application. If the request *message* has an entry named "class" containing the string "BWindow", it limits the search for BLoopers to BWindow objects belonging to the application. If the BWindow class isn't specified, the search encompasses all BLoopers belonging to the BApplication, including BWindow objects.

If the *message* has an entry named "index", this function supplies a BMessenger for the BLooper at that index in the list of the application's BLoopers (or the BWindow at that index in the application's window list). If there's no "index" entry, but there is one labeled "name", it supplies a BMessenger for the BLooper (or BWindow) with the specified name.

If it can't find a BLooper (or BWindow) at the specified "index" or with the requested "name", this function doesn't supply any BMessengers, but rather puts the **B_BAD_INDEX** or **B_NAME_NOT_FOUND** error constant in the reply message in an entry named "error".

If neither an "index" nor a "name" is specified, it places BMessengers for all the application's BLoopers (or BWindows) in the "handlers" array. Failing that, it places **B_ERROR** in an "error" entry.

You can override this function to use a different protocol for requesting handlers, or to prevent the BApplication object from revealing information about any or all of its BLoopers.

See also: **BLooper::HandlersRequested()**

HideCursor(), ShowCursor(), ObscureCursor()

void **HideCursor(**void**)**

void **ShowCursor(**void**)**

void **ObscureCursor(**void**)**

HideCursor() removes the cursor from the screen. **ShowCursor()** restores it. **ObscureCursor()** hides it temporarily, until the user moves the mouse.

See also: **SetCursor()**, **IsCursorHidden()**

IsCursorHidden()

bool **IsCursorHidden(**void**)** const

Returns **TRUE** if the cursor is hidden (but not obscured), and **FALSE** if not.

See also: **HideCursor()**

IsFilePanelRunning() *see* RunFilePanel()

IsLaunching()

bool **IsLaunching(**void**)** const

Returns **TRUE** if the application is in the process of launching—of getting itself ready to run—and **FALSE** once the **ReadyToRun()** function has been called.

IsLaunching() can be called while responding to a message to find out whether the message was received on-launch (to help the application configure itself) or after-launch as an ordinary message.

See also: **ReadyToRun()**

MainMenu() *see* SetMainMenu()

MenusWillShow()

virtual void **MenusWillShow(**void**)** const

Implemented by derived classes to make any necessary changes to the menus in the hierarchy controlled by the application's main menu before any of them is shown to the user. **MenusWillShow()** is called each time the main menu is placed on-screen, just before it's made visible.

See also: **BWindow::MenusWillShow()** in the Interface Kit, **SetMainMenu()**

ObscureCursor() *see* HideCursor()

Pulse()

virtual void **Pulse(**void**)**

Implemented by derived classes to do something at regular intervals. **Pulse()** is called regularly as the result of **B_PULSE** messages, as long as no other messages are pending. By default, pulsing is disabled—the pulse rate is set to 0.0—but you can enable it by calling the **SetPulseRate()** function to set an actual rate.

You can implement **Pulse()** to do whatever you want. However, pulse events aren't accurate enough for actions that require precise timing.

The default version of this function is empty.

See also: **BWindow::Pulse()** in the Interface Kit, **SetPulseRate()**

Quit()

virtual void **Quit(**void**)**

Kills the application by terminating the message loop and causing **Run()** to return. You rarely call this function directly; it's called for you when the application receives a **B_QUIT_REQUESTED** message and **QuitRequested()** returns **TRUE** to allow the application to shut down.

BApplication's **Quit()** differs from the BLooper function it overrides in four important respects:

- It doesn't kill the thread. It merely causes the message loop to exit after it finishes with the current message.
- It therefore always returns, even when called from within the main thread.
- It returns immediately. It doesn't wait for the message loop to exit.
- It doesn't delete the object. It's up to you to delete it after **Run()** returns. (However, for some reason, **Quit()** *does* delete the BApplication object if it's called when no message loop is running.)

Before shutting down, the BApplication object responds to every message it received prior to the **Quit()** call.

See also: **BLooper::Quit()**, **QuitRequested()**

QuitRequested()

virtual bool **QuitRequested(**void**)**

Overrides the BLooper function to decide whether the application should really quit when requested to do so.

BApplication's implementation of this function tries to get the permission of the application's windows before agreeing to quit. It works its way through the list of BWindow objects that belong to the application and forwards the **QuitRequested()** call to each one. If a BWindow agrees to quit (its **QuitRequested()** function returns **TRUE**), the BWindow version of **Quit()** is called to destroy the window. If the window refuses to quit (its **QuitRequested()** function returns **FALSE**), the attempt to destroy the window fails and no other windows are asked to quit.

If it's successful in terminating all the application's windows (or if the application didn't have any windows to begin with), this function returns **TRUE** to indicate that the application may quit; if not, it returns **FALSE**.

An application can replace this window-by-window test of whether the application should quit, or augment it by adding a more global test. It might, for example, put a modal window on-screen that gives the user the opportunity to save documents, terminate on-going operations, or cancel the quit request.

This hook function is called for you when the main thread receives a **B_QUIT_REQUESTED** message; you never call it yourself. However, you *do* have to post the **B_QUIT_REQUESTED** message. Typically, the application's main menu has an item labeled "Quit." When the user invokes the item, it should post a **B_QUIT_REQUESTED** message directly to the BApplication object.

See also: **BLooper::QuitRequested()**, **Quit()**, **SetMainMenu()**

ReadyToRun()

virtual void **ReadyToRun(**void**)**

Implemented by derived classes to complete the initialization of the application. This is a hook function that's called after all messages that the application receives on-launch have been handled. It's called in response to a **B_READY_TO_RUN** message that's posted immediately after the last on-launch message. If the application isn't launched with any messages, **B_READY_TO_RUN** is the first message it receives.

This function is the application's last opportunity to put its initial user interface on-screen. If the application hasn't yet displayed a window to the user (for example, if it hasn't opened a document in response to an on-launch **B_REFS_RECEIVED** or **B_ARGV_RECEIVED** message), it should do so in **ReadyToRun()**.

The default version of **ReadyToRun()** is empty.

See also: **Run()**, **IsLaunching()**

RefsReceived()

virtual void **RefsReceived(**BMessage **message***)**

Implemented by derived classes to do something with one or more database records that have been referred to the application in a *message*. The message has **B_REFS_RECEIVED** as its **what** data member and a single data entry named "refs" that contains one or more **record_ref** (**B_REF_TYPE**) items.

Typically, the records are for documents that the application is requested to open. For example, unless an alternative message is specified, the user's selections in the file panel are reported to the application in a **B_REFS_RECEIVED** message. Similarly, when the user double-clicks a document icon in a Browser window, the Browser sends a **B_REFS_RECEIVED** message to the application that owns the document. In each case, the BApplication object dispatches the message by passing it to this function.

You can use the Storage Kit's **does_ref_conform()** function to discover what kind of record each item in the "refs" entry refers to. For example:

```
void MyApplication::RefsReceived(BMessage *message)
{
    ulong type;
    long count;
    . . .
    message->GetInfo("refs", &type, &count);
    for ( long i = --count; i >= 0; i-- ) {
        record_ref item = message->FindRef("refs", i);
        if ( item.database >= 0 && item.record >= 0 ) {
            if ( does_ref_conform(item, "File") ) {
                BFile file;
                file.SetRef(item);
                if ( file.Open() == B_NO_ERROR )
                    . . .
            }
            else {
                BRecord *record = new BRecord(item);
                . . .
            }
        }
    }
    . . .
}
```

REFS_RECEIVED messages can be received both on-launch (while the application is configuring itself) or after-launch (as ordinary messages received while the application is running).

See also: **does_ref_conform()** in the Storage Kit, **ArgvReceived()**, **ReadyToRun()**, **IsLaunching()**, "**B_REFS_RECEIVED**" on page 6 in the *Message Protocols* appendix

Run()

virtual thread_id **Run(**void**)**

Runs a message loop in the application's main thread. This function must be called from **main()** to start the application running. The loop is terminated and **Run()** returns when **Quit()** is called, or (potentially) when a **QUIT_REQUESTED** message is received. It returns the identifier for the main thread (not that it's of much use once the application has stopped running).

This function overrides BLooper's **Run()** function. Unlike that function, it doesn't spawn a thread for the message loop or return immediately.

See also: the "Overview" to this class above, **BLooper::Run()**, **ReadyToRun()**, **QuitRequested()**

RunFilePanel(), CloseFilePanel(), IsFilePanelRunning()

long **RunFilePanel(**const char **windowTitle* = NULL,
const char **openButtonLabel* = NULL,
const char **cancelButtonLabel* = NULL,
bool *directoriesOnly* = FALSE,
BMessage **message* = NULL**)**

void **CloseFilePanel(**void**)**

bool **IsFilePanelRunning(**void**)**

RunFilePanel() requests the Browser to display a window that lets the user navigate the file system to find desired files and directories. Its arguments are all optional and are used to configure the panel:

- If another *windowTitle* is not specified, the title of the window will be "Open" preceded by the name of the application. For example:

  ```
  WishMaker : Open
  ```

  This title reflects the fact that the panel is typically used to find files the application should open and display to the user.

- If an *openButtonLabel* isn't provided, the principal button in the panel (the default button) will be labeled "Open".

- If a *cancelButtonLabel* isn't provided, the other button in the panel will be labeled "Cancel".

- If the *directoriesOnly* flag is **TRUE**, the user will be able to select only directories, not files. If the flag is **FALSE**, as it is by default, the user won't be able to select directories. Instead, their contents will be displayed in the panel as the user navigates the file system.

- If a *message* is passed, it can contain entries that further configure the panel. It also serves as a model for the message the file panel will send to the application to report which files and directories the user selected. If a *message* isn't provided, this information will be reported in a standard **B_REFS_RECEIVED** message.

If the *message* has any of the following entries, they will be used to help set up the panel:

Data name	Type code	Description
"directory"	**B_REF_TYPE**	The **record_ref** for the directory that the panel should display when it first comes on-screen. If this entry is absent, the panel will initially display the current directory of the current volume.
"frame"	**B_RECT_TYPE**	A BRect that sets the size and position of the panel in screen coordinates. If this

		entry is absent, the Browser will choose an appropriate frame rectangle for the panel.
"filter"	**B_STRING_TYPE**	An array of labels for items that should be displayed in a Filters pop-up menu. The items will be listed in the menu in the same order that they're added to the array. If this item is absent, the file panel won't display a Filters list.
"marked"	**B_STRING_TYPE**	The label that should be marked in the Filters menu. If this item is absent, the first item in the list will be marked.

If the panel is to have a Filters menu, the *message* should have one additional entry for each label in the "filter" array. This entry should list the file types associated with the label and have the label as its name. For example:

```
BMessage *model = new BMessage(OPEN_THESE);

model->AddString("filter", "All files");
model->AddString("filter", "Picture files only");
model->AddString("filter", "Text files only");
model->AddString("filter", "Picture & text files");

model->AddLong("All files", 0);

model->AddLong("Picture files only", MY_IMAGE_A_FILE_TYPE);
model->AddLong("Picture files only", MY_IMAGE_B_FILE_TYPE);

model->AddLong("Text files only", MY_TEXT_FILE_TYPE);

model->AddLong("Picture & text files", MY_IMAGE_A_FILE_TYPE);
model->AddLong("Picture & text files", MY_IMAGE_B_FILE_TYPE);
model->AddLong("Picture & text files", MY_TEXT_FILE_TYPE);

be_app->RunFilePanel(NULL, NULL, FALSE, model);
```

When the user selects a particular filter item, the file panel eliminates files of other types from the display. It shows only files with types associated with the selected item (and directories).

If an item is associated with a file type of 0—as is "All files" in the example above—it won't restrict the display. When the item is selected, the file panel shows every file in the directory. Generally, "All files" should be the first item in the menu and the one that's initially marked.

When the user operates the "Open" (or *openButtonLabel*) button, the file panel sends a message to the BApplication object. If a customized *message* is provided, it's used as the model for the message that's sent. If a *message* isn't provided, a standard **B_REFS_RECEIVED** message is sent instead. It has one data entry:

Data name	Type code	Description
"refs"	**B_REF_TYPE**	References to the database records for the files or directories selected by the user.

If the user selects more than one file or directory, there will be more than one **record_ref** item in the "refs" array.

A customized *message* works much like the model messages assigned to BControl objects and BMenuItems in the Interface Kit. The file panel makes a copy of the model, adds a "refs" entry (as described above) to the copy, and delivers the copy to the BApplication object. All other entries, including those used to configure the panel, remain unchanged. The message can have any command constant you choose, including **B_REFS_RECEIVED**.

The file panel automatically disappears when the user operates the "Open" (or *openButtonLabel*) button—provided that the message has **B_REFS_RECEIVED** as the command constant. If it has a customized constant, it remains on-screen until **CloseFilePanel()** is called (or until the application quits). You can choose to close the panel if the user makes a valid selection, or you can leave it on-screen so the user can continue making selections. **IsFilePanelRunning()** will report whether the file panel is currently displayed on-screen.

The user can close the file panel by operating the "Cancel" (or *cancelButtonLabel*) button. Whenever the panel is closed, by the user or the application, a **B_PANEL_CLOSED** message is sent to the application and the **FilePanelClosed()** hook function is called.

RunFilePanel() returns **B_NO_ERROR** if it succeeds in getting the Browser to put the file panel on-screen. If the Browser isn't running or the file panel is already on-screen, it returns **B_ERROR**. If the Browser is running but the application can't communicate with it, it returns an error code that indicates what went wrong; these codes are the same as those documented for BMessenger's **Error()** function.

See also: **RefsReceived()**, **FilePanelClosed()**

SetMainMenu(), MainMenu()

void **SetMainMenu(**BPopUpMenu **menu***)**

BPopUpMenu ***MainMenu(**void**)**

These functions set and return the application's main menu, the menu that's accessible through the icon that the Browser displays at the left top corner of the screen while the application is the current active application. Because it isn't under the control of a BMenuBar, this menu must be a kind of BPopUpMenu (but one that doesn't operate in radio mode or mark the selected item).

The main menu contains items that affect the application as a whole, rather than ones that affect operations within a particular window. The first item in the menu should be labeled

“About” plus the name of the application and the three dots of an ellipsis. The last item should be “Quit”. A default main menu with just these two items is provided for every application. You can set up your own menu in the following manner:

```
BMenuItem *item;
BPopUpMenu *menu = new BPopUpMenu("", FALSE, FALSE);

item = new BMenuItem("About <application name>...",
                         new BMessage(B_ABOUT_REQUESTED));
item->SetTarget(be_app);
menu->AddItem(item);

item = new BMenuItem("Preferences",
                         new BMessage(SET_PREFERENCES));
item->SetTarget(be_app);
menu->AddItem(item);

item = new BMenuItem("Open", new BMessage(SHOW_FILE_PANEL));
item->SetTarget(be_app);
menu->AddItem(item);

item = new BMenuItem("Quit", new BMessage(B_QUIT_REQUESTED));
item->SetTarget(be_app);
menu->AddItem(item);

be_app->SetMainMenu(menu);
```

B_ABOUT_REQUESTED and **B_QUIT_REQUESTED** are system messages that are dispatched by calling the **AboutRequested()** and **QuitRequested()** hook functions. The other messages in this example would be dispatched by calling **MessageReceived()**.

See also: **AboutRequested()**, **QuitRequested()**

SetCursor()

void **SetCursor(**const void **cursor***)**

Sets the cursor image to the bitmap specified in *cursor*. Each application has control over its own cursor, and can set and reset it as often as necessary. The cursor on-screen will have the shape specified in *cursor* as long as the application remains the active application. If it loses that status and then regains it again, its current cursor is automatically restored.

The first four bytes of *cursor* data is a preamble that gives information about the image, as follows:

- The first byte sets the size of the cursor image. The cursor bitmap is a square and this byte states the number of pixels on one side. Currently, only 16-pixel-by-16-pixel images are acceptable.
- The second byte specifies the depth of the cursor image, in bits per pixel. Currently, only monochrome one-bit-per-pixel images are acceptable.

- The third and fourth bytes set the *hot spot*, the pixel within the cursor image that's used to report the cursor's location. For example, if the cursor is located over a button on-screen so that the hot spot is within the button rectangle, the cursor is said to point to the button. However, if the hot spot lies outside the button rectangle, even if most of the cursor image is within the rectangle, the cursor doesn't point to the button.

  To locate the hot spot, assume that the pixel in the upper left corner of the cursor image is at (0, 0). Identify the vertical *y* coordinate first, then the horizontal *x* coordinate. For example, a hot spot 5 pixels to the right of the upper left corner and 8 pixels down—at (5, 8)—would be specified as "8, 5."

Image data follows these four bytes. Pixel values are specified from left to right in rows starting at the top of the image and working downward. First comes data specifying the color value of each pixel in the image. In a one-bit-per-pixel image, 1 means black and 0 means white.

Following the color data is a mask that indicates which pixels in the image square are transparent and which are opaque. Transparent pixels are marked 0; they let whatever is underneath that part of the cursor bitmap show through. Opaque pixels are marked 1.

The Application Kit defines two standard cursor images. Each is represented by a constant that you can pass to **SetCursor()**:

B_HAND_CURSOR	The hand image that's seen when the computer is first turned on. This is the default cursor.
B_I_BEAM_CURSOR	The standard I-beam image for selecting text.

See also: **HideCursor()**

SetPulseRate()

void **SetPulseRate(**double *microseconds***)**

Sets how often **Pulse()** is called (how often **B_PULSE** messages are posted). The interval set should be a multiple of 100,000.0 microseconds (0.1 second); differences less than 100,000.0 microseconds will not be noticeable. A finer granularity can't be guaranteed.

The default pulse rate is 0.0, which disables the pulsing mechanism. Setting a different rate enables it.

See also: **Pulse()**

ShowCursor() *see* HideCursor()

VolumeMounted(), VolumeUnmounted()

virtual void **VolumeMounted(**long *volume***)**

virtual void **VolumeUnmounted(**long *volume***)**

Implemented by derived classes to take action when a *volume* (typically a floppy disk) is mounted or unmounted. The volume is mounted just before **VolumeMounted()** is called and unmounted just after **VolumeUnmounted()** returns.

The *volume* identifier can be passed to the BVolume constructor to get an object corresponding to the volume.

Currently, there's no way to prevent a volume from being mounted or unmounted.

See also: the BVolume class in the Storage Kit

WindowAt()

BWindow ***WindowAt(**long *index***)** const

Returns the BWindow object recorded in the list of the application's windows at *index*, or **NULL** if *index* is out-of-range. Indices begin at 0, and there are no gaps in the list. Windows aren't listed in any particular order (such as the order they appear on-screen), so the value of *index* has no ulterior meaning. The window list excludes the private windows used by BBitmaps and other objects, but it doesn't distinguish main windows that display documents from palettes, panels, and other supporting windows.

This function can be used to iterate through the window list:

```
BWindow *window;
long i = 0;

while ( window = be_app->WindowAt(i++) ) {
    if ( window->Lock() ) {
        . . .
        window->Unlock();
    }
}
```

This works as long as windows aren't being created or deleted while the list *index* is being incremented. Locking the BApplication object doesn't lock the window list.

It's best for an application to maintain its own window list, one that arranges windows in a logical order, keeps track of any contingencies among them, and can be locked while it's being read.

See also: **CountWindows()**

BClipboard

Derived from:	*none*
Declared in:	<app/Clipboard.h>

Overview

The clipboard is a single, system-wide, temporary repository of data. In its normal use, the clipboard is a vehicle for transferring data between applications, or between different parts of the same application. An application adds some amount of data to the clipboard, then some other application (or the same application) retrieves (or “finds”) that data. This mechanism permits, most notably, the ability to cut, copy, and paste data items. For example, the BTextView object, in the Interface Kit, uses the clipboard to perform just such operations on text.

The BClipboard class represents the clipboard. As there is but a single clipboard per system, the BClipboard class allows only one BClipboard object. You don’t create this object directly in your application; it’s created automatically when you boot the machine (so there’s no public constructor or destructor for the class). Each application knows this object as **be_clipboard**. The **be_clipboard** variable in your application points (ultimately) to the same object as does every other **be_clipboard** in all other applications.

Using the Clipboard

The central BClipboard functions are these:

- **AddData()** lets you add a new item of data to the clipboard. The data that’s added is copied from an argument passed to the function. Each clipboard item is identified (primarily) by its data type (which is represented by one of the standard type constants, such as **B_ASCII_TYPE** or **B_REF_TYPE**, that are defined in **app/AppDefs.h**).

- **FindData()** retrieves data from the clipboard by providing the caller with a pointer to a specific item. This pointer points to data that resides on the clipboard—the function doesn’t copy the data.

You *must* bracket calls to **AddData()** and **FindData()** with calls to **Lock()** and **Unlock()**. This prevents other applications from accessing the clipboard while your application is using it. Conversely, if some other application (or if another thread in your application) holds the lock to the clipboard when you call **Lock()**, your application (or thread) will hang until the current lock holder calls **Unlock()**—in other words, **Lock()** will always succeed, even if it has to wait forever to do so. Currently, unfortunately, there’s no way to tell if the clipboard is already locked, nor can you specify a time limit beyond which you won't wait for the lock.

AddData() calls should also be bracketed by calls to **Clear()** and **Commit()** (see the example below for the calling sequence). Clearing the clipboard removes all data that it currently holds. This may seem harsh, but somebody has to keep the clipboard tidy. The **Commit()** function tells the clipboard that you're serious about the item-additions that you requested in the previous **AddData()** calls. If you don't commit your additions, they'll be lost.

The **Lock()**/**Unlock()** and **Clear()**/**Commit()** calls can bracket groups of **AddData()** and **FindData()** calls. The following code fragments demonstrate the expected sequences of function calls with regard to adding and retrieving clipboard data (the arguments to **FindData()** and **AddData()** aren't fully shown in the examples; see the function descriptions, below, for argument details).

Example 1: Adding Data to the Clipboard

```
/* Lock the clipboard. */
be_clipboard->Lock();

/* Clear the clipboard. */
be_clipboard->Clear();

/* Add some items. */
be_clipboard->AddData(B_DOUBLE_TYPE, . . .);
be_clipboard->AddData(B_FLOAT_TYPE, . . .);

/* Commit the additions and unlock the clipboard. */
be_clipboard->Commit();
be_clipboard->Unlock();
```

Example 2: Retrieving Data from the Clipboard

```
/* Lock the clipboard. */
be_clipboard->Lock();

/* Find a bool. */
bool *bp = (bool *)be_clipboard->FindData(B_BOOL_TYPE, . . .);

/* Copy the bool value (for reasons that are explained in the
 * FindData() description).
 */
bool yesOrNo = *bp;

/* Unlock the clipboard */
be_clipboard->Unlock();
```

It's possible to mix **AddData()** and **FindData()** calls within the same "session," but such a pursuit doesn't correspond to traditional manipulations on selected data.

Member Functions

AddData(), AddText()

void **AddData(**ulong *type*, const void **data*, long *numBytes***)**

void **AddText(**const char **string***)**

These functions add a buffer of data to the clipboard. The **AddData()** function copies *numBytes* bytes of data starting at *data*. The clipboard thinks this data is of the type given by the *type* argument (one of the data type constants—**B_BOOL_TYPE**, **B_DOUBLE_TYPE**, **B_FLOAT_TYPE**, and so on—declared in **AppDefs.h**).

AddText() is a convenience function that adds a copy of *string* to the clipboard. Text items are declared to be **B_ASCII_TYPE**.

You *must* call **Lock()** before calling **AddData()** or **AddText()**. If you don't, your application will visit the debugger. Furthermore, you must call **Unlock()** after you've added your items. Multiple invocations of **AddData()** or **AddText()** (or both) can be performed within the same **Lock()**/**Unlock()** pair. You can add any number of items of the same or different types while you have the clipboard locked.

By convention, you should call **Clear()** immediately before calling **AddData()** or **AddText()** (but after calling **Lock()**). This will remove all items that the clipboard is currently holding.

After you've added your items to the clipboard (but before you call **Unlock()**), you must commit the additions by calling **Commit()**. If you don't commit before you unlock, your additions won't be recorded.

The **FindData()** and **FindText()** functions retrieve data that's been added through **AddData()** and **AddText()** calls.

Clear()

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

Erases all items that are currently on the clipboard. Normally, you call **Clear()** just before you add new data to the clipboard (through invocations of **AddData()** and **AddText()**). You must call **Lock()** before calling **Clear()**; if you don't, the debugger will tap you on the shoulder.

Commit()

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

Forces the clipboard to notice the items you added. All calls (or sequence of calls) to **AddData()** or **AddText()** must be followed by a call to **Commit()**, or you'll lose the additions. The call to **Commit()** must precede the call to **Unlock()** that balances the call to **Lock()** that preceded the call to **Clear()** that worried the cat that killed the rat that ate the malt . . .

CountEntries()

long **CountEntries(**ulong *type***)**

Returns the number of items on the clipboard that are of the specified type. The *type* argument must be one of the data type constants defined in **app/AppDefs.h**. If *type* is **B_ANY_TYPE**, the function returns the total number of current clipboard items.

You must call **Lock()** before invoking this function; if you don't, it returns **NULL**.

DataOwner()

BMessenger **DataOwner(**void**)**

Returns a BMessenger object for the application that last committed data to the clipboard. The BMessenger targets that application's BApplication object.

FindData(), FindText()

void ***FindData(**ulong *type*, long **numBytes***)**
void ***FindData(**ulong *type*, long *index*, long **numBytes***)**

const char ***FindText(**long **numBytes***)**

These functions return a pointer to a particular item on the clipboard.

FindData() returns an item of the requested *type*, which can be any of the data type constants defined in **AppDefs.h** or an application-defined type code. If an *index* is provided, it returns the item at that index; indices begin at 0 and count only items of the specified type. If an index isn't supplied, **FindData()** finds the first item on the clipboard matching the requested type.

FindText() always searches for the first item of type **B_ASCII_TYPE**.

If the item is found, a pointer to it is returned directly by the function, and the number of bytes of data that comprise the item is returned by reference in *numBytes*. Keep in mind that this pointer points to data that lies on the clipboard; if you want a permanent copy of the data, you must copy the data that the pointer points to before you unlock the clipboard (as shown in the example in the section "Using the Clipboard" on page 43).

An individual call or sequence of calls to **FindData()** and **FindText()** must be bracketed by invocations of **Lock()** and **Unlock()**.

If the function can't find the specified item—for example, if the clipboard doesn't have data of the requested *type* or the *index* passed to **FindData()** is out-of-range—it returns a **NULL** pointer and, perhaps more telling, sets *numBytes* to 0. If you don't lock the clipboard before invoking either **FindData()** or **FindText()**, you'll find the debugger.

Lock(), Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

These functions lock and unlock the clipboard. Locking the clipboard gives your application exclusive permission to invoke the other BClipboard functions. (More accurately, the permission extends only to the very thread in which **Lock()** is called.) If some other thread already has the clipboard locked when your thread calls **Lock()**, your thread will wait until the lock-holding thread calls **Unlock()**. Your thread should also invoke **Unlock()** when you're done manipulating the clipboard.

Lock() should invariably be successful and return **TRUE**.

See also: **BLooper::Lock()**

BHandler

Derived from:	public BObject
Declared in:	<app/Handler.h>

Overview

BHandlers are the objects that respond to messages received in message loops. The class declares a hook function—**MessageReceived()**—that derived classes must implement to handle expected messages. BLooper's **DispatchMessage()** function calls **MessageReceived()** to pass incoming messages from the BLooper to the BHandler.

All messages are passed to BHandler objects—even system messages, which are passed by calling a message-specific function, not **MessageReceived()**. These specific functions are declared in classes derived from BHandler—especially BWindow and BView in the Interface Kit and BLooper and BApplication in this Kit. For example, the BApplication class declares a **ReadyToRun()** function to respond to **B_READY_TO_RUN** messages, and the BView class declares a **KeyDown()** function to respond to **B_KEY_DOWN** messages. (BHandler itself declares the function that responds to **B_HANDLERS_REQUESTED** system messages, **HandlersRequested()**.)

All messages that aren't matched to a specific hook function—messages defined by applications rather than the kits—are dispatched by calling **MessageReceived()**.

BHandlers can be chained together in a linked list. The default behavior for **MessageReceived()** is simply to pass the message to the next handler in the chain. However, system messages are not passed from handler to handler.

To be eligible to get messages from a BLooper, a BHandler must be in the BLooper's circle of handlers. At any given time, a BHandler can belong to only one BLooper.

A target BHandler can be designated for a message when calling BLooper's **PostMessage()** function to post it. Messages that a BMessenger object sends are targeted to the BHandler that was named when constructing the BMessenger. Messages that a user drags and drops are targeted to the object (a BView) that controls the part of the window where the message was dropped. The messaging mechanism eventually passes the target BHandler to **DispatchMessage()**, so that the message can be delivered to its designated destination.

Hook Functions

HandlersRequested()	Can be implemented to supply BMessengers for other BHandler objects associated with this BHandler.
MessageReceived()	Implemented to handle received messages.

Constructor and Destructor

BHandler()

BHandler(const char **name* = NULL**)**

Initializes the BHandler by assigning it a *name* and registering it with the messaging system.

~BHandler()

virtual **~BHandler(**void**)**

Removes the BHandler's registration and frees the memory allocated for its name.

Member Functions

AddFilter() *see* SetFilterList()

FilterList() *see* SetFilterList()

HandlersRequested()

virtual void **HandlersRequested(**BMessage **message***)**

Implemented by derived classes to send a **B_HANDLERS_INFO** message in reply to the received **B_HANDLERS_REQUESTED** *message* passed as an argument. The request is for BMessenger objects corresponding to BHandler objects in the application; the BMessengers will permit the requester to direct messages to those BHandlers. This function should place the BMessengers in a "handlers" entry in the reply message—or, failing that, to place an error code in an entry named "error".

Since, by default, BHandlers are not associated with other BHandlers, this base version of the function doesn't supply any BMessengers; it simply puts the **B_ERROR** constant in an "error" entry and sends the reply.

For more information on the protocols that the kits currently use for **B_HANDLERS_INFO** and **B_HANDLERS_REQUESTED** messages, see the versions of this function defined in derived classes.

See also: **BLooper::HandlersRequested()**, **BApplication::HandlersRequested()**, **BWindow::HandlersRequested()**, **BView::HandlersRequested()**, "**B_HANDLERS_REQUESTED**" on page 4 in the *Message Protocols* appendix

Looper()

virtual BLooper ***Looper(**void**)** const

Returns the BLooper object that the BHandler is associated with, or **NULL** if it's not associated with any BLooper. A BHandler must be associated with a BLooper before the BLooper can call upon it to handle any messages it dispatches. (However, strictly speaking, this restriction is imposed when the message is posted or when the BMessenger that will send it is constructed, rather than when it's dispatched.)

BLooper objects are automatically associated with themselves; they can act as handlers only for messages that they receive in their own message loops. All other BHandlers must be explicitly tied to a particular BLooper by calling that BLooper's **AddHandler()** function. A BHandler can be associated with only one BLooper at a time.

In the Interface Kit, when a BView is added to a window's view hierarchy, it's also added as a BHandler to the BWindow object.

See also: **BLooper::AddHandler()**, **BLooper::PostMessage()**, the BMessenger constructor

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Implemented by derived classes to respond to messages that are dispatched to the BHandler. The default (BHandler) implementation of this function doesn't respond to any messages; it simply calls the next handler's version of **MessageReceived()** to pass it the *message*.

You must implement **MessageReceived()** to handle the variety of messages that might be dispatched to the BHandler. It can distinguish between messages by the value recorded in the **what** data member of the BMessage object. For example:

```
void MyHandler::MessageReceived(BMessage *message)
{
    switch ( message->what ) {
    case COMMAND_ONE:
        . . .
        break;
    case COMMAND_TWO:
        . . .
        break;
        case COMMAND_THREE:
            . . .
            break;
        default:
            inherited::MessageReceived(message);
            break;
        . . .
        }
    }
```

When defining a version of **MessageReceived()**, it's always a good idea to incorporate the inherited version as well, as shown in the example above. This ensures, first, that any messages handled by base versions of the function are not overlooked and, second, that the message is passed to the BHandler's next handler if even the inherited functions don't recognize it.

If the message comes to the end of the line—if it's not recognized and there is no next handler—the BHandler version of this function sends a **B_MESSAGE_NOT_UNDERSTOOD** reply to notify the message source.

See also: **SetNextHandler()**, **BLooper::PostMessage()**, **BLooper::DispatchMessage()**

NextHandler() *see* SetNextHandler()

SetFilterList(), FilterList(), AddFilter(), RemoveFilter()

virtual void **SetFilterList(**BList **list***)**

BList ***FilterList(**void**)** const

virtual void **AddFilter(**BMessageFilter **filter***)**

virtual bool **RemoveFilter(**BMessageFilter **filter***)**

These functions manage a list of BMessageFilter objects associated with the BHandler.

SetFilterList() assigns the BHandler a new *list*, replacing any list previously assigned. The list must contain pointers to instances of the BMessageFilter class or, more usefully, to instances of classes that derive from BMessageFilter. If *list* is **NULL**, the current list is removed. **FilterList()** returns the current list of filters.

AddFilter() adds a *filter* to the end of the BHandler's list of filters. It creates the BList object if it doesn't already exist. By default, BHandlers don't maintain a BList of filters until one is assigned or the first BMessageFilter is added. **RemoveFilter()** removes a *filter* from the list. It returns **TRUE** if successful, and **FALSE** if it can't find the specified filter in the list (or the list doesn't exist). It leaves the BList in place even after removing the last filter.

For **SetFilterList()**, **AddFilter()**, and **RemoveFilter()** to work, the BHandler must be assigned to a BLooper object and the BLooper must be locked.

See also: **BLooper::SetCommonFilterList()**, **BLooper::Lock()**, the BMessageFilter class

SetName(), Name()

void **SetName(**const char **string***)**

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

These functions set and return the name that identifies the BHandler. The name is originally set by the constructor. **SetName()** assigns the BHandler a new name, and **Name()** returns the current name. The string returned by **Name()** belongs to the BHandler object; it shouldn't be altered or freed.

See also: the BHandler constructor, **BView::FindView()** in the Interface Kit

SetNextHandler(), NextHandler()

void **SetNextHandler(**BHandler **handler***)**

BHandler ***NextHandler(**void**)** const

These functions set and return the BHandler object that's linked to this BHandler. By default, the **MessageReceived()** function passes any messages that a BHandler can't understand to its next handler.

When a BHandler object is added to a BLooper, the BLooper becomes its next handler by default. The default next handler for a BLooper is the BApplication object; the next handler for the BApplication object is **NULL**. The handler chain for an ordinary BHandler object is therefore BHandler to BLooper to BApplication object.

However, when a BView object is added to a window, the Interface Kit assigns the BView's parent as its next handler (unless the parent is the window's top view, in which case the BWindow object is assigned as the next handler). The handler chain for BViews is therefore BView to BView, up the view hierarchy, to the BWindow to the BApplication object.

SetNextHandler() can alter any of these default assignments. For it to work, the BHandler must be assigned to a BLooper object and the BLooper must be locked.

See also: **MessageReceived()**

BLooper

Derived from:	public BHandler
Declared in:	<app/Looper.h>

Overview

A BLooper object runs a message loop in a thread that it spawns for that purpose. It offers applications a simple way of creating a thread with a message interface.

Various classes in the Be software kits derive from BLooper in order to associate threads with significant entities in the application and to set up message loops with special handling for system messages. In the Application Kit, the BApplication object runs a message loop in the application's main thread. (Unlike other BLoopers, the BApplication object doesn't spawn a separate thread, but takes over the thread in which the application was launched.) In the Interface Kit, each BWindow object runs a loop to handle messages that report activity in the user interface.

Running the Loop

Constructing a BLooper object gets it ready to work, but doesn't actually begin the message loop. Its **Run()** function must be called to spawn the thread and initiate the loop. Some derived classes may choose to call **Run()** within the class constructor,

```
MyLooper::MyLooper(const char *name, long priority)
        : BLooper(name, priority)
{
    . . .
    Run();
}
```

so that simply constructing the object yields a fully functioning message loop. Other classes may need to keep object initialization separate from loop initiation. (The BWindow class in the Interface Kit is an example of the former approach, BApplication of the latter.)

Receiving and Dispatching Messages

Messages are posted to the BLooper's thread by calling its **PostMessage()** function. This simply puts the message in a queue. Messages can also be delivered to the BLooper's queue—somewhat more indirectly—by a BMessenger object or by the **SendReply()** function of a BMessage object.

No matter how they get there, the thread takes messages from the queue one at a time, in the order that they arrive, and calls **DispatchMessage()** for each one. **DispatchMessage()** hands the message to a BHandler object; the BHandler kicks off the thread's specific response to the message.

Posting or sending a message to a thread initiates activity within that thread, beginning with the **DispatchMessage()** function. Since **DispatchMessage()** immediately transfers responsibility for incoming messages to BHandler objects, BHandlers determine what happens in the BLooper's thread. Everything that the thread does, it does through BHandlers responding to messages. The BLooper merely runs the posting and dispatching mechanism.

The BLooper object is locked when **DispatchMessage()** is called; it stays locked until the thread has finished responding to the message.

Acting as the Handler

When a message is posted to a thread, a target BHandler can be named for it. Messages that aren't posted to a specific target are handled by the BLooper itself—in other words, the BLooper acts as the default handler. (The BLooper class derives from BHandler for just this reason.)

Thus, a BLooper object can play both roles—the BLooper role of running the message loop and the BHandler role of responding to messages. For it to act as a handler, you must derive a class from BLooper and define a **MessageReceived()** function that can respond to the messages dispatched to it.

However, the BLooper class can also be used without change, as it's defined in the Kit—as long as all messages are targeted to a another handler.

Eligible Handlers

A BLooper keeps a list of the BHandler objects that are eligible for the messages it dispatches. **AddHandler()** places a BHandler in the list, and **RemoveHandler()** removes it. A BHandler can be associated with only one BLooper at a time. (The BLooper is an automatic member of the list; it cannot be removed and associated with another BLooper.)

A BHandler's **Looper()** function will reveal which BLooper it currently belongs to. The BLooper itself doesn't reveal the membership of its list.

A BHandler can't get messages dispatched by any BLooper except the one it's associated with. However, this eligibility constraint is imposed not by **DispatchMessage()**, but by the BMessenger constructor when a target BHandler is proposed for the messages it will send and by **PostMessage()** when a BHandler is named as the target of a message posted to the BLooper.

Hook Functions

DispatchMessage()	Passes incoming messages to a BHandler; can be overridden to change the way certain messages or classes of messages are dispatched.
QuitRequested()	Can be implemented to decide whether a request to terminate the message loop and destroy the BLooper should be honored or not.

Constructor and Destructor

BLooper()

BLooper(const char **name* = NULL, long *priority* = B_NORMAL_PRIORITY**)**

Assigns the BLooper object a *name* and sets up its message queue, but doesn't spawn a thread or begin the message loop. Call **Run()** to spawn the thread that the BLooper will oversee. **Run()** creates the thread at the specified *priority* level and initiates its message loop.

The *priority* determines how much attention the thread will receive from the scheduler, and consequently how much CPU time it will get relative to other threads. You must choose one of the discrete priority levels defined in **kernel/OS.h**; intermediate priorities are not possible. The defined priorities, from lowest to highest, are:

B_LOW_PRIORITY	For threads running in the background that shouldn't interrupt other threads.
B_NORMAL_PRIORITY	For all ordinary threads, including the main thread.
B_DISPLAY_PRIORITY	For threads associated with objects in the user interface, including window threads.
B_URGENT_DISPLAY_PRIORITY	For interface threads that deserve more attention than ordinary windows.
B_REAL_TIME_DISPLAY_PRIORITY	For threads that animate the on-screen display.

B_URGENT_PRIORITY	For threads performing time-critical computations.
B_REAL_TIME_PRIORITY	For threads that control real-time processes that need unfettered access to the CPUs.

Some derived classes may want to call **Run()** in the constructor, so that the object is set in motion at the time it's created. This is what the BWindow class in the Interface Kit does. Other derived classes might want to keep a separation between constructing the object and running it. The BApplication class maintains this distinction.

BLooper objects should always be dynamically allocated (with **new**), never statically allocated on the stack.

See also: **Run()**, **BHandler::SetName()**

~BLooper()

virtual **~BLooper(**void**)**

Frees the message queue and all pending messages, stops the message loop, and destroys the thread in which it ran. BHandlers that have been added to the BLooper are not deleted.

With the exception of the BApplication object, BLoopers should be destroyed by calling the **Quit()** function (or **QuitRequested()**), not by using the **delete** operator.

See also: **Quit()**

Member Functions

AddCommonFilter() *see* SetCommonFilterList()

AddHandler(), RemoveHandler()

virtual void **AddHandler(**BHandler **handler***)**

virtual bool **RemoveHandler(**BHandler **handler***)**

AddHandler() adds *handler* to the BLooper's list of BHandler objects, and **RemoveHandler()** removes it. Only BHandlers that have been added to the list are eligible to respond to the messages the BLooper dispatches. (However, this constraint is imposed not by **DispatchMessage()**, but by **PostMessage()** and the BMessenger constructor.) A BHandler can belong to no more than one BLooper, but can change its affiliation from time to time.

AddHandler() also calls the *handler*'s **SetNextHandler()** function to assign it the BLooper as its default next handler. **RemoveHandler()** calls the same function to set the *handler*'s next handler to **NULL**.

AddHandler() fails if the *handler* already belongs to a BLooper. **RemoveHandler()** returns **TRUE** if it succeeds in removing the BHandler from the BLooper, and **FALSE** if not or if the *handler* doesn't belong to the BLooper in the first place. For either function to work, the BLooper must be locked.

See also: **BHandler::Looper()**, **BHandler::SetNextHandler()**, **PostMessage()**, the BMessenger class

CommonFilterList() *see* SetCommonFilterList()

CurrentMessage(), DetachCurrentMessage()

BMessage ***CurrentMessage(**void**)** const

BMessage ***DetachCurrentMessage(**void**)**

Both these functions return a pointer to the message that the BLooper's thread is currently processing, or **NULL** if it's currently between messages. That's all that **CurrentMessage()** does. **DetachCurrentMessage()** also detaches the message from the message loop, so that:

- It will no longer be the current message. The current message will be **NULL** until the thread gets another message from the queue.
- The thread won't automatically delete the message when the message cycle ends and it's ready to get the next message. It becomes the caller's responsibility to delete the message later (or to post it once more so that it will again be subject to automatic deletion).

Since the message won't be deleted automatically, you have time to reply to it later. However, if the thread that initiated the message is waiting for a reply, you should send one (or get rid of the BMessage) without much delay. If a reply hasn't already been sent by the time the message is deleted, the BMessage destructor sends back a default **B_NO_REPLY** message to indicate that a real reply won't be forthcoming. But if the message isn't deleted and a reply isn't sent, the initiating thread will continue to wait. (BMessage's **IsSourceWaiting()** function will let you know whether the message source is waiting for a reply.)

Detaching a message is useful only when you want to stretch out the response to it beyond the end of the message cycle, perhaps passing responsibility for it to another thread while the BLooper's thread continues to get and respond to other messages.

Since the current message is passed as an argument to BLooper's **DispatchMessage()** and BHandler's **MessageReceived()** hook functions, you may never need to call **CurrentMessage()** to get hold of it.

However, classes derived from BLooper (BApplication and BWindow, in particular) dispatch system messages by calling a message-specific function, not **MessageReceived()**. Typically, these functions are passed only part of the information contained in the BMessage. In such a case, you will have to call **CurrentMessage()** to get complete information about the instruction or event the BMessage object reports.

For example, in the Interface Kit, a **KeyDown()** function might check whether the Control key was pressed at the time of the key-down event as follows:

```
void MyView::KeyDown(ulong key)
{
    BMessage *message = Window()->CurrentMessage();
    if ( message->FindLong("modifiers") & B_CONTROL_KEY ) {
        . . .
    }
    . . .
}
```

See also: **BHandler::MessageReceived()**, **BMessage::WasSent()**

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BHandler **target***)**

Dispatches messages as they're received by the BLooper's thread. Precisely how they're dispatched depends on the *message* and the designated *target* BHandler. The BWindow and BApplication classes that derive from BLooper implement their own versions of this function to provide for special dispatching for system messages. Each class defines its own set of such messages.

The *target* may be the BHandler object that was named when the *message* was posted, the BHandler that was passed when the BMessenger was constructed, the handler that was designated as the target for a reply message, or (for a BWindow) the BView where the *message* was dropped. Or it might be the BLooper itself, acting in its capacity as the default handler. For system messages it may be **NULL**; if so, the dispatcher must figure out a target for the message based on the contents of the BMessage object.

DispatchMessage() is the first stop in the message-handling mechanism. The BLooper's thread calls it automatically as it reads messages from the queue—you never call it yourself.

BLooper's version of **DispatchMessage()** dispatches **B_QUIT_REQUESTED** messages that are targeted to the BLooper itself by calling its own **QuitRequested()** function. It dispatches **B_HANDLERS_REQUESTED** messages by calling the *target*'s **HandlersRequested()** function. All other messages are forwarded to the *target*'s **MessageReceived()** function.

You can override this function to dispatch the messages that your own application defines or recognizes. Of course, you can also just wait for these messages to fall through to **MessageReceived()**—the choice is yours. If you do override **DispatchMessage()**, you should:

- Call the base class version of the function *after* you've handled your own messages,
- Exclude all messages that you've handled yourself from the base version call, and
- Lock the BLooper while the message is being handled.

For example:

```
void MyLooper::DispatchMessage(BMessage *msg, BHandler *target)
{
    switch ( msg->what ) {
    case MY_MESSAGE1:
        . . .
        break;
    case MY_MESSAGE2:
        . . .
        break;
    default:
        inherited::DispatchMessage(msg, target);
        break;
    }
}
```

Don't delete the messages you handle when you're through with them; they're deleted for you.

The system locks the BLooper before calling **DispatchMessage()** and keeps it locked for the duration of the thread's response to the message (until **DispatchMessage()** returns).

See also: the BMessage class, **BHandler::MessageReceived()**, **QuitRequested()**

HandlersRequested()

virtual void **HandlersRequested(**BMessage **message***)**

Responds to a **B_HANDLERS_REQUESTED** *message* by sending a **B_HANDLERS_INFO** message in reply. The request is for BMessenger objects that can deliver messages targeted to BHandlers that have been added to the BLooper.

The incoming *message* may ask for a particular BHandler associated with the BLooper, or it may ask for all of them. If it has an entry named "index", the BLooper looks for the BHandler at that index in its list of eligible handlers. Otherwise, if the message has an entry labeled "name", the BLooper looks for the associated BHandler with that name. If it finds a BHandler object at the requested index or with the requested name, it places a BMessenger for that object in the **B_HANDLERS_INFO** reply under the name "handlers". However, if it can't find the requested object, it adds the **B_BAD_INDEX** or **B_NAME_NOT_FOUND** error constant to the reply message under the name "error".

If the incoming **B_HANDLERS_REQUESTED** message doesn't request a particular BHandler by index or name, the BLooper adds BMessengers for all eligible BHandlers to the "handlers" array of the reply. The array should contain at least one BMessenger, the one corresponding to the BLooper itself.

See also: **BHandler::HandlersRequested()**

IsLocked() *see* LockOwner()

Lock(), Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

These functions provide a mechanism for locking data associated with the BLooper, so that one thread can't alter the data while another thread is in the middle of doing something that depends on it. Only one thread can have the BLooper locked at any given time. **Lock()** waits until it can lock the object, then returns **TRUE**. It returns **FALSE** only if the BLooper can't be locked at all—for example, if it was destroyed by another thread.

Calls to **Lock()** and **Unlock()** can be nested. If **Lock()** is called more than once from the same thread, it will take an equal number of **Unlock()** calls from that thread to unlock the BLooper. (If **Lock()** is called from another thread, it waits until the thread that owns the lock unlocks the BLooper. It then obtains the lock and returns **TRUE**.)

Locking is the basic mechanism for operating safely in a multithreaded environment. It's especially important for the kit classes derived from BLooper—BApplication and BWindow.

However, it's generally not necessary to lock a BLooper when calling functions defined in the class itself or in a derived class. For example, BApplication and BWindow functions are implemented to call **Lock()** and **Unlock()** when necessary. Moreover, the BLooper is locked for you whenever it dispatches a message. It remains locked until the response to the message is complete.

Functions you define in classes derived from BLooper (or from BApplication and BWindow) should also call **Lock()** and **Unlock()**. In addition, you should employ the locking mechanism when calling functions of a class that's closely associated with a BLooper—for example, when calling functions of a BView that's attached to a BWindow.

Although locking is important and useful, you shouldn't be too blithe about it. While you hold a BLooper's lock, no other thread can acquire it. If another thread calls a function that tries to lock, the thread will hang until you unlock. Each thread should hold the lock as briefly as possible.

See also: **LockOwner()**

LockOwner(), IsLocked()

inline thread_id **LockOwner(**void**)** const

inline bool **IsLocked(**void**)** const

LockOwner() returns the thread that currently has the BLooper locked, or –1 if the BLooper isn't locked.

IsLocked() returns **TRUE** if the calling thread has the BLooper locked (if it's the lock owner) and **FALSE** if not (if some other thread is the owner or the BLooper isn't locked).

See also: **Lock()**

Looper()

virtual BLooper ***Looper(**void**)** const

Overrides the BHandler version of this function to return the BLooper object itself. This prevents the BLooper from acting as a handler for messages posted to any other thread. A BLooper can take on the role of BHandler only for messages delivered to its own thread.

See also: **BHandler::Looper()**, **PostMessage()**

MessageQueue()

BMessageQueue ***MessageQueue(**void**)** const

Returns the queue that holds messages posted or sent to the BLooper's thread. You rarely need to examine the message queue directly; it's made available so you can cheat fate by looking ahead.

See also: the BMessageQueue class

PostMessage()

long **PostMessage(**BMessage **message*, BHandler **target* = NULL**)**
long **PostMessage(**ulong *command*, BHandler **target* = NULL**)**

Places a *message* in the BLooper's message queue and arranges for it to be dispatched to the *target* BHandler. If a *target* isn't mentioned, the message will be dispatched to the BLooper. The BLooper acts as the default handler for all messages not specifically targeted to another object.

However, if the named *target* is associated with a different BLooper (if the *target*'s **Looper()** function returns **NULL** or some other BLooper object), the posting fails and the *message* is deleted. (A BHandler must be associated with a particular BLooper before it can be the target for messages posted to that object. It can't get messages from any other

BLooper except the one it belongs to. For example, BViews in the Interface Kit are restricted to receiving messages posted to the BWindows to which they're attached.)

Once posted, the BMessage object belongs to the BLooper's thread, so you should not modify it, post it again, assign it to some other object, or delete it. It will be deleted automatically after it has been received and responded to.

If a *command* is passed rather than a message, **PostMessage()** creates a BMessage object, initializes its **what** data member to *command*, and posts it. This simply saves you the step of constructing a BMessage when it won't contain any data. For example, this code

```
myWindow->PostMessage(command, target);
```

is equivalent to:

```
myWindow->PostMessage(new BMessage(command), target);
```

To post the message, the *command* version of this function calls the version that takes a full BMessage argument. Thus, if you override just the *message* version, you'll affect how both operate.

This function returns **B_NO_ERROR** if successful, **B_MISMATCHED_VALUES** if the posting fails because the proposed handler is invalid, and **B_ERROR** if it fails because the BLooper itself is invalid.

See also: **BHandler::Looper()**, **DispatchMessage()**

PreferredHandler()

virtual BHandler ***PreferredHandler(**void**)** const

Implemented by derived classes to return a preferred BHandler for messages posted to the BLooper. This function simply informs those who are about to post messages to the BLooper who they might name as the message handler. For example:

```
myLooper->PostMessage(msg, myLooper->PreferredHandler());
```

The BLooper class itself doesn't do anything with the preferred handler; it's not a default value for any BLooper operation.

In the Interface Kit, BWindow objects name the current focus view as the preferred handler. This makes it possible for other objects—such as BMenuItems and BButtons—to target messages to the BView that's currently in focus, whatever view that may happen to be at the time. For example, by posting its messages to the window's preferred handler, a "Cut" menu item can make sure that it always acts on whatever view contains the current selection. See the chapter on the Interface Kit for information on windows, views, and the role of the focus view.

The BLooper version of this function simply returns **NULL**, to indicate that generic BLoopers don't have a preferred handler. Note, however, that when a **NULL** handler is passed to **PostMessage()**, that function designates the BLooper itself as the target. For example, if **PreferredHandler()** returned **NULL** in the line of code shown above, the message would be dispatched to *myLooper* by default. Thus, in effect, a generic BLooper is its own preferred handler, even though **PreferredHandler()** returns **NULL**.

See also: **BControl::SetTarget()** and **BMenuItem::SetTarget()** in the Interface Kit, **PostMessage()**

Quit()

virtual void **Quit(**void**)**

Exits the message loop, frees the message queue, kills the thread, and deletes the BLooper object.

When called from the BLooper's thread, all this happens immediately. Any pending messages are ignored and destroyed. Because the thread dies, **Quit()** doesn't return.

However, when called from another thread, **Quit()** waits until all previously posted messages (all messages already in the queue) work their way through the message loop and are handled. It then destroys the BLooper and returns only after the loop, queue, thread, and object no longer exist.

Quit() therefore terminates the BLooper synchronously; when it returns, you know that everything has been destroyed. To quit the BLooper asynchronously, you can post a **B_QUIT_REQUESTED** message to the thread (that is, a BMessage with **B_QUIT_REQUESTED** as its **what** data member). **PostMessage()** places the message in the queue and returns immediately.

When it gets a **B_QUIT_REQUESTED** message, the BLooper calls the **QuitRequested()** virtual function. If **QuitRequested()** returns **TRUE**, as it does by default, it then calls **Quit()**.

See also: **QuitRequested()**

QuitRequested()

virtual bool **QuitRequested(**void**)**

Implemented by derived classes to determine whether the BLooper should quit when requested to do so. The BLooper calls this function to respond to **B_QUIT_REQUESTED** messages. If it returns **TRUE**, the BLooper calls **Quit()** to exit the message loop, kill the thread, and delete itself. If it returns **FALSE**, the request is denied and no further action is taken.

BLooper's default implementation of **QuitRequested()** always returns **TRUE**.

A request to quit that's delivered to the BApplication object is, in fact, a request to quit the entire application, not just one thread. BApplication therefore overrides **QuitRequested()** to pass the request on to each window thread before shutting down.

For BWindow objects in the Interface Kit, a request to quit might come from the user clicking the window's close button (a quit-requested event for the window), from the user's decision to quit the application (a quit-requested event for the application), from a "Close" menu item, or from some other occurrence that forces the window to close.

Classes derived from BWindow typically implement **QuitRequested()** to give the user a chance to save documents before the window is destroyed, or to cancel the request.

If an application can be launched more than once (**B_MULTIPLE_LAUNCH**) and its entire interface is essentially contained in one window, quitting the window might be tantamount to quitting the application. In this case, the window's **QuitRequested()** function should pass the request along to the BApplication object. For example:

```
bool MyWindow::QuitRequested()
{
    . . .
    be_app->PostMessage(B_QUIT_REQUESTED);
    return TRUE;
}
```

After asking the application to quit, **QuitRequested()** returns **TRUE** to immediately dispose of the window. If it returns **FALSE**, BApplication's version of the function will again request the window to quit.

If you call **QuitRequested()** from your own code, be sure to also provide the code that calls **Quit()**:

```
if ( myLooper->QuitRequested() )
    myLooper->Quit();
```

See also: **BApplication::QuitRequested()**, **Quit()**

RemoveCommonFilter() *see* SetCommonFilterList()

Run()

virtual thread_id **Run(**void**)**

Spawns a thread at the priority level that was specified when the BLooper was constructed and begins running a message loop in that thread. If successful, this function returns the thread identifier. If unsuccessful, it returns **B_NO_MORE_THREADS** or **B_NO_MEMORY** to indicate why.

A BLooper can be run only once. If called a second time, **Run()** returns **B_ERROR**, but doesn't disrupt the message loop already running. < Currently, it drops into the debugger so you can correct the error. >

The message loop is terminated when **Quit()** is called, or (potentially) when a **B_QUIT_REQUESTED** message is received. This also kills the thread and deletes the BLooper object.

See also: the BLooper constructor, the BApplication class, **Quit()**

SetCommonFilterList(), CommonFilterList(), AddCommonFilter(), RemoveCommonFilter()

virtual void **SetCommonFilterList(**BList **list***)**

BList ***CommonFilterList(**void**)** const

virtual void **AddCommonFilter(**BMessageFilter **filter***)**

virtual void **RemoveCommonFilter(**BMessageFilter **filter***)**

These functions manage a list of filters that can apply to any message the BLooper receives, regardless of its target BHandler. They complement a similar set of functions defined in the BHandler class. When a filter is associated with a BHandler, it applies only to messages targeted to that BHandler. When it's associated with a BLooper as a common filter, it applies to all messages that the BLooper dispatches, regardless of the target.

In addition to the list of common filters, a BLooper can maintain a filter list in its role as a BHandler. As for other BHandlers, these filters apply only if the BLooper is the target of the message.

SetCommonFilterList() assigns the BLooper a new *list* of common filters, replacing any list previously assigned. The list must contain pointers to instances of the BMessageFilter class or, more usefully, instances of classes that derive from BMessageFilter. If *list* is **NULL**, the current list is removed without a replacement. **CommonFilterList()** returns the current list of common filters.

AddCommonFilter() adds a *filter* to the end of the list of common filters. It creates the BList object if it doesn't already exist. By default, BLoopers don't keep a BList of common filters until one is assigned or **AddCommonFilter()** is called for the first time. **RemoveCommonFilter()** removes a *filter* from the list. It returns **TRUE** if successful, and **FALSE** if it can't find the specified filter in the list (or the list doesn't exist). It leaves the BList in place even after removing the last filter.

For **SetCommonFilterList()**, **AddCommonFilter()**, and **RemoveCommonFilter()** to work, the BLooper must be locked.

See also: **BHandler::SetFilterList()**, **Lock()**, the BMessageFilter class

Thread(), Team()

thread_id **Thread(**void**)** const

team_id **Team(**void**)** const

These functions identify the thread that runs the message loop and the team to which it belongs. **Thread()** returns **B_ERROR** if **Run()** hasn't yet been called to spawn the thread and begin the loop. **Team()** should always return the application's **team_id**.

Unlock() *see* Lock()

BMessage

Derived from:	public BObject
Declared in:	<app/Message.h>

Overview

A BMessage bundles information so that it can be conveyed from one application to another, one thread of execution to another, or even one object to another. Servers use BMessage objects to notify applications about events. An application can use them to communicate with other applications or to initiate activity in a different thread of the same application. In the Interface Kit, BMessages package information that the user can drag from one location on-screen and drop on another. They also hold data that's copied to the clipboard. Behind the scenes in the Storage Kit, they convey queries and hand back requested information.

A BMessage is simply a container. The class defines functions that let you put information into a message, determine what kinds of information are present in a message that's been delivered to you, and get the information out. It also has a function that let's you reply to a message once it's received. But it doesn't have functions that can make the initial delivery. For that it depends on the help of other classes in the Application Kit, particularly BLooper and BMessenger. See "Messaging" on page 6 of the chapter introduction for an overview of the messaging mechanism and how BMessage objects work with these other classes.

Message Contents

When information is added to a BMessage, it's copied into dynamically allocated memory and stored under a name. If more than one piece of information is added under the same name, the BMessage sets up an array of data for that name. The name (along with an optional index into the array) is then used to retrieve the data.

For example, this code adds a floating-point number to a BMessage under the name "pi",

```
BMessage *msg = new BMessage;
msg->AddFloat("pi", 3.1416);
```

and this code locates it:

```
float pi = msg->FindFloat("pi");
```

Names can be arbitrarily assigned. There's no limit on the number of named entries a message can contain or on the size of an entry. However, since the search is linear, combing through a very long list of names to find a particular piece of data may be inefficient. Also, because of the amount of data that must be moved, an extremely large message (over 100,000 bytes, say) can slow the delivery mechanism. It's sometimes better to put some of the information in a file and just refer to the file in the message.

Message Constants

In addition to named data, a BMessage carries a coded constant that indicates what kind of message it is. The constant is stored in the object's one public data member, called **what**. For example, a message that notifies an application that the user pressed a key on the keyboard has **B_KEY_DOWN** as the **what** data member (and information about the event stored under names like "key", "char", and "modifiers"). An application-defined message that delivers a command to do something might have a constant such as **SORT_ITEMS**, **CORRECT_SPELLING**, or **SCROLL_TO_BOTTOM** in the **what** field. Simple messages can consist of just a constant and no data. A constant like **RECEIPT_ACKNOWLEDGED** or **CANCEL** may be enough to convey a complete message.

By convention, the constant alone is sufficient to identify a message. It's assumed that all messages with the same constant are used for the same purpose and contain the same kinds of data.

The **what** constant must be defined in a protocol known to both sender and receiver. The constants for system messages are defined in **app/AppDefs.h**. Each constant names a kind of event—such as **B_KEY_DOWN**, **B_REFS_RECEIVED**, **B_PULSE**, **B_QUIT_REQUESTED**, and **B_VALUE_CHANGED**—or it carries an instruction to do something (such as **B_ZOOM** and **B_ACTIVATE**).

It's important that the constants you define for your own messages not be confused with the constants that identify system messages. For this reason, we've adopted a strict convention for assigning values to all Be-defined message constants. The value assigned will always be formed by combining four characters into a multicharacter constant; the characters are limited to uppercase letters and the underbar. For example, **B_KEY_DOWN** and **B_VALUE_CHANGED** are defined as follows:

```
enum {
    . . .
    B_KEY_DOWN = '_KYD',
    B_VALUE_CHANGED = '_VCH',
    . . .
};
```

Use a different convention to define your own message constants (or you'll risk having your message misinterpreted as a report of, say, a mouse-moved event). Include some lowercase letters, numerals, or symbols (other than the underbar) in your multicharacter constants, or assign numeric values that can't be confused with the value of four concatenated characters.

Type Codes

Data added to a BMessage is associated with a name and stored with two relevant pieces of information:

- The number of bytes in the data, and
- A type code indicating what kind of data it is.

Type codes are defined in **app/AppDefs.h** for the common data types listed below:

B_CHAR_TYPE	A single character
B_SHORT_TYPE	A **short** integer
B_LONG_TYPE	A **long** integer
B_UCHAR_TYPE	An **unsigned char** (the **uchar** defined type)
B_USHORT_TYPE	An **unsigned short** (the **ushort** defined type)
B_ULONG_TYPE	An **unsigned long** (the **ulong** defined type)
B_BOOL_TYPE	A boolean value (the **bool** defined type)
B_FLOAT_TYPE	A **float**
B_DOUBLE_TYPE	A **double**
B_POINTER_TYPE	A pointer of some type (including **void ***)
B_OBJECT_TYPE	An object pointer (such as BMessage *)
B_MESSENGER_TYPE	A BMessenger object
B_POINT_TYPE	A BPoint object
B_RECT_TYPE	A BRect object
B_RGB_COLOR_TYPE	An **rgb_color** structure
B_PATTERN_TYPE	A **pattern** structure
B_ASCII_TYPE	Text in ASCII format
B_RTF_TYPE	Text in Rich Text Format
B_STRING_TYPE	A null-terminated character string
B_MONOCHROME_1_BIT_TYPE	Raw data for a monochrome bitmap (1 bit/pixel)
B_GRAYSCALE_8_BIT_TYPE	Raw data for a grayscale bitmap (8 bits per pixel)
B_COLOR_8_BIT_TYPE	Raw bitmap data in the **B_COLOR_8_BIT** color space
B_RGB_24_BIT_TYPE	Raw bitmap data in the **B_RGB_32_BIT** color space
B_TIFF_TYPE	Bitmap data in the Tag Image File Format
B_REF_TYPE	A **record_ref**
B_RECORD_TYPE	A **record_id**
B_TIME_TYPE	A representation of a date
B_MONEY_TYPE	A monetary amount
B_RAW_TYPE	Raw, untyped data—a stream of bytes

You can add data to a message even if its type isn't on this list. A BMessage will accept any kind of data; you must simply invent your own codes for unlisted types.

To prevent confusion, the values you assign to the type codes you invent shouldn't match any values assigned to the standard type codes listed above—nor should they match any codes that might be added to the list in the future. The value assigned to all Be-defined type codes is a multicharacter constant, with the characters restricted to uppercase letters and the underbar. For example, **B_DOUBLE_TYPE** and **B_POINTER_TYPE** are defined as follows:

```
enum {
    . . .
    B_DOUBLE_TYPE = 'DBLE',
    B_POINTER_TYPE  = 'PNTR',
    . . .
};
```

This is the same convention used for message constants. Be reserves all such combinations of uppercase letters and underbars for its own use.

Assign values to your constants that can't be mistaken for values that might be assigned in system software. If you assign multicharacter values, make sure at least one of the characters is a lowercase letter, a numeral, or some kind of symbol (other than an underbar). If you assign numeric values, make sure they don't fall in the range 0x41414141 through 0x5f5f5f5f. For example, you might safely define constants like these:

```
#define PRIVATE_TYPE   0x1f3d
#define OWN_TYPE       'Rcrd'
```

Publishing Message Protocols

The messaging system is most interesting—and most useful—when data types are shared by a variety of applications. Shared types open avenues for applications to cooperate with each other. You are therefore encouraged to publish the data types that your application defines and can accept in a BMessage, along with their assigned type codes.

Contact Be (**devsupport@be.com**) to register any types you intend to publish, so that you can be sure to choose a code that hasn't already been adopted by another developer, and we'll endeavor to make sure that no one else usurps the code you've chosen.

If your application can respond to certain kinds of remote messages, you should publish the message protocol—the constant that should initialize the **what** data member of the BMessage, the names of expected data entries, the types of data they contain, the number of data items allowed in each entry, and so on. If your application sends replies to these messages, you should publish the reply protocols as well.

By making the specifications for your messages public, you encourage other applications to make use of the services your application offers, and you contribute to an integrated set of applications on the BeBox.

Error Reporting

BMessage functions that add, find, replace, or get information about message data set a descriptive error code for the object, which the **Error()** function returns. The code is set to **B_NO_ERROR** if all is well; otherwise it indicates what went wrong during the last function call. Some functions also return the error code directly, but some do not.

Before proceeding with the next operation, it's a good idea to call **Error()** to be sure there was no error on the last one.

Data Members

ulong **what**	A coded constant that captures what the message is about. For example, a message that's delivered as the result of a mouse-down event will have **B_MOUSE_DOWN** as its **what** data member. An application that requests information from another application might put a **TRANSMIT_DATA** or **SEND_INFO** command in the **what** field. A message that's posted as the result of the user clicking a Cancel button might simply have **CANCEL** as the **what** data member and include no other information.

Constructor and Destructor

BMessage()

BMessage(ulong *command***)**
BMessage(BMessage **message***)**
BMessage(void**)**

Assigns *command* as the BMessage's **what** data member, and ensures that the object otherwise starts out empty. Given the definition of a message constant such as,

```
#define RECEIPT_ACKNOWLEDGED  0x80
```

a complete message can be created as simply as this:

```
BMessage *msg = new BMessage(RECEIPT_ACKNOWLEDGED);
```

As a public data member, **what** can also be set explicitly. The following two lines of code are equivalent to the one above:

```
BMessage *msg = new BMessage;
msg->what = RECEIPT_ACKNOWLEDGED;
```

Other information can be added to the message by calling **AddData()** or a kindred function.

A BMessage can also be constructed as a copy of another *message*. It's necessary to copy any messages you receive that you want to keep, since the thread that receives the message automatically deletes it before getting the next message. (More typically, you'd copy any data you want to save from the message, but not the BMessage itself.)

As an alternative to copying a received message, you can sometimes detach it from the message loop so that it won't be deleted (see **DetachCurrentMessage()** in the BLooper class).

Messages should be dynamically allocated with the **new** operator, as shown in the examples above, rather than statically allocated on the stack (since they must live on after the functions that send them return).

See also: **BLooper::DetachCurrentMessage()**

~BMessage()

virtual **~BMessage(**void**)**

Frees all memory allocated to hold message data. If the message sender is expecting a reply but hasn't received one, a default reply (with **B_NO_REPLY** as the **what** data member) is sent before the message is destroyed.

Don't delete the messages that you post to a thread, send to another application, or assign to another object. Like letters or parcels sent through the mail, BMessage objects become the property of the receiver. Each message loop routinely deletes the BMessages it receives after the application is finished responding to them.

Member Functions

AddData(), AddBool(), AddLong(), AddFloat(), AddDouble(), AddRef(), AddMessenger(), AddPoint(), AddRect(), AddObject(), AddString()

long **AddData(**const char **name*, ulong *type*, const void **data*, long *numBytes***)**

long **AddBool(**const char **name*, bool *aBool***)**

long **AddLong(**const char **name*, long *aLong***)**

long **AddFloat(**const char **name*, float *aFloat***)**

long **AddDouble(**const char **name*, double *aDouble***)** long **AddRef(**const char **name*, record_ref *aRef***)**

long **AddMessenger(**const char **name*, BMessenger *aRef***)**

long **AddPoint(**const char **name*, BPoint *aPoint***)**

long **AddRect(**const char **name*, BRect *aRect***)**

long **AddObject(**const char **name*, BObject **anObject***)**

long **AddString(**const char **name*, const char **aString***)**

These functions put data in the BMessage. **AddData()** copies *numBytes* of *data* into the object, and assigns the data a *name* and a *type* code. The *type* must be a specific data type; it should not be **B_ANY_TYPE**.

AddData() copies whatever the *data* pointer points to. For example, if you want to add a string of characters to the message, *data* should be the string pointer (**char ***). If you want to add only the string pointer, not the characters themselves, *data* should be a pointer to the pointer (**char ****).

The other functions—**AddBool()**, **AddLong()**, **AddFloat()**, and so on—are simplified versions of **AddData()**. They each add a particular type of data to the message and register it under the appropriate type code, as shown below:

Function	Adds type	Assigns type code
AddBool()	a **bool**	**B_BOOL_TYPE**
AddLong()	a **long** or **ulong**	**B_LONG_TYPE**
AddFloat()	a **float**	**B_FLOAT_TYPE**
AddDouble()	a **double**	**B_DOUBLE_TYPE**
AddRef()	a **record_ref**	**B_REF_TYPE**
AddMessenger()	a BMessenger object	**B_MESSENGER_TYPE**
AddPoint()	a BPoint object	**B_POINT_TYPE**
AddRect()	a BRect object	**B_RECT_TYPE**
AddObject()	a pointer to an object	**B_OBJECT_TYPE**
AddString()	a character string	**B_STRING_TYPE**

Each of these ten type-specific functions calculates the number of bytes in the data they add. **AddString()**, like **AddData()**, takes a pointer to the data it adds. The string must be null-terminated; the null character is counted and copied into the message. The other functions are simply passed the data directly. For example, **AddLong()** takes a **long** and **AddRef()** a **record_ref**, whereas **AddData()** would be passed a pointer to a **long** and a pointer to a **record_ref**. **AddObject()** adds the object pointer it's passed to the message, not the object data structure; **AddData()** would take a pointer to the pointer.

If more than one item of data is added under the same name, the BMessage creates an array of data for that name. Each successive call appends another data element to the end of the array. For example, the following code creates an array named “primes” with 37 stored at index 0, 223 stored at index 1, and 1,049 stored at index 2.

```
BMessage *msg = new BMessage(NUMBERS);
long x = 37;
long y = 223;
long z = 1049;

msg->AddLong("primes", x);
msg->AddFloat("pi", 3.1416);
msg->AddLong("primes", y);
msg->AddData("primes", B_LONG_TYPE, &z, sizeof(long));
```

Note that entering other data between some of the elements of an array—in this case, “pi”—doesn’t increment the array index.

All elements in a named array must be of the same type; it’s an error to try to mix types under the same name.

These functions return **B_ERROR** if the data is too massive to be added to the message, **B_BAD_TYPE** if the data can’t be added to an existing array because it’s the wrong type, or **B_NO_ERROR** if the operation was successful.

See also: **FindData()**, **GetInfo()**

CountNames()

long **CountNames(**ulong *type***)**

Returns the number of named entries in the BMessage that store data of the specified *type*. An array of information held under a single name counts as one entry; each name is counted only once, no matter how many data items are stored under that name.

If *type* is **B_ANY_TYPE**, this function counts all named entries. If *type* is a specific type, it counts only entries that store data registered as that type.

See also: **GetInfo()**

Error()

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

Returns an error code that specifies what went wrong with the last BMessage operation, or **B_NO_ERROR** if there wasn't an error. It's important to check for an error before continuing with any code that depends on the result of a BMessage function. For example:

```
float pi = msg->FindFloat("pi");
if ( msg->Error() == B_NO_ERROR ) {
    float circumference = pi * diameter;
    . . .
}
```

The error code is reset each time a BMessage function is called that adds, finds, alters, or provides information about message data. It's also reset to **B_NO_ERROR** whenever **Error()** itself is called. Cache the return value if you write code that needs to check the current error code more than once.

Possible error returns include the following:

Error code	Is set when
B_NAME_NOT_FOUND	Trying to find, or get information about, data stored under an invalid name
B_BAD_INDEX	Trying to find, or get information about, data stored at an index that's out-of-range
B_BAD_TYPE	Attempting to add data of the wrong type to an existing array, or asking about named data of a given type when the name and type don't match
B_BAD_REPLY	Trying to send a reply to a message that hasn't itself been sent.
B_DUPLICATE_REPLY	Trying to send a reply when one has already been sent and received
< **B_MESSAGE_TO_SELF**	Attempting to send a reply when the source and destination threads are the same >
B_BAD_THREAD_ID	Attempting to send a reply to a thread that no longer exists
B_ERROR	Attempting to add too much data to a message

See also: **AddData()**, **FindData()**, **HasData()**, **GetInfo()**

FindData(), FindBool(), FindLong(), FindFloat(), FindDouble(), FindRef(), FindMessenger(), FindPoint(), FindRect(), FindObject(), FindString()

void ***FindData(**const char **name*, ulong *type*, long **numBytes***)**
void ***FindData(**const char **name*, ulong *type*, long *index,* long **numBytes***)**

bool **FindBool(**const char **name*, long *index* = 0**)**

long **FindLong(**const char **name*, long *index* = 0**)**

float **FindFloat(**const char **name*, long *index* = 0**)**

double **FindDouble(**const char **name*, long *index* = 0**)**

record_ref **FindRef(**const char **name*, long *index* = 0**)**

BMessenger **FindMessenger**const char **name*, long *index* = 0**)**

BPoint **FindPoint(**const char **name*, long *index* = 0**)**

BRect **FindRect(**const char **name*, long *index* = 0**)**

BObject ***FindObject(**const char **name*, long *index* = 0**)**

const char ***FindString(**const char **name*, long *index* = 0**)**

These functions retrieve data from the BMessage. Each looks for data stored under the specified *name*. If more than one data item has the same name, an *index* can be provided to tell the function which item in the *name* array it should find. Indices begin at 0. If an index isn't provided, the function will find the first, or only, item in the array.

FindData() returns a pointer to the requested data item and records the size of the item (the number of bytes it takes up) in the variable referred to by *numBytes*. It asks for data of a specified *type*. If the *type* is **B_ANY_TYPE**, it returns a pointer to the data no matter what type it actually is. But if *type* is a specific data type, it returns the data only if the *name* entry holds data of that particular type.

It's important to keep in mind that **FindData()** always returns a pointer to the data, never the data itself. If the data *is* a pointer—for example, a pointer to an object—it returns a pointer to the pointer. The variable that's assigned the returned pointer must be doubly indirect. For example:

```
MyClass **object;
long numBytes;
object = (MyClass **)message->FindData("name",
                                    B_OBJECT_TYPE, &numBytes);
if ( message->Error() == B_NO_ERROR ) {
    (*object)->GetSomeInformation();
    . . .
}
```

The other functions similarly return the requested item—but do so as a specifically declared data type. They match the corresponding **Add**...**()** functions and search for named data of the declared type, as described below:

Function	Finds data	Registered as type
FindBool()	a **bool**	**B_BOOL_TYPE**
FindLong()	a **long** or **ulong**	**B_LONG_TYPE**
FindFloat()	a **float**	**B_FLOAT_TYPE**
FindDouble)	a **double**	**B_DOUBLE_TYPE**
FindRef()	a **record_ref**	**B_REF_TYPE**
FindMessenger()	a BMessenger object	**B_MESSENGER_TYPE**
FindPoint()	a BPoint object	**B_POINT_TYPE**
FindRect()	a BRect object	**B_RECT_TYPE**
FindObject()	a pointer to an object	**B_OBJECT_TYPE**
FindString()	a character string	**B_STRING_TYPE**

FindString() returns a pointer to a null-terminated string of characters (as would **FindData()**); it expects the null-terminator to have been copied into the message. The rest of the functions return the data directly, not through a pointer. For example, **FindLong()** returns a **long**, whereas **FindData()** would return a pointer to a **long**. **FindObject()** returns a pointer to an object, whereas **FindData()**, as illustrated above, would return a pointer to the pointer to the object.

If you want to keep the data returned by **FindData()** and **FindString()**, you must copy it; it will be destroyed when the BMessage is deleted.

If these functions can't find any data associated with *name*, or if they can't find data in the *name* array at *index*, or if they can't find *name* data of the requested *type* (or the type the function returns), they register an error. You can rely on the values they return only if **Error()** reports **B_NO_ERROR** and the data was correctly recorded when it was added to the message.

When they fail, **FindData()**, **FindString()**, and **FindObject()** return **NULL** pointers. **FindRect()** returns an invalid rectangle and **FindRef()** returns an invalid **record_ref** with both data members set to –1. The other functions return values set to 0, which may be indistinguishable from valid values.

Finding a data item doesn't remove it from the BMessage.

See also: **GetInfo()**, **AddData()**

Flatten(), Unflatten()

void **Flatten(**char ***stream*, long **numBytes***)**

void **Unflatten(**const char **stream***)**

These functions write the data stored in a BMessage to a "flat" (untyped) stream of bytes, and reconstruct a BMessage object from such a stream.

Flatten() allocates enough memory to hold all the information stored in the BMessage object, then copies the information to that memory. It places a pointer to the allocated memory in the variable referred to by the *stream* argument, and writes the number of bytes that were allocated to the variable referred to by *numBytes*. It's the responsibility of the caller to free the memory that **Flatten()** allocates when it's no longer needed. (Since the stream is allocated by **malloc()**, call **free()** to get rid of it.)

Unflatten() empties the BMessage of any information it may happen to contain, then initializes the object from information stored in *stream*. The pointer passed to **Unflatten()** must be to the start of a *stream* that **Flatten()** allocated and initialized. Neither function frees the stream.

GetInfo()

```
bool GetInfo(const char *name, ulong *typeFound, long *countFound = NULL)
bool GetInfo(ulong type, long index,
                    char **nameFound,
                    ulong *typeFound,
                    long *countFound = NULL)
```

Provides information about the data entries stored in the BMessage.

When passed a *name* that matches a name within the BMessage, **GetInfo()** places the type code for data stored under that name in the variable referred to by *typeFound* and writes the number of data items with that name into the variable referred to by *countFound*. It then returns **TRUE**. If it can't find a *name* entry within the BMessage, it registers an error, sets the *countFound* variable to 0, and returns **FALSE** (without modifying the *typeFound* variable).

When passed a *type* and an *index*, **GetInfo()** looks only at entries that store data of the requested type and provides information about the entry at the requested index. Indices begin at 0 and are type specific. For example, if the requested *type* is **B_LONG_TYPE** and the BMessage contains a total of three named entries that store **long** data, the first entry would be at *index* 0, the second at 1, and the third at 2—no matter what other types of data actually separate them in the BMessage, and no matter how many data items each entry contains. (Note that the index in this case ranges over entries, each with a different name, not over the data items within a particular named entry.) If the requested type is **B_ANY_TYPE**, this function looks at all entries and gets information about the one at *index* whatever its type.

If successful in finding data of the *type* requested at *index*, **GetInfo()** returns **TRUE**. It provides information about the entry through the last three arguments:

- It places a pointer to the name of the data entry in the variable referred to by *nameFound*.
- It puts the code for the type of data the entry contains in the variable referred to by *typeFound*. This will be the same as the *type* requested, unless the requested type is **B_ANY_TYPE**, in which case *typeFound* will be the actual type stored under the name.

- It records the number of data items stored within the entry in the variable referred to by *countFound*.

If **GetInfo()** can't find data of the requested *type* at *index*, it registers an error, sets the *countFound* variable to 0, and returns **FALSE**.

This version of **GetInfo()** can be used to iterate through all the BMessage's data. For example:

```
char  *name;
ulong  type;
long   count;

for ( long i = 0;
      msg->GetInfo(B_ANY_TYPE, i, &name, &type, &count);
      i++ ) {
    . . .
}
```

If the index is incremented from 0 in this way, all data of the requested type will have been read when **GetInfo()** returns **FALSE**. If the requested type is **B_ANY_TYPE**, as shown above, it will reveal the name and type of every entry in the BMessage.

See also: **HasData()**, **AddData()**, **FindData()**

HasData(), HasBool(), HasLong(), HasFloat(), HasDouble(), HasRef(), HasMessenger(), HasPoint(), HasRect(), HasObject(), HasString()

bool **HasData(**const char **name*, ulong *type*, long *index* = 0**)**

bool **HasBool(**const char **name*, long *index* = 0**)**

bool **HasLong(**const char **name*, long *index* = 0**)**

bool **HasFloat(**const char **name*, long *index* = 0**)**

bool **HasDouble(**const char **name*, long *index* = 0**)**

bool **HasRef(**const char **name*, long *index* = 0**)**

bool **HasMessenger**const char **name*, long *index* = 0**)**

bool **HasPoint(**const char **name*, long *index* = 0**)**

bool **HasRect(**const char **name*, long *index* = 0**)**

bool **HasObject(**const char **name*, long *index* = 0**)**

bool **HasString(**const char **name*, long *index* = 0**)**

These functions test whether the BMessage contains data of a given name and type.

If *type* is **B_ANY_TYPE** and no *index* is provided, **HasData()** returns **TRUE** if the BMessage stores any data at all under the specified *name*, regardless of its type, and **FALSE** if the name passed doesn't match any within the object.

If *type* is a particular type code, **HasData()** returns **TRUE** only if the BMessage has a *name* entry that stores data of that type. If the *type* and *name* don't match, it returns **FALSE**.

If an *index* is supplied, **HasData()** returns **TRUE** only if the BMessage has a *name* entry that stores a data item of the specified *type* at that particular *index*. If the index is out of range, it returns **FALSE**.

The other functions—**HasBool()**, **HasFloat()**, **HasPoint()**, and so on—are specialized versions of **HasData()**. They test for a particular type of data stored under the specified *name*.

An error code is set (which **Error()** will return) whenever any of these functions returns **FALSE**.

See also: **GetInfo()**

IsEmpty() *see* MakeEmpty()

IsReply() *see* WasSent()

IsSourceRemote() *see* WasSent()

IsSourceWaiting() *see* WasSent()

IsSystem()

bool **IsSystem(**void**)**

Returns **TRUE** if the **what** data member of the BMessage object identifies it as a system-defined message, and **FALSE** if not.

Unlike the **GetInfo()** and **HasData()** functions, a return of **FALSE** does not indicate an error. **IsSystem()** resets the error code that **Error()** returns to **B_NO_ERROR** whether the BMessage is a system message or not.

MakeEmpty(), IsEmpty()

long **MakeEmpty(**void**)**

bool **IsEmpty(**void**)**

MakeEmpty() removes and frees all data that has been added to the BMessage, without altering the **what** constant. It returns **B_NO_ERROR**.

IsEmpty() returns **TRUE** if the BMessage has no data (whether or not it was emptied by **MakeEmpty()**), and **FALSE** if it has some.

Both functions reset the error code to **B_NO_ERROR** in all cases.

See also: **RemoveName()**

Previous() *see* **WasSent()**

PrintToStream()

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

Prints information about the BMessage to the standard output stream (**stdout**). Each entry of named data is reported in the following format,

```
#entry name, type = type, count = count
```

where *name* is the name that the data is registered under, *type* is the constant that indicates what type of data it is, and *count* is the number of data items in the named array.

RemoveName()

bool **RemoveName(**const char **name***)**

Removes all data entered in the BMessage under *name*, frees the memory that was allocated to hold the data, and returns **TRUE**. If there is no data entered under *name*, this function registers an error (**B_NAME_NOT_FOUND**) and returns **FALSE**.

See also: **MakeEmpty()**

ReplaceData(), ReplaceBool(), ReplaceLong(), ReplaceFloat(), ReplaceDouble(), ReplaceRef(), ReplaceMessenger(), ReplacePoint(), ReplaceRect(), ReplaceObject(), ReplaceString()

long **ReplaceData(**const char **name*, ulong *type*,
    const void **data*, long *numBytes***)**
long **ReplaceData(**const char **name*, ulong *type*, long *index*,
    const void **data*, long *numBytes***)**

long **ReplaceBool(**const char **name*, bool *aBool***)**
long **ReplaceBool(**const char **name*, long *index*, bool *aBool***)**

long **ReplaceLong(**const char **name*, long *aLong***)**
long **ReplaceLong(**const char **name*, long *index*, long *aLong***)**

long **ReplaceFloat(**const char **name*, float *aFloat***)**
long **ReplaceFloat(**const char **name*, long *index*, float *aFloat***)**

long **ReplaceDouble(**const char **name*, double *aDouble***)**
long **ReplaceDouble(**const char **name*, long *index*, double *aDouble***)**

long **ReplaceRef(**const char **name*, record_ref *aRef***)**
long **ReplaceRef(**const char **name*, long *index*, record_ref *aRef***)**

long **ReplaceMessenger(**const char **name*, BMessenger *aMessenger***)**
long **ReplaceMessenger(**const char **name*, long *index*, BMessenger *aMessenger***)**

long **ReplacePoint(**const char **name*, BPoint *aPoint***)**
long **ReplacePoint(**const char **name*, long *index*, BPoint *aPoint***)**

long **ReplaceRect(**const char **name*, BRect *aRect***)**
long **ReplaceRect(**const char **name*, long *index*, BRect *aRect***)**

long **ReplaceObject(**const char **name*, BObject **anObject***)**
long **ReplaceObject(**const char **name*, long *index*, BObject **anObject***)**

long **ReplaceString(**const char **name*, const char **aString***)**
long **ReplaceString(**const char **name*, long *index*, const char **aString***)**

These functions replace a data item in the *name* entry with another item passed as an argument. If an *index* is provided, they replace the item in the *name* array at that index; if an *index* isn't mentioned, they replace the first (or only) item stored under *name*. If an *index* is provided but it's out-of-range, the replacement fails.

ReplaceData() replaces an item in the *name* entry with *numBytes* of *data*, but only if the *type* code that's specified for the data matches the type of data that's already stored in the entry. The *type* must be specific; it can't be **B_ANY_TYPE**.

The other functions are simplified versions of **ReplaceData()**. They each handle the specific type of data declared for their last arguments. They succeed if this type matches the type of data already in the *name* entry, and fail if it does not.

If successful, all these functions return **B_NO_ERROR**. If unsuccessful, they register and return an error code—**B_BAD_INDEX** if the *index* is out-of-range, **B_NAME_NOT_FOUND** if the *name* entry doesn't exist, or **B_BAD_TYPE** if the entry doesn't contain data of the specified type.

See also: **AddData()**

ReturnAddress() *see* WasSent()

SendReply()

long **SendReply(**BMessage **message*, BMessage ***reply***)**
long **SendReply(**ulong *command*, BMessage ***reply***)**
long **SendReply(**BMessage **message*, BHandler **replyTarget* = NULL**)**
long **SendReply(**ulong *command*, BHandler **replyTarget* = NULL**)**

Sends a reply *message* back to the sender of the BMessage (in the case of a synchronous reply) or to a target BHandler (in the case of an asynchronous reply). Whether the reply is synchronous or asynchronous depends on how the message it replies to was sent:

- The reply is delivered synchronously if the message sender is waiting for it to arrive. The function that sent the BMessage doesn't return until it receives the reply. If an expected reply has not been sent by the time the BMessage object is deleted, a default **B_NO_REPLY** message is returned to the sender.

- The reply is delivered asynchronously if the message sender isn't waiting for a reply. In this case, the sending function designates a target BHandler (and, through the BHandler, a target BLooper) for any replies that might be sent, then returns immediately after putting the BMessage in the pipeline. The default target for a reply is the sender's BApplication object.

SendReply() works only for BMessage objects that have been processed through a message loop and delivered to you. However, it doesn't work for messages that were posted to the loop, only for those that were sent or dragged. If it's called when a reply isn't allowed, the *message* is deleted and an error is recorded.

The *message* that's passed to **SendReply()** should not be modified, passed to another messaging function, used as a model message, or deleted. It becomes the responsibility of the messaging service and the eventual receiver.

If a *command* is passed rather than a *message*, **SendReply()** constructs the reply BMessage, initializes its **what** data member with the *command* constant, and sends it just like any other reply.

If you want to delay sending a reply and keep the BMessage object beyond the time it's scheduled to be deleted, you may be able to detach it from the message loop. See **DetachCurrentMessage()** in the BLooper class.

SendReply() sends a message—a reply message, to be sure, but a message nonetheless. It therefore is just another message-sending function. It behaves exactly like the other message-sending function, BMessenger's **SendMessage()**:

- By passing it a *reply* argument, you can ask for a synchronous reply to the reply message it sends. It won't return until it receives the reply.

- By supplying a *targetHandler* argument, you can arrange for an expected asynchronous reply. If a specific target isn't specified, the BApplication object will handle the reply if one is sent.

This function returns **B_NO_ERROR** if the reply is successfully sent. If not, it returns one of the error codes explained under the **Error()** function.

See also: **BMessenger::SendMessage()**, **BLooper::DetachCurrentMessage()**, **WasSent()**, **Error()**

Unflatten() *see* Flatten()

WasDropped(), DropPoint()

bool **WasDropped(**void**)**

BPoint **DropPoint(**BPoint **offset* = NULL**)**

WasDropped() returns **TRUE** if the user delivered the BMessage by dragging and dropping it, and **FALSE** if the message was posted or sent in application code or if it hasn't yet been delivered at all.

DropPoint() reports the point where the cursor was located when the message was dropped (when the user released the mouse button). It directly returns the point in the screen coordinate system and, if an *offset* argument is provided, returns it by reference in coordinates based on the image or rectangle the user dragged. The *offset* assumes a coordinate system with (0.0, 0.0) at the left top corner of the dragged rectangle or image.

Since any value can be a valid coordinate, **DropPoint()** produces reliable results only if **WasDropped()** returns **TRUE**.

See also: **BView::DragMessage()**

WasSent(), IsSourceRemote(), IsSourceWaiting(), IsReply(), Previous(), ReturnAddress()

bool **WasSent(**void**)**

bool **IsSourceRemote(**void**)**

bool **IsSourceWaiting(**void**)**

bool **IsReply(**void**)**

BMessage ***Previous(**void**)**

BMessenger **ReturnAddress(**void**)**

These functions can help if you're engaged in an exchange of messages or managing an ongoing communication.

WasSent() indicates whether it's possible to send a reply to a message. It returns **TRUE** for a BMessage that was sent or dropped, and **FALSE** for a message that was posted or has not yet been delivered by any means. (When, in a future release, it's possible to reply to a posted message, this function would be more clearly named **WasDelivered()**.) Regardless of the return value, **WasSent()** sets the current error code to **B_NO_ERROR**.

IsSourceRemote() returns **TRUE** if the message had its source in another application, and **FALSE** if the source is local or the message hasn't been delivered yet. It resets the error code to **B_NO_ERROR** in both cases.

IsSourceWaiting() returns **TRUE** if the message sender is waiting for a synchronous reply, and **FALSE** if not. The sender can request and wait for a reply when calling either BMessenger's **SendMessage()** or BMessage's **SendReply()** function.

IsReply() returns **TRUE** if the BMessage is a reply to a previous message (if it was sent by the **SendReply()** function), and **FALSE** if not. It resets the error code to **B_NO_ERROR** in either case.

Previous() returns the previous message, or **NULL** if the BMessage isn't a reply.

ReturnAddress() returns a BMessenger that can be used to reply to the BMessage. Calling the BMessenger's **SendMessage()** function is equivalent to calling **SendReply()**, except that the return message won't be marked as a reply. If a reply isn't allowed (if the BMessage wasn't sent or dropped), a **B_BAD_VALUE** error is registered to indicate that the returned BMessenger is invalid. Call **Error()** to check. If the BMessenger is valid, **Error()** will return **B_NO_ERROR**.

If you want to use the **ReturnAddress()** BMessenger to send a synchronous reply, you must do so before the BMessage is deleted and default reply is sent.

See also: **BMessenger::SendMessage()**, **SendReply()**

Operators

new

void *__operator new(__size_t *numBytes*__)__

Allocates memory for a BMessage object, or takes the memory from a previously allocated cache. The caching mechanism is an efficient way of managing memory for objects that are created frequently and used for short periods of time, as BMessages typically are.

delete

void **operator delete(**void **memory*, size_t *numBytes***)**

Frees memory allocated by the BMessage version of **new**, which may mean restoring the memory to the cache.

BMessageFilter

Derived from:	public BObject
Declared in:	<app/MessageFilter.h>

Overview

A BMessageFilter is an object that holds a hook function, **Filter()**, that can look at incoming messages before they're dispatched to their designated handlers. The object also keeps the conditions that must be met for the function to be called. Applications implement the **Filter()** function in classes derived from BMessageFilter.

A BMessageFilter can be attached to a message loop in one of two ways:

- If assigned to a BHandler object, the filter will be applied only to messages targeted to the BHandler.
- If assigned to a BLooper object as a common filter, it can be applied to any message regardless of the designated target. (A BLooper can also be assigned specific filters in its role as a BHandler.)

All applicable filters in both categories are applied to a message before it's dispatched to the target BHandler. Common filters are applied before handler-specific filters.

The same BMessageFilter object can be assigned to more than one BHandler or BLooper object; it will not be destroyed when the BHandler or BLooper is deleted.

See also: **BHandler::SetFilterList()**, **BLooper::SetCommonFilterList()**

Hook Functions

Filter()	Implemented by derived classes to respond to a incoming message before the message is dispatched to a target BHandler.

Constructor and Destructor

BMessageFilter()

BMessageFilter(message_delivery *delivery*, message_source *source***)**
BMessageFilter(message_delivery *delivery*, message_source *source*,
ulong *command***)**

Initializes the BMessageFilter object so that its **Filter()** function will be called for every incoming message that meets the specified *delivery*, *source*, and *command* criteria. The first argument, *delivery*, is a constant that specifies how the message must arrive:

B_DROPPED_DELIVERY	Only messages that were dragged and dropped should be filtered.
B_PROGRAMMED_DELIVERY	Only messages that were posted or sent in application code (by calling **PostMessage()** or a **Send...()** function) should be filtered.
B_ANY_DELIVERY	All messages, no matter how they were delivered, should be filtered.

The second argument, *source*, specifies where the message must originate:

B_LOCAL_SOURCE	Only messages that originate locally, from within the application, should be filtered.
B_REMOTE_SOURCE	Only messages that are delivered from a remote source should be filtered.
B_ANY_SOURCE	All messages, no matter what their source, should be filtered.

Filtering can also be limited to a particular type of message. If a *command* constant is specified, only messages that have **what** data members matching the constant will be filtered. If a *command* isn't specified, the command constant won't be a criterion in selecting which messages to filter; any message that meets the other criteria will be filtered, no matter what its **what** data member may be.

The filtering criteria are conjunctive; an arriving message must meet all the criteria specified for **Filter()** to be called.

See also: **Filter()**

~BMessageFilter()

virtual **~BMessageFilter(**void**)**

Does nothing.

Member Functions

Command(), FiltersAnyCommand()

inline ulong **Command(**void**)**

inline bool **FiltersAnyCommand(**void**)**

Command() returns the command constant (**what** data member) that an arriving message must match for the filter to apply. **FiltersAnyCommand()** returns **TRUE** if the filter applies to messages regardless of their **what** data members, and **FALSE** if it's limited to a certain type of message.

Because all command constants are valid, including negative numbers and 0, **Command()** returns a reliable result only if **FiltersAnyCommand()** returns **FALSE**.

See also: the BMessageFilter constructor, the BMessage class

Filter()

virtual filter_result **Filter(**BMessage **message*, BHandler ***target***)**

Implemented by derived classes to examine an arriving message just before it's dispatched. The *message* is passed as the first argument; the second argument indirectly points to the *target* BHandler object that's slated to respond to the message.

You can implement this function to do anything you please with the *message*, including replace the designated *target* with another BHandler object. For example:

```
filter_result MyFilter::Filter(BMessage *msg, BHandler **target)
{
    . . .
    if ( *target->IsIndisposed() )
        *target = *target->FindReplacement();
    . . .
    return B_DISPATCH_MESSAGE;
}
```

The replacement target must be associated with the same BLooper as the original target. If the new target has filters that apply to the *message*, those filtering functions will be called before the message is dispatched.

This function should return a constant that instructs the BLooper whether or not to dispatch the message as planned:

B_DISPATCH_MESSAGE	Dispatch the message.
B_SKIP_MESSAGE	Don't dispatch the message and don't filter it any further; this function took care of handling it.

The default (BMessageFilter) version of this function does nothing but return **B_DISPATCH_MESSAGE**.

See also: the BMessageFilter constructor

FiltersAnyCommand() *see* Command()

MessageDelivery()

inline message_delivery **MessageDelivery(**void**)**

Returns the constant, set when the BMessageFilter object was constructed, that describes the category of messages that can be filtered, based on how they were delivered.

See also: the BMessageFilter constructor

MessageSource()

inline message_source **MessageSource(**void**)**

Returns the constant, set when the BMessageFilter object was constructed, that describes the category of messages that can be filtered, based on the source of the message.

See also: the BMessageFilter constructor

BMessageQueue

Derived from:	public BObject
Declared in:	<app/MessageQueue.h>

Class Description

A BMessageQueue maintains a queue where messages (BMessage objects) are temporarily stored as they wait to be received in a message loop. Every BLooper object uses a BMessageQueue to manage the flow of incoming messages; all messages delivered to the BLooper's thread are placed in the queue. The BLooper removes the oldest message from the queue, passes it to a BHandler, waits for the thread to finish its response, deletes the message, then returns to the queue to get the next message.

For the most part, applications can ignore the queue—that is, they can treat it as an implementation detail. Messages are posted to a thread (placed in the queue) by calling BLooper's **PostMessage()** function. Or they can be sent to the main thread of another application by constructing a BMessenger object and calling **SendMessage()**.

A BLooper calls upon a BHandler's **MessageReceived()** function—and other, message-specific hook functions—to handle the messages it takes from the queue. Applications can simply implement the functions that are called to respond to received messages and not bother about the mechanics of the message loop and queue.

However, if necessary, you can manipulate the queue directly, or perhaps just look ahead to see what messages are coming. The BLooper has a **MessageQueue()** function that returns its BMessageQueue object.

See also: the BMessage class, **BLooper::MessageQueue()**

Constructor and Destructor

BMessageQueue()

BMessageQueue(void**)**

Ensures that the queue starts out empty. Messages are placed in the queue by calling **AddMessage()** and are removed by calling **NextMessage()**.

BMessageQueues are constructed by BLooper objects.

See also: **AddMessage()**, **NextMessage()**

~BMessageQueue()

virtual **~BMessageQueue(**void**)**

Deletes all the objects in the queue and all the data structures used to manage the queue.

Member Functions

AddMessage()

void **AddMessage(**BMessage **message***)**

Adds *message* to the queue.

See also: **NextMessage()**

CountMessages()

long **CountMessages(**void**)** const

Returns the number of messages currently in the queue.

FindMessage()

BMessage ***FindMessage(**ulong *what*, long *index* = 0**)** const
BMessage ***FindMessage(**long *index***)** const

Returns a pointer to the BMessage that's positioned in the queue at *index*, where indices begin at 0 and count only those messages that have **what** data members matching the *what* value passed as an argument. If a *what* argument is omitted, indices count all messages in the queue. If an *index* is omitted, the first message that matches the *what* constant is found. The lower the index, the longer the message has been in the queue.

If no message matches the specified *what* and *index* criteria, this function returns **NULL**.

The returned message is not removed from the queue.

See also: **NextMessage()**

IsEmpty()

bool **IsEmpty(**void**)** const

Returns **TRUE** if the BMessageQueue contains no messages, and **FALSE** if it has at least one.

See also: **CountMessages()**

Lock(), Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

These functions lock and unlock the BMessageQueue, so that another thread won't alter the contents of the queue while it's being read. **Lock()** doesn't return until it has the queue locked; it always returns **TRUE**. **Unlock()** releases the lock so that someone else can lock it. Calls to these functions can be nested.

See also: **BLooper::Lock()**

NextMessage()

BMessage ***NextMessage(**void**)**

Returns the next message—the message that has been in the queue the longest—and removes it from the queue. If the queue is empty, this function returns **NULL**.

RemoveMessage()

void **RemoveMessage(**BMessage **message***)**

Removes a particular *message* from the queue and deletes it.

See also: **FindMessage()**

Unlock() *see* Lock()

BMessenger

Derived from: public BObject

Declared in: <app/Messenger.h>

Overview

A BMessenger object is an agent for sending messages to a particular destination. Each BMessenger knows about a BLooper object and a specific BHandler for that BLooper. The messages it sends are delivered to the BLooper and—provided they're not system messages—dispatched by the BLooper to the BHandler. The destination objects can belong to the same application as the message sender, but typically are in a remote application. It's more efficient to post a message within the same application than to ask a BMessenger to send it.

BMessenger objects can be transported across application boundaries. You can create one for a particular BLooper/BHandler combination in your application, then pass it by value to a remote application. That application can then use the BMessenger to target the BHandler in your application. This is, in fact, the only way for an application to get a BMessenger that can target a remote object other than a BApplication object.

Constructor and Destructor

BMessenger()

BMessenger(ulong *signature*, team_id *team* = –1**)**
BMessenger(const BHandler **target***)**
BMessenger(const BMessenger &*messenger***)**
BMessenger(void**)**

Initializes the BMessenger so that it can send messages to an application identified by its *signature* or by its *team*. The application must be running when the BMessenger is constructed.

If the *signature* passed is **NULL**, the application is identified by its team only. If the *team* specified is –1, as it is by default, the application is identified by its signature only. If both a real *signature* and a valid *team* identifier are passed, they must match—the *team* must be for the application that the *signature* identifies. If more than one instance of the *signature* application happens to be running, the *team* picks out a particular instance as the

BMessenger's target. Without a valid *team* argument, the constructor arbitrarily picks one of the instances.

BMessengers constructed in this way send messages to the main thread of the remote application, where they're received and handled by that application's BApplication object. This type of messenger is needed to initiate communication with another application.

A BMessenger can also be an agent for a *target* BHandler object. It sends messages to the BLooper associated with the BHandler, and the BLooper dispatches them to the BHandler.

The *target* BHandler object must be able to tell the BMessenger (through its **Looper()** function) which BLooper object it's associated with. The BMessenger asks for this information at the time of construction. Therefore, the *target* must either be a BLooper itself or have been added to a BLooper's list of eligible handlers. < For the BMessenger to remain valid, the *target* BHandler must retain its affiliation with the same BLooper. >

The purpose of constructing a BMessenger for a local target is to give a remote application access to that object. You can add the BMessenger to a message and send the message to the remote application. That application can then use the BMessenger to target the BHandler in your application.

A BMessenger can also be constructed as a copy of another BMessenger,

```
BMessenger newOne(anotherMessenger);
```

or be assigned from another object:

```
BMessenger newOne = anotherMessenger;
```

If the constructor can't make a connection to the *signature* application—possibly because no such application is running—it registers a **B_BAD_VALUE** error, which the **Error()** function will return. If passed an invalid *team* identifier, it registers a **B_BAD_TEAM_ID** error. If the *team* and the *signature* don't match, it registers a **B_MISMATCHED_VALUES** error. If it can't discover a BLooper from the *target* BHandler, it registers a **B_BAD_HANDLER** error.

It's a good idea to check for an error before asking the new BMessenger to send a message. For example:

```
BMessenger *outlet = new BMessenger(some_signature);
if ( outlet->Error() == B_NO_ERROR ) {
    BMessage *msg = new BMessage(CHANGE_NAME);
    msg->AddString("old", formerName);
    msg->AddString("new", currentName);
    outlet->SendMessage(msg);
    if ( outlet->Error() == B_NO_ERROR )
        . . .
}
```

A BMessenger can send messages to only one destination. Once constructed, you can cache it and reuse it repeatedly to communicate with that object. It should be freed after it's no longer needed (or if there's a long delay between messages and it's possible that the user might have quit the destination application and restarted it again, or that the application may have destroyed the target BHandler).

The BRoster object can provide signature and team information about possible destinations.

See also: the BRoster and BMessage classes, **Error()**

~BMessenger()

~BMessenger(void**)**

Frees all memory allocated by the BMessenger, if any was allocated at all.

Member Functions

Error()

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

Returns an error code that describes what went wrong with the attempt to construct the BMessenger or to have it send a message, or **B_NO_ERROR** if nothing went wrong. Possible errors include:

B_BAD_VALUE	The constructor can't connect the BMessenger to the remote application, most likely because an application with the specified signature isn't running.
B_MISMATCHED_VALUES	The constructor failed because the specified signature and team arguments designated two different applications.
B_BAD_TEAM_ID	The constructor can't establish a connection to the specified team, most likely because there is no such team.
B_BAD_HANDLER	The BHandler passed to the constructor was not associated with a BLooper.
B_BAD_PORT_ID	**SendMessage()** can't deliver the message, most likely because the destination application has been killed.

Calling this function resets the error code to **B_NO_ERROR**, so you must cache the value returned if you need to check the current error more than once.

FindHandler(), FindAllHandlers()

BMessenger **FindHandler(**BMessage **message***)**
BMessenger **FindHandler(**long *index*, const char **class* = NULL**)**
BMessenger **FindHandler(**const char **name*, const char **class* = NULL**)**

BMessage ***FindAllHandlers(**const char **class* = NULL**)**

These functions send a **B_HANDLERS_REQUESTED** message and wait for a **B_HANDLERS_INFO** reply.

The *message* is passed to **FindHandler()** should have **B_HANDLERS_REQUESTED** as the **what** data member and should ask for a BMessenger for just one BHandler. If an *index* or a *name* is passed instead of a *message*, **FindHandler()** creates the message and adds that information in an entry named "index" or "name". If the index or name is restricted to a *class*, it adds the class name in an entry labeled "class".

When it gets the reply, **FindHandler()** returns the requested BMessenger. It may register a **B_ERROR**, **B_NAMED_NOT_FOUND**, or **B_BAD_INDEX** error taken from the reply, or a **B_BAD_PORT_ID** error if there's a problem sending the message. In the case of any error, the BMessenger is not to be trusted.

FindAllHandlers() requests BMessengers for a group of BHandlers, which may be restricted to a particular *class*. It returns the **B_HANDLERS_INFO** reply, or **NULL** if there's an error in sending a message.

See the various **HandlersRequested()** functions for information on the protocols that the software kits currently expect the **B_HANDLERS_REQUESTED** and **B_HANDLERS_INFO** messages to follow.

See also: **BHandler::HandlersRequested()**

IsValid()

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

Returns **TRUE** if the destination BLooper object to which the BMessenger sends messages remains valid, and **FALSE** if not (if, for example, it has been deleted).

This function doesn't check whether the target BHandler is still valid; it reports only on the status of the destination BLooper.

SendMessage()

> long **SendMessage(**BMessage **message*, BMessage ***reply***)**
> long **SendMessage(**ulong *command*, BMessage ***reply***)**
> long **SendMessage(**BMessage **message*, BHandler **replyTarget* = NULL**)**
> long **SendMessage(**ulong *command*, BHandler **replyTarget* = NULL**)**

Sends a *message*. The BMessage object becomes the responsibility of the BMessenger. You shouldn't try to modify it, post it, send it again, use it as a model message, or free it; it will be freed automatically when it's no longer needed.

If a *command* is passed instead of a full *message*, this function constructs a BMessage object with *command* as its **what** data member and sends it just like any other message. This is simply a convenience for sending messages that contain no data. The following two lines of code are roughly equivalent:

```
myMessenger->SendMessage(NEVERMORE);
myMessenger->SendMessage(new BMessage(NEVERMORE));
```

This function can ask for a synchronous reply to the message or designate a BHandler for an asynchronous reply:

- Supplying a *reply* argument requests a message back from the destination. Before returning, **SendMessage()** waits for the reply and places a pointer to the BMessage it receives in the variable that *reply* refers to.

  The caller is responsible for deleting the *reply* message. If the destination doesn't send a reply, the system sends one with **B_NO_REPLY** as the **what** data member. Check the reply message before proceeding. If there's an error in sending the message, the variable that *reply* refers to is set to **NULL**.

- If a *reply* isn't requested, **SendMessage()** returns immediately; any reply to the *message* will be received asynchronously. If a *replyTarget* is specified, the reply will be directed to that BHandler object. If one isn't specified, it will be directed to the BApplication object.

  The *replyTarget* is subject to the same restriction as a target BHandler passed to the BMessenger constructor: It must be associated with a BLooper object (or be a BLooper itself) < and it must retain that association until the reply arrives >.

If all goes well, **SendMessage()** returns **B_NO_ERROR**. If not, it returns an error code, typically **B_BAD_PORT_ID**. The return value is also registered with the **Error()** function; see that function for more information.

(It's an error for a thread to send a message to itself and expect a synchronous reply. The thread can't respond to the message and wait for a reply at the same time.)

See also: **BMessage::SendReply()**

Team()

inline team_id **Team(**void**)**

Returns the identifier for the team that receives the messages the BMessenger sends.

Operators

= (assignment)

BMessenger &**operator =(**const BMessenger &*messenger***)**

Assigns one BMessenger to another. After the assignment the two objects are identical and independent copies of each other, with no shared data.

new

void ***operator new(**size_t *numBytes***)**

Prevents confusion with a private version of the **new** operator used internally by the Application Kit. This version of **new** is no different from the operator used with other classes.

BRoster

Derived from:	*none*
Declared in:	<app/Roster.h>

Overview

The BRoster object keeps a roster of all applications currently running on the BeBox. It can provide information about any of those applications, add another application to the roster by launching it, or get information about an application to help you decide whether to launch it.

There's just one roster and it's shared by all applications. When an application starts up, a global variable, **be_roster**, is initialized to point to the shared object. You always access the roster through this variable; you never directly instantiate a BRoster in application code.

The BRoster identifies applications in three ways:

- By **record_ref** references to the executable files where they reside.
- By their signatures. The signature is a unique identifier for the application assigned in a resource at compile time or by the BApplication constructor at run time. You can obtain signatures for the applications you develop by contacting Be's developer support staff. They can also tell you what the signatures of other applications are. (See the introduction to this chapter for more on signatures.)
- At run time, by their **team_id**s. A team is a group of threads sharing an address space; every application is a team.

If an application is launched more than once, the roster will include one entry for each instance of the application that's running. These instances will have the same signature, but different team identifiers.

Constructor and Destructor

The BRoster class doesn't have a public constructor or destructor. This is because an application doesn't need to construct or destroy a BRoster of its own. The system constructs one BRoster object for all applications and assigns it to the **be_roster** global variable. A BRoster is therefore readily available from the time the application is launched until the time it quits.

Member Functions

GetAppInfo(), GetRunningAppInfo(), GetActiveAppInfo()

long **GetAppInfo(**ulong *signature*, app_info **appInfo***)** const
long **GetAppInfo(**record_ref *executable*, app_info **appInfo***)** const

long **GetRunningAppInfo(**team_id *team*, app_info **appInfo***)** const

long **GetActiveAppInfo(**app_info **appInfo***)** const

These functions provide information about the application identified by its *signature*, by a database reference to its *executable* file, by its *team*, or simply by its status as the current active application. They place the information in the structure referred to by *appInfo*.

GetRunningAppInfo() reports on a particular instance of a running application, the one that was assigned the *team* identifier at launch. **GetActiveAppInfo()** similarly reports on a running application, the one that happens to be the current active application.

If it can, **GetAppInfo()** also tries to get information about an application that's running. If a running application has the *signature* identifier or was launched from the *executable* file, **GetAppInfo()** queries it for the information. If more than one instance of the *signature* application is running, or if more than one instance was launched from the same *executable* file, it arbitrarily picks one of the instances to report on.

Even if the application isn't running—if none of the applications currently in the roster are identified by *signature* or were launched from the *executable* file—**GetAppInfo()** can still provide some information about it, perhaps enough information for you to call **Launch()** to get it started.

If they're able to fill in the **app_info** structure with meaningful values, these functions return **B_NO_ERROR**. However, **GetActiveAppInfo()** returns **B_ERROR** if there's no active application. **GetRunningAppInfo()** returns **B_BAD_TEAM_ID** if *team* isn't, on the face of it, a valid team identifier for a running application. **GetAppInfo()** returns **B_BAD_VALUE** if the *signature* doesn't correspond to an application on-disk, and simply **B_ERROR** if the *executable* doesn't refer to a valid record in the database or doesn't refer to a record for an executable file.

The **app_info** structure contains the following fields:

ulong **signature**	The signature of the application. (This will be the same as the *signature* passed to **GetAppInfo()**.)
thread_id **thread**	The identifier for the application's main thread of execution, or –1 if the application isn't running. (The main thread is the thread in which the application is launched and in which its **main()** function runs.)
team_id **team**	The identifier for the application's team, or –1 if the application isn't running. (This will be the same as the *team* passed to **GetRunningAppInfo()**.)
port_id **port**	The port where the application's main thread receives messages, or –1 if the application isn't running.
record_ref **ref**	A reference to the file that was, or could be, executed to run the application. (This will be the same as the *executable* passed to **GetAppInfo()**.)
ulong **flags**	A mask that contains information about the behavior of the application.

The **flags** mask can be tested (with the bitwise **&** operator) against these two constants:

B_BACKGROUND_APP	The application won't appear in the Browser's application menu (because it doesn't have a user interface).
B_ARGV_ONLY	The application can't receive messages. Information can be passed to it at launch only, in an array of argument strings (as on the command line).

The **flags** mask also contains a value that explains the application's launch behavior. This value must be filtered out of **flags** by combining **flags** with the **B_LAUNCH_MASK** constant. For example:

```
ulong behavior = theInfo.flags & B_LAUNCH_MASK;
```

The result will match one of these three constants:

B_EXCLUSIVE_LAUNCH	The application can be launched only if an application with the same signature isn't already running.
B_SINGLE_LAUNCH	The application can be launched only once from the same executable file. However, an application

	with the same signature might be launched from a different executable. For example, if the user copies an executable file to another directory, a separate instance of the application can be launched from each copy.
B_MULTIPLE_LAUNCH	There are no restrictions. The application can be launched any number of times from the same executable file.

These flags affect BRoster's **Launch()** function. **Launch()** can always start up a **B_MULTIPLE_LAUNCH** application. However, it can't launch a **B_SINGLE_LAUNCH** application if a running application was already launched from the same executable file. It can't launch a **B_EXCLUSIVE_LAUNCH** application if an application with the same signature is already running.

See also: "Launch Information" on page 19 of the chapter introduction, **Launch()**, **BApplication::GetAppInfo()**

GetAppList()

void **GetAppList(**BList **teams***)** const
void **GetAppList(**ulong *signature*, BList **teams***)** const

Fills in the *teams* BList with team identifiers for applications in the roster. Each item in the list will be of type **team_id**. It must be cast to that type when retrieving it from the list, as follows:

```
team_id who = (team_id)teams->ItemAt(someIndex);
```

The list will contain one item for each instance of an application that's running. For example, if the same application has been launched three times, the list will include the **team_id**s for all three running instances of that application.

If a *signature* is passed, the list identifies only applications running under that signature. If a *signature* isn't specified, the list identifies all running applications.

See also: **TeamFor()**, the BMessenger constructor

IsRunning() *see* TeamFor()

Launch()

long **Launch(**ulong *signature*, BMessage **message* = NULL,
team_id **team* = NULL**)**
long **Launch(**ulong *signature*, BList **messages*,
team_id **team* = NULL**)**
long **Launch(**ulong *signature*, long *argc*, char ***argv*,
team_id **team* = NULL**)**

long **Launch(**record_ref *executable*, BMessage **message* = NULL,
team_id **team* = NULL**)**
long **Launch(**record_ref *executable*, BList **messages*,
team_id **team* = NULL**)**
long **Launch(**record_ref *executable*, long *argc*, char ***argv*,
team_id **team* = NULL**)**

Launches the application identified by its *signature* or by a reference to its *executable* file in the database.

If a *message* is specified, it will be sent to the application on-launch where it will be received and responded to before the application is notified that it's ready to run. Similarly, if a list of *messages* is specified, each one will be delivered on-launch. The BMessage objects (and the container BList) will be deleted for you.

Sending an on-launch message is appropriate only if it helps the launched application configure itself before it starts getting other messages. To launch an application and send it an ordinary message, call **Launch()** to get it running, then set up a BMessenger object for the application and call BMessenger's **SendMessage()** function.

Instead of messages, you can launch an application with an array of argument strings that will be passed to its **main()** function. *argv* contains the array and *argc* counts the number of strings. If the application accepts messages, this information will also be packaged in a **B_ARGV_RECEIVED** message that the application will receive on-launch.

If successful, **Launch()** places the identifier for the newly launched application in the variable referred to by *team* and returns **B_NO_ERROR**. If unsuccessful, it sets the *team* variable to –1, destroys all the messages it was passed (and the BList that contained them), and returns one of the following error codes:

B_BAD_VALUE	The *signature* passed is not valid or it doesn't designate an available application. This return value may also signify that an attempt is being made to send an on-launch message to an application that doesn't accept messages (that is, to a **B_ARGV_ONLY** application).
B_ERROR	The *executable* file can't be found.

B_ALREADY_RUNNING	The application is already running and can't be launched again (it's a **B_SINGLE_LAUNCH** or **B_EXCLUSIVE_LAUNCH** application).
B_LAUNCH_FAILED	The attempt to launch the application failed for some other reason, such as insufficient memory.

See also: the BMessenger class, **GetAppInfo()**

RemoveApp()

void **RemoveApp(**team_id *team***)**

Removes the application identified by *team* from the roster of running applications.

TeamFor(), IsRunning()

team_id **TeamFor(**ulong *signature***)** const
team_id **TeamFor(**record_ref *executable***)** const

bool **IsRunning(**ulong *signature***)** const
bool **IsRunning(**record_ref *executable***)** const

Both these functions query whether the application identified by its *signature*, or by a reference to its *executable* file in the database, is running. **TeamFor()** returns its team identifier if it is, and **B_ERROR** if it's not. **IsRunning()** returns **TRUE** if it is, and **FALSE** if it's not.

If the application is running, you probably will want its team identifier (to set up a BMessenger, for example). Therefore, it's most economical to simply call **TeamFor()** and forego **IsRunning()**.

If more than one instance of the *signature* application is running, or if more than one instance was launched from the same *executable* file, **TeamFor()** arbitrarily picks one of the instances and returns its **team_id**.

See also: **GetAppList()**

Global Variables, Constants, and Defined Types

This section lists the global variables, constants, and defined types that are defined by the Application Kit. There's just a few defined types, three global variables—**be_app**, **be_roster**, and **be_clipboard**—and a handful of constants. Error codes are documented in the chapter on the Support Kit.

Although the Application Kit defines the constants for all system messages (such as **B_REFS_RECEIVED**, **B_ACTIVATE**, and **B_KEY_DOWN**), only those that mark system management and application messages are listed here. Those that designate interface messages are documented in the chapter on the Interface Kit.

Global Variables

be_app

<app/Application.h>

BApplication ***be_app**

This variable provides global access to your application's BApplication object. It's initialized by the BApplication constructor.

See also: the BApplication class

be_clipboard

<app/Clipboard.h>

BClipboard ***be_clipboard**

This variable gives applications access to the shared repository of data for cut, copy, and paste operations. It's initialized at startup; an application has just one BClipboard object.

See also: the BClipboard class

be_roster

<app/Roster.h>

BRoster ***be_roster**

This variable points to the global BRoster object that's shared by all applications. The BRoster keeps a roster of all running applications and can add applications to the roster by launching them.

See also: the BRoster class

Constants

Application Flags

<app/Roster.h>

Defined constant
B_BACKGROUND_APP
B_ARGV_ONLY
B_LAUNCH_MASK

These constants are used to get information from the **flags** field of an **app_info** structure.

See also: **BRoster::GetAppInfo()**, "Launch Constants" below

Application Messages

<app/AppDefs.h>

Enumerated constant	Enumerated constant
B_ACTIVATE	**B_ARGV_RECEIVED**
B_READY_TO_RUN	**B_REFS_RECEIVED**
B_APP_ACTIVATED	**B_PANEL_CLOSED**
B_ABOUT_REQUESTED	**B_PULSE**
B_QUIT_REQUESTED	

These constants represent the system messages that are received and recognized by the BApplication class. Application messages concern the application as a whole, rather than any particular window thread. See the introduction to this chapter and the BApplication class for details.

See also: "Application Messages" on page 16 of the chapter introduction, "System Management Messages" on page 113 below

Cursor Constants

<app/AppDefs.h>

const unsigned char **B_HAND_CURSOR**[]
const unsigned char **B_I_BEAM_CURSOR**[]

These constants contain all the data needed to set the cursor to the default hand image or to the standard I-beam image for text selection.

See also: **BApplication::SetCursor()**

Data Type Codes

<app/AppDefs.h>

Enumerated constant	Enumerated constant
B_BOOL_TYPE	**B_ASCII_TYPE**
B_CHAR_TYPE	**B_STRING_TYPE**
B_UCHAR_TYPE	**B_RTF_TYPE**
B_SHORT_TYPE	**B_PATTERN_TYPE**
B_USHORT_TYPE	**B_RGB_COLOR_TYPE**
B_LONG_TYPE	**B_RECORD_TYPE**
B_ULONG_TYPE	**B_TIME_TYPE**
B_FLOAT_TYPE	**B_MONEY_TYPE**
B_DOUBLE_TYPE	**B_RAW_TYPE**
B_POINTER_TYPE	**B_MONOCHROME_1_BIT_TYPE**
B_OBJECT_TYPE	**B_GRAYSCALE_8_BIT_TYPE**
B_POINT_TYPE	**B_COLOR_8_BIT_TYPE**
B_RECT_TYPE	**B_RGB_24_BIT_TYPE** [sic]
B_MESSENGER_TYPE	**B_TIFF_TYPE**
B_REF_TYPE	**B_ANY_TYPE**

These constants are used in a BMessage object to describe the types of data the message holds. **B_ANY_TYPE** refers to all types; the others refer only to a particular type. See the BMessage class for more information on what they mean.

See also: “Type Codes” on page 71 of the BMessage class overview

filter_result Constants

<app/MessageFilter.h>

Enumerated constant

B_SKIP_MESSAGE
B_DISPATCH_MESSAGE

These constants list the possible return values of a filter function.

See also: **BMessageFilter::Filter()**

Launch Constants

<app/Roster.h>

Defined constant

B_MULTIPLE_LAUNCH
B_SINGLE_LAUNCH
B_EXCLUSIVE_LAUNCH

These constants explain whether an application can be launched any number of times, only once from a particular executable file, or only once for a particular application signature. This information is part of the **flags** field of an **app_info** structure and can be extracted using the **B_LAUNCH_MASK** constant.

See also: **BRoster::GetAppInfo()**, “Application Flags” above

Message Constants

<app/AppDefs.h>

Enumerated constant

B_NO_REPLY
B_MESSAGE_NOT_UNDERSTOOD

B_HANDLERS_INFO

B_SIMPLE_DATA

B_CUT
B_COPY
B_PASTE

These constants mark messages that the system sometimes puts together, but that aren’t dispatched like system messages. See “Standard Messages” in the *Message Protocols* appendix for details.

See also: **BMessage::SendReply()**, the BTextView class in the Interface Kit, **BHandler::HandlersRequested()**

message_delivery Constants

<app/MessageFilter.h>

Enumerated constant

B_ANY_DELIVERY
B_DROPPED_DELIVERY
B_PROGRAMMED_DELIVERY

These constants distinguish the delivery criterion for the application of a BMessageFilter.

See also: the BMessageFilter constructor

message_source Constants

<app/MessageFilter.h>

Enumerated constant

B_ANY_SOURCE
B_REMOTE_SOURCE
B_LOCAL_SOURCE

These constants list the possible constraints on the message source for the application of a BMessageFilters.

See also: the BMessageFilter constructor

System Management Messages

<app/AppDefs.h>

Enumerated constant

B_QUIT_REQUESTED
B_HANDLERS_REQUESTED

These constants represent system messages that are used to help run the messaging system. They're received and recognized by generic BLooper objects.

See also: "System Management Messages" on page 15 of the introduction, "Application Messages" on page 110 above

Defined Types

app_info

<app/Roster.h>

```
typedef struct {
      ulong signature;
      thread_id thread;
      team_id team;
      port_id port;
      record_ref ref;
      ulong flags;
} app_info
```

This structure is used by BRoster's **GetAppInfo()**, **GetRunningAppInfo()**, and **GetActiveAppInfo()** functions to report information about an application. See those functions for a description of its various fields.

See also: **BRoster::GetAppInfo()**

filter_result

<app/MessageFilter.h>

typedef enum { . . . } **filter_result**

This type distinguishes between the **B_SKIP_MESSAGE** and **B_DISPATCH_MESSAGE** return values for a filter function.

See also: **BMessageFilter::Filter()**

message_delivery

<app/MessageFilter.h>

typedef enum { . . . } **message_delivery**

This type enumerates the delivery criteria for filtering a message.

See also: the BMessageFilter constructor

message_source

<app/MessageFilter.h>

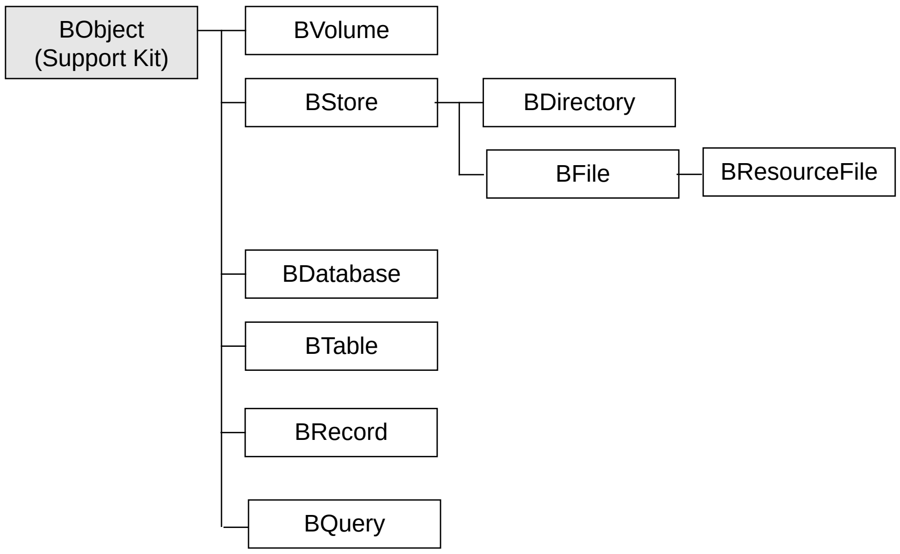
typedef enum { . . . } **message_source**

This type enumerates the source criteria for filtering a message.

See also: the BMessageFilter constructor

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:

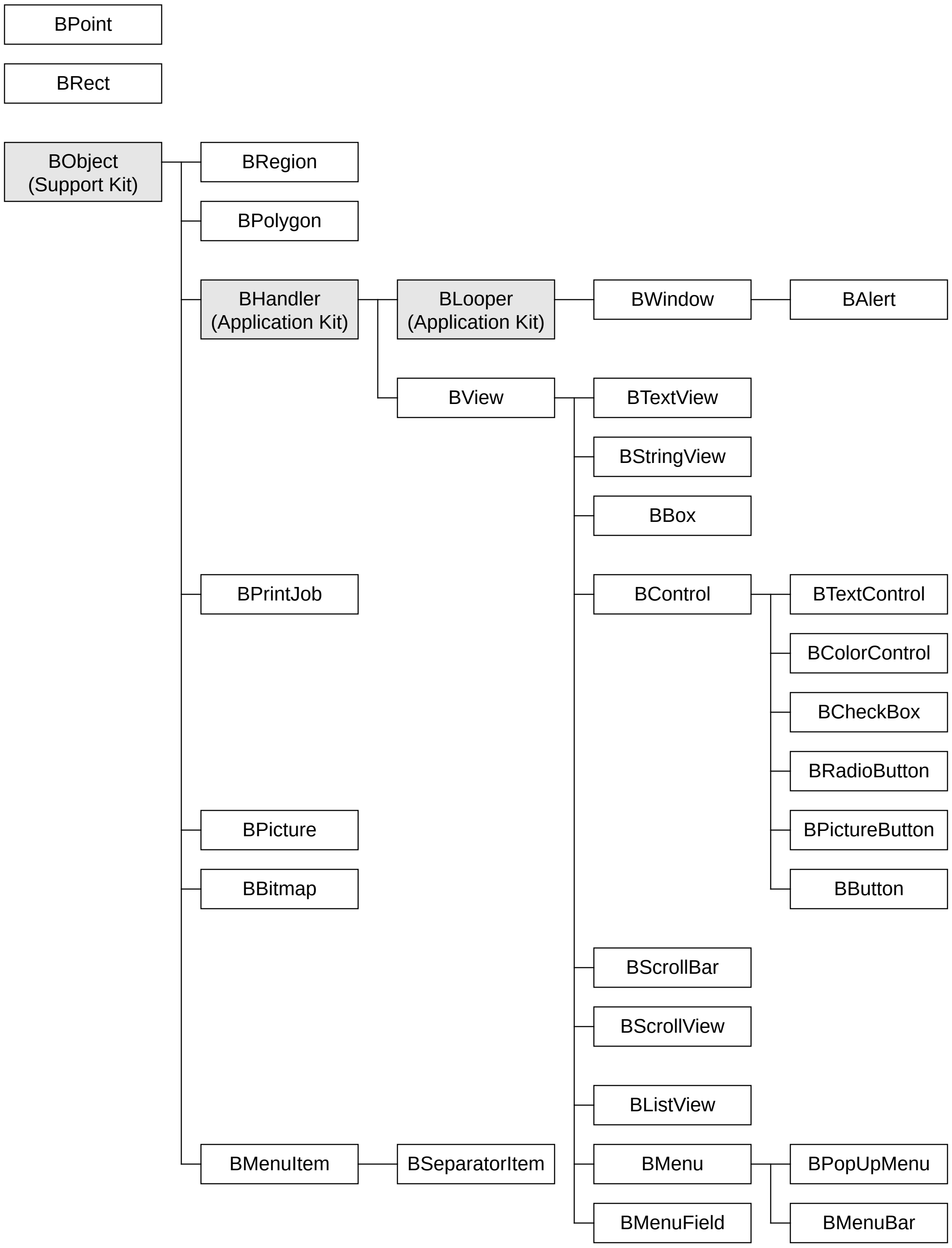
- The 'ICON' resource that's named "BAPP" holds the icon that's displayed for the application.
- The 'ICON' that takes, as a name, the application's signature converted to a string holds the data that's displayed for documents created by the application.

'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.

4 The Interface Kit

Interface Kit Inheritance Hierarchy



4 The Interface Kit

Most Be applications have an interactive and graphical user interface. When they start up, they present themselves to the user on-screen in one or more windows. The windows display areas where the user can do something—there may be menus to open, buttons to click, text fields to type in, images to drag, and so on. Each user action on the keyboard or mouse is packaged as an *interface message* and reported to the application. The application responds to each message as it is received. At least part of the response is always a change in what the window displays—so that users can see the results of their work.

To run this kind of user interface, an application has to do three things. It must:

- Manage a set of windows,
- Draw within the windows, and
- Respond to interface messages.

The application, in effect, carries on a conversation with the user. It draws to present itself on-screen, the user does something with the keyboard or mouse, the event is reported to the application in a message, and the application draws in response, prompting more user actions and more messages.

The Interface Kit structures this interaction with the user. It defines a set of C++ classes that give applications the ability to manage windows, draw in them, and efficiently respond to the user's instructions. Taken together, these classes define a framework for interactive applications. By programming with the Kit, you'll be able to construct an application that effectively uses the capabilities of the BeBox.

This chapter first introduces the conceptual framework for the user interface, then describes all the classes, functions, types, and constants the Kit defines. The reference material that follows this introduction assumes the concepts and terminology presented here.

Framework for the User Interface

A graphical user interface is organized around windows. Each window has a particular role to play in an application and is more or less independent of other windows. While working on the computer, users think in terms of windows—what's in them and what can be done with them—perhaps more than in terms of applications.

The design of the software mirrors the way the user interface works: it's also organized around windows. Within an application, each window runs in its own thread and is represented by a separate BWindow object. The object is the application's interface to the window the system provides; the thread is where all the work that's centered on the window takes place.

Because every window has its own thread, the user can, for example, scroll the contents of one window while watching an animation in another, or start a time-consuming computation in an application and still be able to use the application's other windows. A window won't stop working when the user turns to another window.

Commands that the user gives to a particular window initiate activity within that window's thread. When the user clicks a button within a window, for example, everything that happens in response to the click happens in the window thread (unless the application arranges for other threads to be involved). In its interaction with the user, each window acts on its own, independently of other windows.

Application Server Windows

In a multitasking environment, any number of applications might be running at the same time, each with its own set of windows on-screen. The windows of all running applications must cooperate in a common interface. For example, there can be only one active window at a time—not one per application, but one per machine. A window that comes to the front must jump over every other window, not just those belonging to the same application. When the active window is closed, the window behind it must become active, even if it belongs to a different application.

Because it would be difficult for each application to manage the interaction of its windows with every other application, windows are assigned, at the lowest level, to a separate entity, the Application Server. The Server's principal role in the user interface is to provide applications with the windows they require.

Everything a program or a user does is centered on the windows the Application Server provides. Users type into windows, click buttons in windows, drag images to windows, and so on; applications draw in windows to display the text users type, the buttons they can click, and the images they can drag.

The Application Server, therefore, is the conduit for an application's message input and drawing output:

- It monitors the keyboard and mouse and sends messages reporting each user keystroke and mouse action to the application.
- It receives drawing instructions from the application and interprets them to render images within windows.

The Server relieves applications of much of the burden of basic user-interface work. The Interface Kit organizes and further simplifies an application's interaction with the Server.

BWindow Objects

Every window in an application is represented by a separate BWindow object. Constructing the BWindow establishes a connection to the Application Server—one separate from, but initially dependent on, the connection previously established by the BApplication object. The Server creates a window for the new object and dedicates a separate thread to it.

The BWindow object is a kind of BLooper, so it spawns a thread for the window in the application's address space and begins running a message loop where it receives and responds to interface messages from the Server. The window thread in the application is directly connected to the dedicated thread in the Server.

The BWindow object, therefore, is in position to serve three crucial roles:

- It can act as the application's interface to a Server window. It has functions that the application can call to manipulate the window programmatically—move it, resize it, close it, and so on. It also declares the hook functions that the system calls to notify the application that the user manipulated the window.
- It can organize message-handling within the window thread. Since it runs the window's message loop, it gets to decide how each message should be handled. It's the focus and central distribution point for all messages that initiate activity in the thread.
- As the entity that holds rendered images, it can manage the objects that produce those images. (This is discussed under "BView Objects" below.)

All other Interface Kit objects play roles that depend on a BWindow. They draw in a window, respond to interface messages received by a window, or act in support of other objects that draw and respond to messages.

BView Objects

For purposes of drawing and message-handling, a window can be divided up into smaller rectangular areas called *views*. Each view corresponds to one part of what the window displays—a scroll bar, a document, a list, a button, or some other more or less self-contained portion of the window's contents.

An application sets up a view by constructing a BView object and associating it with a particular BWindow. The BView object is responsible for drawing within the view rectangle, and for handling interface messages directed at that area.

Drawing Agent

A window is a tablet that can retain and display rendered images, but it can't draw them; for that it needs a set of BViews. A BView is an agent for drawing, but it can't render the images it creates; for that it needs a BWindow. The two kinds of objects work hand in hand.

Each BView object is an autonomous graphics environment for drawing. Some aspects of the environment, such as the list of possible colors, are shared by all BViews and all applications. But within those broad limits, every BView maintains an independent graphics state. It has its own coordinate system, current colors, drawing mode, clipping region, pen position, and so on.

The BView class defines the functions that applications call to carry out elemental drawing tasks—such as stroking lines, filling shapes, drawing characters, and imaging bitmaps. These functions are typically used to implement another function—called **Draw()**—in a class derived from BView. This view-specific function draws the contents of the view rectangle.

The BWindow will call the BView's **Draw()** function whenever the window's contents (or at least the part that the BView has control over) need to be updated. A BWindow first asks its BViews to draw when the window is initially placed on-screen. Thereafter, they might be asked to refresh the contents of the window whenever the contents change or when they're revealed after being hidden or obscured. A BView might be called upon to draw at any time.

Because **Draw()** is called on the command of others, not the BView, it can be considered to draw *passively*. It presents the view as it currently appears. For example, the **Draw()** function of a BView that displays editable text would draw the characters that the user had inserted up to that point.

BViews also draw *actively* in response to messages reporting the user's actions. For example, text is highlighted as the user drags over it and is replaced as the user types. Each change is the result of a system message reported to the BView. For passive drawing, the BView implements a function (**Draw()**) that others may call. For active drawing, it calls the drawing functions itself (it may even call **Draw()**).

Message Handler

The drawing that a BView does is often designed to prompt a user response of some kind—an empty text field with a blinking caret invites typed input, a menu item or a button invites a click, an icon looks like it can be dragged, and so on.

When the user acts, system messages that report the resulting events are sent to the BWindow object, which determines which BView elicited the user action and should respond to it. For example, a BView that draws typed text can expect to respond to messages reporting the user's keystrokes. A BView that draws a button gets to handle the messages that are generated when the button is clicked. The BView class derives from BHandler, so BView objects are eligible to handle messages dispatched by the BWindow.

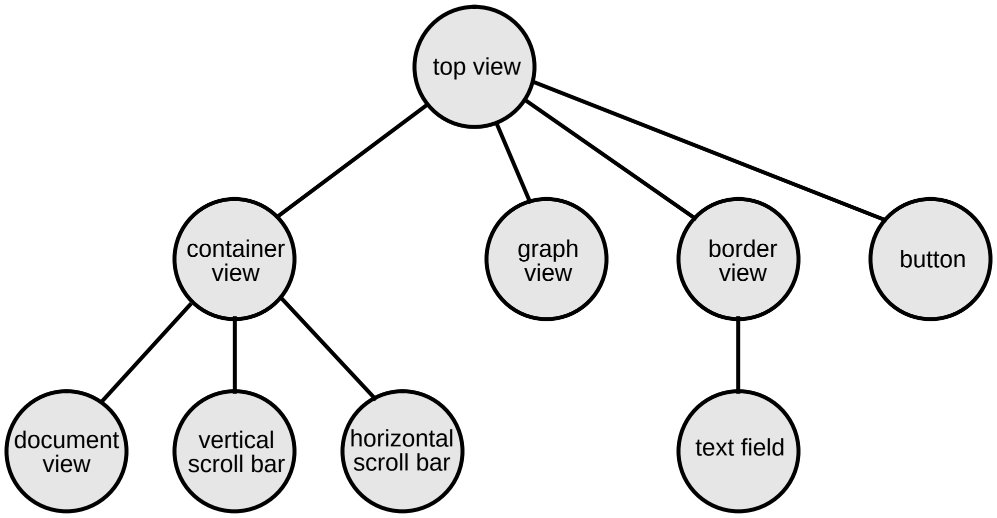
Just as classes derived from BView implement **Draw()** functions to draw within the view rectangle, they also implement the hook functions that respond to interface messages. These functions are discussed later, under “Hook Functions for Interface Messages” on page 44.

Largely because of its graphics role and its central role in handling interface messages, BView is the biggest and most diverse class in the Interface Kit. Most other Interface Kit classes are derived from it.

The View Hierarchy

A window typically contains a number of different views—all arranged in a hierarchy beneath the *top view*, a view that’s exactly the same size as the content area of the window. The top view is a companion of the window; it’s created by the BWindow object when the BWindow is constructed. When the window is resized, the top view is resized to match. Unlike other views, the top view doesn’t draw or respond to messages; it serves merely to connect the window to the views that the application creates and places in the hierarchy.

As illustrated in the diagram below, the view hierarchy can be represented as a branching tree structure with the top view at its root. All views in the hierarchy (except the top view) have one, and only one, parent view. Each view (including the top view) can have any number of child views.



In this diagram, the top view has four children, the container view has three, and the border view one. Child views are located within their parents, so the hierarchy is one of overlapping rectangles. The container view, for example, takes up some of the top view’s area and divides its own area into a document view and two scroll bars.

When a new BView object is created, it isn’t attached to a window and it has no parent. It’s added to a window by making it a child of a view already in the view hierarchy. This is done with the **AddChild()** function. A view can be made a child of the window’s top view by calling BWindow’s version of **AddChild()**.

Until it’s assigned to a window, a BView can’t draw and won’t receive reports of events. BViews know how to produce images, but it takes a window to display and retain the images they create.

Drawing and Message-Handling in the View Hierarchy

The view hierarchy determines what’s displayed where on-screen, and also how user actions are associated with the responsible BView object:

- When the views in a window are called upon to draw, parents draw before their children; children draw in front of their ancestors.
- Mouse events (like the mouse-down and mouse-up events that result from a click) are associated with the view where the cursor is located. Since the cursor points to the frontmost view at any given location, it’s likely to be pointing at a view close to the bottom of the hierarchy. It’s those views—the ones that have no children—that are responsible for most of the drawing and message-handling for the window. Views farther up the hierarchy tend to contain and organize those at the bottom.

Overlapping Siblings

Although children wait for their parents when it comes time to draw and parents defer to their offspring when it comes to time to respond to interface messages, sibling views are not so well-behaved. Siblings don’t draw in any predefined order. This doesn’t matter, as long as the view rectangles of the siblings don’t overlap. If they do overlap, it’s indeterminate which view will draw last—that is, which one will draw on top of the other.

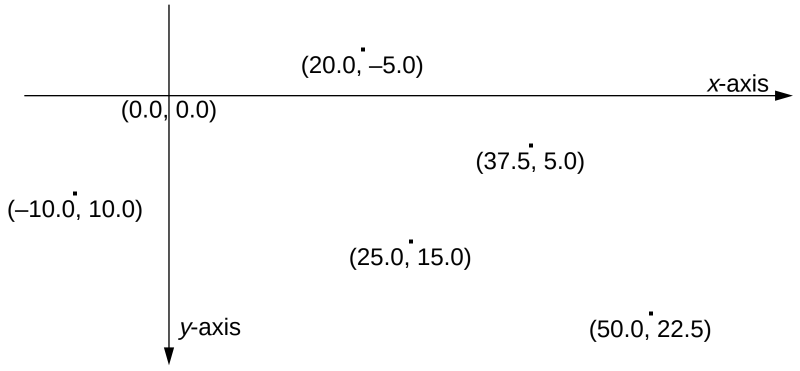
Similarly, it’s indeterminate which view will be associated with mouse events in the area the siblings share. It may be one view or it may be the other, and it won’t necessarily be the one that drew the image the user sees.

Therefore, it’s strongly recommended that sibling views should be arranged so that they don’t overlap.

The Coordinate Space

To locate windows and views, draw in them, and report where the cursor is positioned over them, it’s necessary to have some conventional way of talking about the display surface. The same conventions are used whether the display device is a monitor that shows images on a screen or a printer that puts them on a page.

In Be software, the display surface is described by a standard two-dimensional coordinate system where the *y*-axis extends downward and the *x*-axis extends to the right, as illustrated below:



y coordinate values are greater towards the bottom of the display and smaller towards the top, *x* coordinate values are greater to the right and smaller to the left.

The axes define a continuous coordinate space where distances are measured by floating-point values (**float**s). All quantities in this space—including widths and heights, *x* and *y* coordinates, font sizes, angles, and the size of the pen—are floating point numbers.

Floating-point coordinates permit precisely stated measurements that can take advantage of display devices with higher resolutions than the screen. For example, a vertical line 0.4 units wide would be displayed using a single column of pixels on-screen, the same as a line 1.4 units wide. However, a 300 dpi printer would use two pixel columns to print the 0.4-unit line and six to print the 1.4-unit line.

A coordinate unit is 1/72 of an inch, roughly equal to a typographical point. However, all screens are considered to have a resolution of 72 pixels per inch (regardless of the actual dimension), so coordinate units count screen pixels. One unit is the distance between the centers of adjacent pixels on-screen.

Coordinate Systems

Specific coordinate systems are associated with the screen, with windows, and with the views inside windows. They differ only in where the two axes are located:

- The global or *screen coordinate system* has its origin, (0.0, 0.0), at the left top corner of the screen. It’s used for positioning windows on-screen, < for arranging multiple screens connected to the same machine, > and for comparing coordinate values that weren’t originally stated in a common coordinate system.

- A *window coordinate system* has its origin at the left top corner of the content area of a window. It’s used principally for positioning views within the window. Each window has its own coordinate system so that locations within the window can be specified without regard to where the window happens to be on-screen.

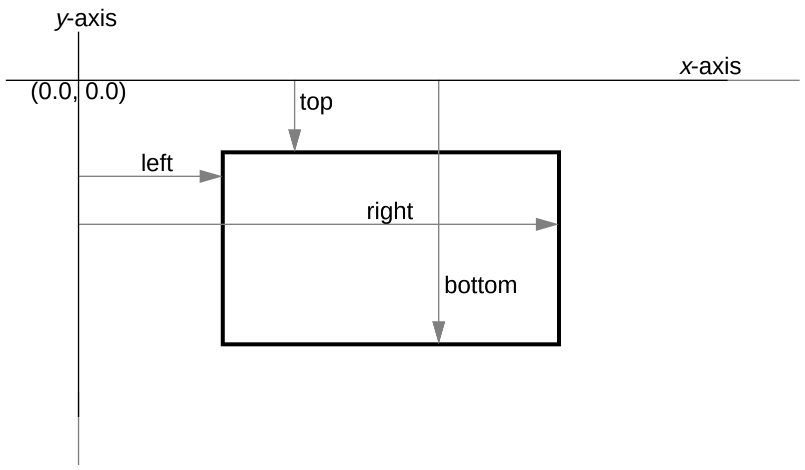
- A *view coordinate system* has its default origin at the left top corner of the view rectangle. However, scrolling can shift view coordinates and move the origin. View-specific coordinates are used for all drawing operations and to report the cursor location in most system messages.

Coordinate Geometry

The Interface Kit defines a handful of basic classes for locating points and areas within a coordinate system:

- A BPoint object is the simplest way to specify a coordinate location. Each object stores two values—an *x* coordinate and a *y* coordinate—that together locate a specific point, (*x*, *y*), within a given coordinate system.

- A BRect object represents a rectangle; it's the simplest way to designate an area within a coordinate system. The BRect class defines a rectangle as a set of four coordinate values—corresponding to the rectangle's left, top, right, and bottom edges, as illustrated below:



The sides of the rectangle are therefore parallel to the coordinate axes. The left and right sides delimit the range of *x* coordinate values within the rectangle, and the top and bottom sides delimit the range of *y* coordinate values. For example, if a rectangle's left top corner is at (0.8, 2.7) and its right bottom corner is at (11.3, 49.5), all points having *x* coordinates ranging from 0.8 through 11.3 and *y* coordinates from 2.7 through 49.5 lie inside the rectangle.

If the top of a rectangle is the same as its bottom, or its left the same as its right, the rectangle defines a straight line. If the top and bottom are the same and also the left and right, it collapses to a single point. Such rectangles are still valid—they specify real locations within a coordinate system. However, if the top is greater than the bottom or the left greater than the right, the rectangle is invalid; it has no meaning.

- A BPolygon object represents a polygon, a closed figure with an arbitrary number of sides. The polygon is defined as an ordered set of points. It encloses the area that would be outlined by connecting the points in order, then connecting the first and last points to close the figure. Each point is therefore a potential vertex of the polygon.
- A BRegion object defines a set of points. A region can be any shape and even include discontinuous areas.

Mapping Coordinates to Pixels

The device-independent coordinate space described above must be mapped to the pixel grid of a particular display device—the screen, a printer, or some other piece of hardware that's capable of rendering an image. For example, to display a rectangle, it's necessary to find the pixel columns that correspond to its right and left sides and the pixel rows that correspond to its top and bottom.

This depends entirely on the resolution of the device. In essence, each device-independent coordinate value must be translated internally to a device-dependent value—an integer index to a particular column or row of pixels. In the coordinate space of the device, one unit equals one pixel.

This translation is easy for the screen, since, as mentioned above, there's a one-to-one correspondence between coordinate units and pixels. It reduces to rounding floating-point coordinates to integers. For other devices, however, the translation means first scaling the coordinate value to a device-specific value, then rounding. For example, the point (12.3, 40.8) would translate to (12, 41) on the screen, but to (51, 170) on a 300 dpi printer.

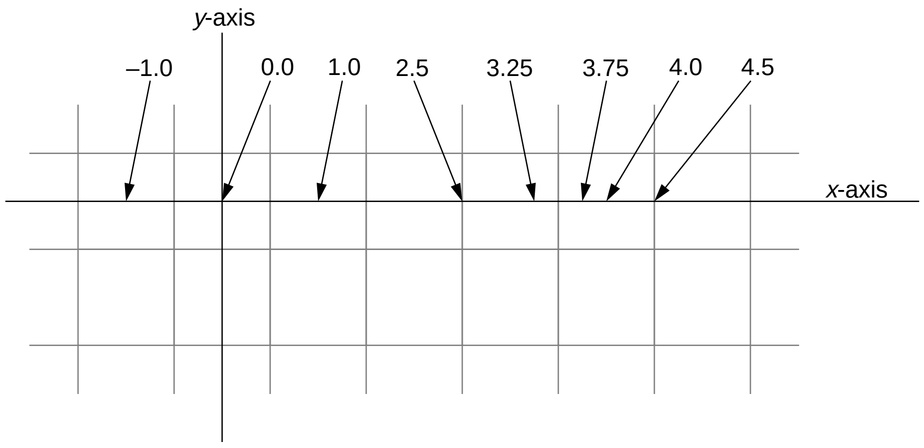
Screen Pixels

To map coordinate locations to device-specific pixels, you need to know only two things:

- The resolution of the device, and
- The location of the coordinate axes relative to pixel boundaries.

The axes are located in the same place for all devices: The *x*-axis runs left to right along the middle of a row of pixels and the *y*-axis runs down the middle of a pixel column. They meet at the very center of a pixel.

Because coordinate units match pixels on the screen, this means that all integral coordinate values (those without a fractional part) fall midway across a screen pixel. The following illustration shows where various *x* coordinate values fall on the *x*-axis. The broken lines represent the division of the screen into a pixel grid:



As this illustration shows, it's possible to have coordinate values that lie on the boundary between two pixels. A later section, "Picking Pixels to Stroke and Fill" on page 34, describes how these values are mapped to one pixel or the other.

Drawing

Drawing is done by BView objects. As discussed above, the views within a window are organized into a hierarchy—there can be views within views—but each view is an independent drawing agent and maintains a separate graphics environment. This section discusses the framework in which BViews draw, beginning with view coordinate systems. Detailed descriptions of the functions mentioned here can be found in the BView and BWindow class descriptions.

View Coordinate Systems

As a convenience, each view is assigned a coordinate system of its own. By default, the coordinate origin—(0.0, 0.0)—is located at the left top corner of the view rectangle. (For an overview of the coordinate systems assumed by the Interface Kit, see "The Coordinate Space" on page 14 above.)

When a view is added as a child of another view, it's located within the coordinate system of its parent. A child is considered part of the contents of the parent view. If the parent moves, the child moves with it; if the parent view scrolls its contents, the child view is shifted along with everything else in the view.

Since each view retains its own internal coordinate system no matter who its parent is, where it's located within the parent, or where the parent is located, a BView's drawing and message-handling code doesn't need to be concerned about anything exterior to itself. To do its work, a BView need look no farther than the boundaries of its own view rectangle.

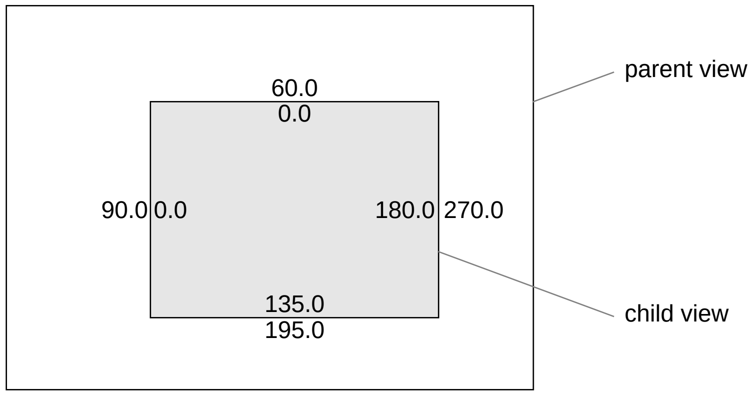
Frame and Bounds Rectangles

Although a BView doesn't have to look outside its own boundaries, it does have to know where those boundaries are. It can get this information in two forms:

- Since a view is located within the coordinate system of its parent, the view rectangle is initially defined in terms of the parent's coordinates. This defining rectangle for a view is known as its *frame rectangle*. (See the BView constructor and the **Frame()** function.)

- When translated from the parent's coordinates to the internal coordinates of the view itself, the same rectangle is known as the *bounds rectangle*. (See the **Bounds()** function.)

The illustration below shows a child view 180.0 units wide and 135.0 units high. When viewed from the outside, from the perspective of its parent's coordinate system, it has a frame rectangle with left, top, right, and bottom coordinates at 90.0, 60.0, 270.0, and 195.0, respectively. But when viewed from the inside, in the view's own coordinate system, it has a bounds rectangle with coordinates at 0.0, 0.0, 180.0, and 135.0:



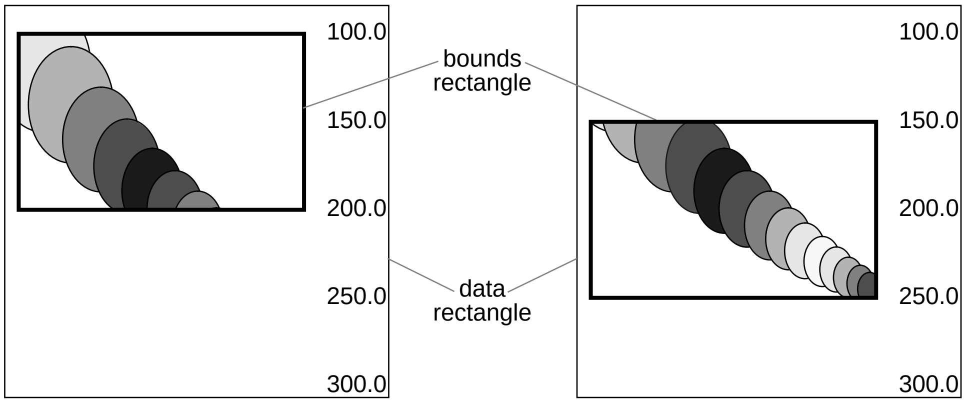
When a view moves to a new location in its parent, its frame rectangle changes but not its bounds rectangle. When a view scrolls its contents, its bounds rectangle changes, but not its frame. The frame rectangle positions the view in the world outside; the bounds rectangle positions the contents inside the view.

Since a BView does its work in its own coordinate system, it refers to the bounds rectangle more often than to the frame rectangle.

Scrolling

A BView scrolls its contents by shifting coordinate values within the view rectangle—that is, by altering the bounds rectangle. If, for example, the top of a view's bounds rectangle is at 100.0 and its bottom is at 200.0, scrolling downward 50.0 units would put the top at 150.0 and the bottom at 250.0. Contents of the view with *y* coordinate values of 150.0 to 200.0, originally displayed in the bottom half of the view, would be shifted to the top half.

Contents with *y* coordinate values from 200.0 to 250.0, previously unseen, would become visible at the bottom of the view. This is illustrated below:



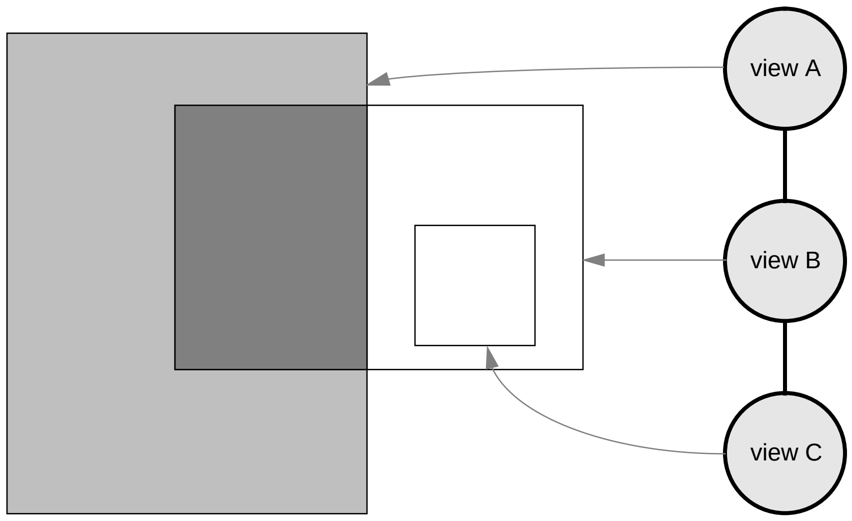
Scrolling doesn't move the view—it doesn't alter the frame rectangle—it moves only what's displayed inside the view. In the illustration above, a "data rectangle" encloses everything the BView is capable of drawing. For example, if the view is able to display an entire book, the data rectangle would be large enough to enclose all the lines and pages of the book laid end to end. However, since a BView can draw only within its bounds rectangle, everything in the data rectangle with coordinates that fall outside the bounds rectangle would be invisible. To make unseen data visible, the bounds rectangle must change the coordinates that it encompasses. Scrolling can be thought of as sliding the view's bounds rectangle to a new position on its data rectangle, as is shown in the illustration above. However, as it appears to the user, it's moving the data rectangle under the bounds rectangle. The view doesn't move; the data does.

The Clipping Region

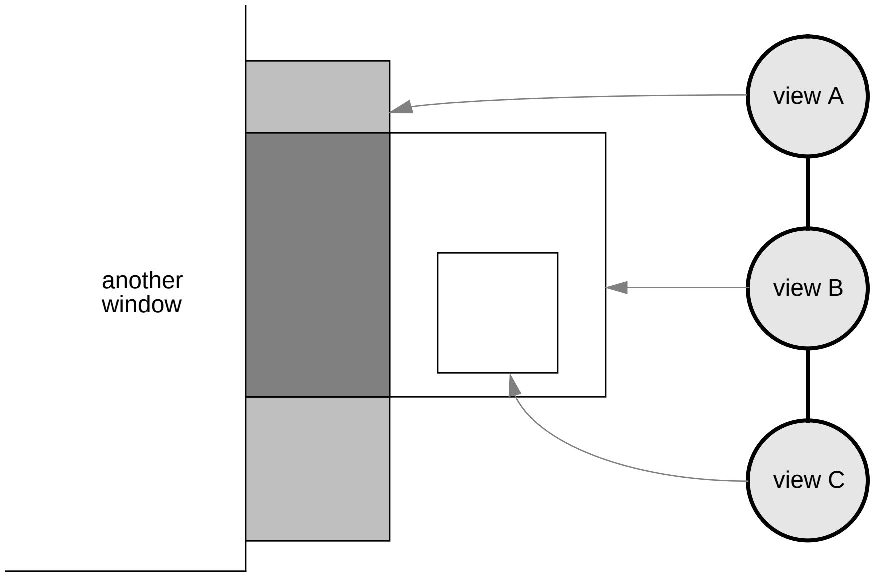
The Application Server clips the images that a BView produces to the region where it's permitted to draw.

This region is never any larger than the view's bounds rectangle; a view cannot draw outside its bounds. Furthermore, since a child is considered part of its parent, a view can't draw outside the bounds rectangle of its parent either—or, for that matter, outside the bounds rectangle of any ancestor view. In addition, since child views draw after, and therefore logically in front of, their parents, a view concedes some of its territory to its children.

Thus, the *visible region* of a view is the part of its bounds rectangle that's inside the bounds rectangles of all its ancestors, minus the frame rectangles of its children. This is illustrated in the figure below. It shows a hierarchy of three views. The area filled with a crosshatch pattern is the visible region of view *A*; it omits the area occupied by its child, view *B*. The visible region of view *B* is colored dark gray; it omits the part of the view that lies outside its parent. View *C* has no visible region, for it lies outside the bounds rectangle of its ancestor, view *A*:



The visible region of a view might be further restricted if its window is obscured by another window or if the window it's in lies partially off-screen. The visible region includes only those areas that are actually visible to the user. For example, if the three views in the illustration above were in a window that was partially blocked by another window, their visible regions might be considerably smaller. This is illustrated below:



Note that in this case, view *A* has a discontinuous visible region.

The Application Server clips the drawing that a view does to a region that's never any larger than the visible region. On occasion, it may be smaller. For the sake of efficiency, while a view is being automatically updated, the *clipping region* excludes portions of the visible region that don't need to be redrawn:

- When a view is scrolled, the Application Server may be able to shift some of its contents from one portion of the visible region to another. The clipping region excludes any part of the visible region that the Server was able to update on its own; it includes only the part where the BView must produce images that were not previously visible.

- If a view is resized larger, the clipping region may include only the new areas that were added to the visible region. (But see the *flags* argument for the BView constructor.)

- If only part of a view is invalidated (by the **Invalidate()** function), the clipping region is the intersection of the visible region and the invalid rectangle.

An application can also limit the clipping region for a view by passing a BRegion object to **ConstrainClippingRegion()**. The clipping region won't include any areas that aren't in the region passed. The Application Server calculates the clipping region as it normally would, but intersects it with the specified region.

You can obtain the current clipping region for a view by calling **GetClippingRegion()**. (See also the BRegion class description.)

The View Color

Every view has a basic, underlying color. It's the color that fills the view rectangle before the BView does any drawing. The user may catch a glimpse of this color when the view is first shown on-screen, when it's resized larger, and when it's erased in preparation for an update. It will also be seen wherever the BView fails to draw in the visible region.

In a sense, the view color is the canvas on which the BView draws. It doesn't enter into any of the object's drawing operations except to provide a background. Although it's one of the BView's graphics parameters, it's not one that any drawing functions refer to.

By default, the view color is white. You can assign a different color to a view by calling BView's **SetViewColor()** function. If you set the color to **B_TRANSPARENT_32_BIT**, the Application Server won't erase the view's clipping region before an update. This is appropriate only if the view erases itself by touching every pixel in the clipping region when it draws.

The Mechanics of Drawing

Views draw through a set of primitive functions such as:

- **DrawString()**, which draws a string of characters,
- **DrawBitmap()**, which produces an image from a bitmap,
- **DrawPicture()**, which executes a set of recorded drawing instructions,
- **StrokeLine()**, **StrokeArc()**, and other **Stroke...()** functions, which stroke lines along defined paths, and
- **FillEllipse()**, **FillRect()**, and other **Fill...()** functions, which fill closed shapes.

The way these functions work depends not only on the values that they're passed—the particular string, bitmap, arc, or ellipse that's to be drawn—but on previously set values in the BView's graphics environment.

Graphics Environment

Each BView object maintains its own graphics environment for drawing. The view color, coordinate system, and clipping region are fundamental parts of that environment, but not the only parts. It also includes a number of parameters that can be set and reset at will to affect the next image drawn. These parameters are:

- Font attributes that determine the appearance of text the BView draws. (See **SetFontName()** and its companion functions.)
- A symbol set that determines how character codes are mapped to visual symbols (glyphs). (See **SetSymbolSet()**.)
- Two pen parameters—a location and a size. The pen location determines where the next drawing will occur and the pen size determines the thickness of stroked lines. (See **MovePenBy()** and **SetPenSize()**.)
- Two current colors—a *high color* and a *low color*—that can be used either alone or in combination to form a pattern or halftone. The high color is used for most drawing. The low color is sometimes set to the underlying view color so that it can be used to erase other drawing or, because it matches the view background, make it appear that drawing has not touched certain pixels.

  (The high and low colors roughly match what other systems call the fore and back, or foreground and background, colors. However, neither color truly represents the color of the foreground or background. The terminology "high" and "low" is meant to keep the sense of two opposing colors and to match how they're defined in a pattern. A pattern bit is turned on for the high color and turned off for the low color. See the **SetHighColor()** and **SetLowColor()** functions and the "Patterns" section below.)

- A drawing mode that determines how the next image is to be rendered. (See the "Drawing Modes" section below and the **SetDrawingMode()** function.)

By default, a BView's graphics parameters are set to the following values:

Font	Kate (a 9-point bitmap font, no rotation, 90° shear)
Symbol Set	Macintosh
Pen position	(0.0, 0.0)
Pen size	1.0 coordinate units
High color	Black (red, green, and blue components all equal to 0)
Low color	White (red, green, and blue components all equal to 255)
Drawing mode	Copy mode (**B_OP_COPY**)
View color	White (red, green, and blue components all equal to 255)
Clipping region	The visible region of the view
Coordinate system	Origin at the left top corner of the bounds rectangle

However, as the next section, "Views and the Server" on page 31, explains, these values take effect only when the BView is assigned to a window.

The Pen

The pen is a fiction that encompasses two properties of a view's graphics environment: the current drawing location and the thickness of stroked lines.

The pen location determines where the next image will be drawn—but only if another location isn't explicitly passed to the drawing function. Some drawing functions alter the pen location—as if the pen actually moves as it does the drawing—but usually it's set by calling **MovePenBy()** or **MovePenTo()**.

The pen that draws lines (through the various **Stroke...()** functions) has a malleable tip that can be made broader or narrower by calling the **SetPenSize()** function. The larger the pen size, the thicker the line that it draws.

The pen size is expressed in coordinate units, which must be translated to a particular number of pixels for the display device. This is done by scaling the pen size to a device-specific value and rounding to the closest integer. For example, pen sizes of 2.6 and 3.3 would both translate to 3 pixels on-screen, but to 7 and 10 pixels respectively on a 300 dpi printer.

The size is never rounded to 0; no matter how small the pen may be, the line never disappears. If the pen size is set to 0.0, the line will be as thin as possible—it will be drawn using the fewest possible pixels on the display device. (In other words, it will be rounded to 1 for all devices.)

If the pen size translates to a tip that's broader than one pixel, the line is drawn with the tip centered on the path of the line. Roughly the same number of pixels are colored on both sides of the path.

A later section, "Picking Pixels to Stroke and Fill" on page 34, illustrates how pens of different sizes choose the pixels to be colored.

Colors

The high and low colors are specified as **rgb_color** values—full 32-bit values with separate red, green, and blue color components, plus an alpha component for transparency. Although there may sometimes be limitations on the colors that can be rendered on-screen, there are no restrictions on the colors that can be specified.

The way colors are specified for a bitmap depends on the color space in which they're interpreted. The color space determines the *depth* of the bitmap data (how many bits of information are stored for each pixel) and its *interpretation* (whether the data represents shades of gray or true colors, whether it's segmented into color components, what the components are, how they're arranged, and so on). Five possible color spaces are recognized:

B_MONOCHROME_1_BIT	One bit of data per pixel, where 1 is black and 0 is white.
B_GRAYSCALE_8_BIT	Eight bits of data per pixel, where a value of 255 is black and 0 is white.
B_COLOR_8_BIT	Eight bits of data per pixel, interpreted as an index into a list of 256 colors. The list is part of the system color map, and is the same for all applications.
B_RGB_16_BIT	< This color space is currently undefined. >
B_RGB_32_BIT	Four components of data per pixel—red, green, blue, and alpha—with eight bits per component. A component value of 255 yields the maximum amount of red, green, or blue, and a value of 0 indicates the absence of that color. < The alpha component is currently ignored. It will specify the coverage of the color—how transparent or opaque it is. > The components in the **B_RGB_32_BIT** color space are meshed rather than separated into distinct planes; all four components are specified for the first pixel before the four components for the second pixel, and so on. Unlike an **rgb_color**, the color components are arranged in reverse order—blue, green, red—followed by alpha. This is the natural order for many display devices.

The screen can be configured to display colors in either the **B_COLOR_8_BIT** color space or the **B_RGB_32_BIT** color space. When it's in the **B_COLOR_8_BIT** color space, specified

rgb_colors are displayed as the closest 8-bit color in the color list. (See the BBitmap class and the **system_colors()** global function.)

Patterns

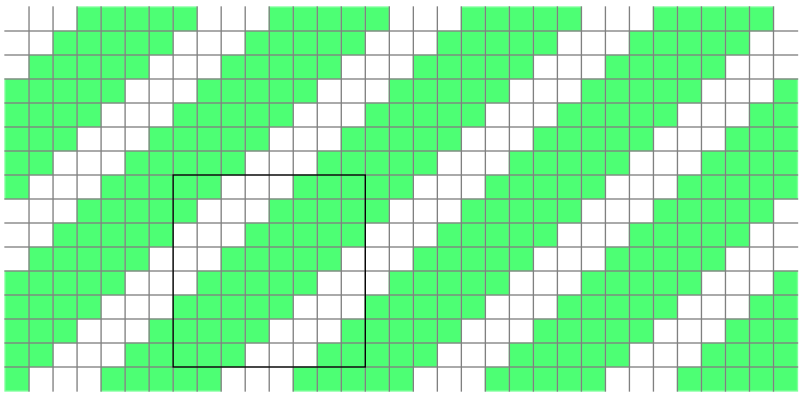
Functions that stroke a line or fill a closed shape don't draw directly in either the high or the low color. Rather they take a *pattern*, an arrangement of one or both colors that's repeated over the entire surface being drawn.

By combining the low color with the high color, patterns can produce dithered colors that lie somewhere between two hues in the **B_COLOR_8_BIT** color space. Patterns also permit drawing with less than the solid high color (for intermittent or broken lines, for example) and can take advantage of drawing modes that treat the low color as if it were transparent, as discussed below.

A pattern is defined as an 8-pixel by 8-pixel square. The **pattern** type is 8 bytes long, with one byte per row and one bit per pixel. Rows are specified from top to bottom and pixels from left to right. Bits marked 1 designate the high color; those marked 0 designate the low color. For example, a pattern of wide diagonal stripes could be defined as follows:

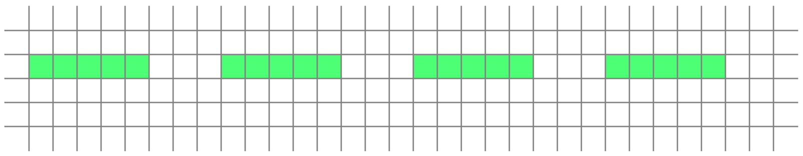
```
pattern stripes = { 0xc7, 0x8f, 0x1f, 0x3e,
                    0x7c, 0xf8, 0xf1, 0xe3 };
```

Patterns repeat themselves across the screen, like tiles that are laid side by side. The pattern defined above looks like this:



The dotted lines in this illustration show the separation of the screen into pixels. The thicker black line outlines one 8-by-8 square that the pattern defines.

The outline of the shape being filled or the width of the line being stroked determines where the pattern is revealed. It's as if the screen was covered with the pattern just below the surface, and stroking or filling allowed some of it to show through. For example, stroking a one-pixel wide horizontal path in the pattern illustrated above would result in a dotted line, with the dashes (in the high color) slightly longer than the spaces between (in the low color):



When stroking a line or filling a shape, the pattern serves as the source image for the current drawing mode, as explained under “Drawing Modes” below. The nature of the mode determines how the pattern interacts with the destination image, the image already in place.

The Interface Kit defines three patterns:

- **B_SOLID_HIGH** consists only of the high color,
- **B_SOLID_LOW** has only the low color, and
- **B_MIXED_COLORS** mixes the two colors evenly, like the pattern on a checkerboard.

B_SOLID_HIGH is the default pattern for all drawing functions. Applications can define as many other patterns as they need.

Drawing Modes

When a BView draws, it in effect transfers an image to a target location somewhere in the view rectangle. The drawing mode determines how the image being transferred interacts with the image already in place at that location. The image being transferred is known as the *source image*; it might be a bitmap or a pattern of some kind. The image already in place is known as the *destination image*.

In the simplest and most straightforward kind of drawing, the source image is simply painted on top of the destination; the source replaces the destination. However, there are other possibilities. There are nine different drawing modes—nine distinct ways of combining the source and destination images. The modes are designated by **drawing_mode** constants that can be passed to **SetDrawingMode()**:

B_OP_COPY	**B_OP_MIN**	**B_OP_ADD**
B_OP_OVER	**B_OP_MAX**	**B_OP_SUBTRACT**
B_OP_ERASE	**B_OP_INVERT**	**B_OP_BLEND**

B_OP_COPY is the default mode and the simplest. It transfers the source image to the destination, replacing whatever was there before. The destination is ignored.

In the other modes, however, some of the destination might be preserved, or the source and destination might be combined to form a result that’s different from either of them. For these modes, it’s convenient to think of the source image as an image that exists somewhere independent of the destination location, even though it’s not actually visible. It’s the image that would be rendered at the destination in **B_OP_COPY** mode.

The modes work for all BView drawing functions—including those that stroke lines and fill shapes, those that draw characters, and those that image bitmaps. The way they work depends foremost on the nature of the source image—whether it's a *pattern* or a *bitmap*. For the **Fill...()** and **Stroke...()** functions, the source image is a pattern that has the same shape as the area being filled or the area the pen touches as it strokes a line. For **DrawBitmap()**, the source image is a rectangular bitmap.

- Only a source pattern has designated "high" and "low" colors. Even if a source bitmap has colors that match the current high and low colors, they're not handled like the colors in a pattern; they're treated just like any other color in the bitmap.
- On the other hand, only a source bitmap can have transparent pixels. In the **B_COLOR_8_BIT** color space, a pixel is made transparent by assigning it the **B_TRANSPARENT_8_BIT** value. In the **B_RGB_32_BIT** color space, a pixel assigned the **B_TRANSPARENT_32_BIT** value is considered transparent. These values have meaning only for source bitmaps, not for source patterns. If the current high or low color in a pattern happens to have a transparent value, it's still treated as the high or low color, not like transparency in a bitmap.

The way the drawing modes work also depends on the color space of the source image and the color space of the destination. The following discussion concentrates on drawing where the source and destination both contain colors. This is the most common case, and also the one that's most general.

When applied to colors, the nine drawing modes fall naturally into four groups:

- The **B_OP_COPY** mode, which copies the source image to the destination.
- The **B_OP_OVER**, **B_OP_ERASE**, and **B_OP_INVERT** modes, which—despite their differences—all treat the low color in a pattern as if it were transparent.
- The **B_OP_ADD**, **B_OP_SUBTRACT**, and **B_OP_BLEND** modes, which combine colors in the source and destination images.
- The **B_OP_MIN** and **B_OP_MAX** modes, which choose between the source and destination colors.

The following paragraphs describe each of these groups in turn.

Copy Mode. In **B_OP_COPY** mode, the source image replaces the destination. This is the default drawing mode and the one most commonly used. Because this mode doesn't have to test for particular color values in the source image, look at the colors in the destination, or compute colors in the result, it's also the fastest of the modes.

If the source image contains transparent pixels, their transparency will be retained in the result; the transparent value is copied just like any other color. However, the appearance of a transparent pixel when shown on-screen is indeterminate. If a source image has transparent portions, it's best to transfer it to the screen in **B_OP_OVER** or another mode.

In all modes other than **B_OP_COPY**, a transparent pixel in a source bitmap preserves the color of the corresponding destination pixel.

Transparency Modes. Three drawing modes—**B_OP_OVER**, **B_OP_ERASE**, and **B_OP_INVERT**—are designed specifically to make use of transparency in the source image; they're able to preserve some of the destination image. In these modes (and only these modes) the low color in a source pattern acts just like transparency in a source bitmap.

- The **B_OP_OVER** mode places the source image "over" the destination; the source provides the foreground and the destination the background. In this mode, the source image replaces the destination image (just as in the **B_OP_COPY** mode)—except where a source bitmap has transparent pixels and a source pattern has the low color. Transparency in a bitmap and the low color in a pattern retain the destination image in the result.

  By masking out the unwanted parts of a rectangular bitmap with transparent pixels, this mode can place an irregularly shaped source image on top of a background image. Transparency in the source foreground lets the destination background show through. The versatility of **B_OP_OVER** makes it the second most commonly used mode, after **B_OP_COPY**.

- The **B_OP_ERASE** mode doesn't draw the source image at all. Instead, it erases the destination image. Like **B_OP_OVER**, it preserves the destination image wherever a source bitmap is transparent or a source pattern has the low color. But everywhere else—where the source bitmap isn't transparent and the source pattern has the high color—it removes the destination image, replacing it with the low color.

  Although this mode can be used for selective erasing, it's simpler to erase by filling an area with the **B_SOLID_LOW** pattern in **B_OP_COPY** mode.

- The **B_OP_INVERT** mode, like **B_OP_ERASE**, doesn't draw the source image. Instead, it inverts the colors in the destination image. As in the case of the **B_OP_OVER** and **B_OP_ERASE** modes, where a source bitmap is transparent or a source pattern has the low color, the destination image remains unchanged in the result. Everywhere else, the color of the destination image is inverted.

These three modes also work for monochrome images. If the source image is monochrome, the distinction between source bitmaps and source patterns breaks down. Two rules apply:

- If the source image is a monochrome bitmap, it acts just like a pattern. A value of 1 in the bitmap designates the current high color and a value of 0 designates the current low color. Thus, 0, rather than **B_TRANSPARENT_32_BIT** or **B_TRANSPARENT_8_BIT**, becomes the transparent value.

- If the source and destination are both monochrome, the high color is necessarily black (1) and the low color is necessarily white (0)—but otherwise the drawing modes work as described. With the possible colors this severely restricted, the three modes are reduced to boolean operations: **B_OP_OVER** is the same as a logical '*OR*',

B_OP_INVERT the same as logical '*exclusive OR*', and **B_OP_ERASE** the same as an inversion of logical '*AND*'.

Blending Modes. Three drawing modes—**B_OP_ADD**, **B_OP_SUBTRACT**, and **B_OP_BLEND**—combine the source and destination images, pixel by pixel, and color component by color component. As in most of the other modes, transparency in a source bitmap preserves the destination image in the result. Elsewhere, the result is a combination of the source and destination. The high and low colors of a source pattern aren't treated in any special way; they're handled just like other colors.

- **B_OP_ADD** adds each component of the source color to the corresponding component of the destination color, with a component value of 255 as the limit. Colors become brighter, closer to white.

  By adding a uniform gray to each pixel in the destination, for example, the whole destination image can be brightened by a constant amount.

- **B_OP_SUBTRACT** subtracts each component of the source color from the corresponding component of the destination color, with a component value of 0 as the limit. Colors become darker, closer to black.

  For example, by subtracting a uniform amount from the red component of each pixel in the destination, the whole image can be made less red.

- **B_OP_BLEND** averages each component of the source and destination colors (adds the source and destination components and divides by 2). The two images are merged into one.

These modes work only for color images, not for monochrome ones. If the source or destination is specified in the **B_COLOR_8_BIT** color space, the color will be expanded to a full **B_RGB_32_BIT** value to compute the result; the result is then contracted to the closest color in the **B_COLOR_8_BIT** color space.

Selection Modes. Two drawing modes—**B_OP_MAX** and **B_OP_MIN**—compare each pixel in the source image to the corresponding pixel in the destination image and select one to keep in the result. If the source pixel is transparent, both modes select the destination pixel. Otherwise, **B_OP_MIN** selects the darker of the two colors and **B_OP_MAX** selects the brighter of the two. If the source image is a uniform shade of gray, for example, **B_OP_MAX** would substitute that shade for every pixel in the destination image that was darker than the gray.

Like the blending modes, **B_OP_MIN** and **B_OP_MAX** work only for color images.

Views and the Server

Windows lead a dual life—as on-screen entities provided by the Application Server and as BWindow objects in the application. BViews have a similar dual existence—each BView object has a shadow counterpart in the Server. The Server knows the view's location, its place in the window's hierarchy, its visible area, and the current state of its graphics parameters. Because it has this information, the Server can more efficiently associate a user action with a particular view and interpret the BView's drawing instructions.

BWindows become known to the Application Server when they're constructed; creating a BWindow object causes the Server to produce the window that the user will eventually see on-screen. A BView, on the other hand, has no effect on the Server when it's constructed. It becomes known to the Server only when it's attached to a BWindow. The Server must look through the application's windows to see what views it has.

A BView that's not attached to a window therefore lacks a counterpart in the Server. This restricts what some functions can do. Four groups of functions are affected:

- Drawing functions—**DrawBitmap()**, **FillRect()**, **StrokeLine()**, and so on—don't work for unattached views. A BView can't draw unless it's in a window.

- The scrolling functions—**ScrollTo()** and **ScrollBy()**—require the BView to be in a window. Manipulations of a view's coordinate system are carried out in its Server counterpart.

- Functions that indirectly depend on a BView's graphics parameters—such as **GetMouse()**, which reports the cursor location in the BView's coordinates, and **StringWidth()**, which returns how much room a string would take up in the BView's font—also require the BView to belong to a window. These functions need information that an unattached BView can't provide.

- The functions that set and return graphics parameters—such as **SetDrawingMode()**, **PenLocation()**, **SetFontSize()**, and **SetHighColor()**—are also restricted. A view's graphic state is kept within the Server (where it's needed to carry out drawing instructions); BViews that the Server doesn't know about don't have a valid graphics state.

  Nevertheless, it's possible to assign a value to a graphics parameter before the BView is attached to a window. The value is simply cached until the view becomes part of a window's view hierarchy. It's then set as the current value for the parameter. Values set while the BView belongs to a window change the current value, but not the cached value. Therefore, if the BView is removed from the view hierarchy and reinstated as part of another hierarchy, the last cached value will be reestablished as the current value.

  Functions that return graphics parameters report the current value while the BView is attached to a window, and the cached value when it's unattached.

Because of these restrictions, you may find it difficult to complete the initialization of a BView at the time it's constructed. Instead, you may need to wait until the BView receives an **AttachedToWindow()** notification informing it that it has been added to a window's view hierarchy. This function is called for each view that's added to a window, beginning with the root view being attached, followed by each of its children, and so on down the hierarchy. After all views have been notified with an **AttachedToWindow()** function call, they each get an **AllAttached()** notification, but in the reverse order. A parent view that must adjust itself to calculations made by a child view when it's attached to a window can wait until **AllAttached()** to do the work.

These two function calls are matched by another pair—**DetachedFromWindow()** and **AllDetached()**—which notify BViews that they're about to be removed from the window.

The Update Mechanism

The Application Server sends a message to a BWindow whenever any of the views within the window need to be updated. The BWindow then calls the **Draw()** function of each out-of-date BView so that it can redraw the contents of its on-screen display.

Update messages can arrive at any time. A BWindow receives one whenever:

- The window is first placed on-screen, or is shown again after having been hidden.
- Any part of the window becomes visible after being obscured.
- The views in the window are rearranged—for example, if a view is resized or a child is removed from the hierarchy.
- Something happens to alter what a particular view displays. For example, if the contents of a view are scrolled, the BView must draw any new images that scrolling makes visible. If one of its children moves, it must fill in the area the child view vacated.
- The application forces an update by "invalidating" a view, or a portion of a view.

Update messages take precedence over other kinds of messages. To keep the on-screen display as closely synchronized with event handling as possible, the window acts on update messages as soon as they arrive. They don't need to wait their turn in the message queue.

(Update messages do their work quietly and behind the scenes. You won't find them in the BWindow's message queue, they aren't handled by BWindow's **DispatchMessage()** function, and they aren't returned by BLooper's **CurrentMessage()**.)

Forcing an Update

When a user action or a BView function alters a view in a window—for example, when a view is resized or its contents are scrolled—the Application Server knows about it. It makes sure that an update message is sent to the window so the view can be redrawn.

However, if code that's specific to your application alters a view, you'll need to inform the Server that the view needs updating. This is done by calling the **Invalidate()** function. For example, if you write a function that changes the number of elements a view displays, you might invalidate the view after making the change, as follows:

```
void MyView::SetNumElements(long count)
{
    if ( numElements == count )
        return;
    numElements = count;
    Invalidate();
}
```

Invalidate() ensures that the view's **Draw()** function—which presumably looks at the new value of the **numElements** data member—will be called automatically.

At times, the update mechanism may be too slow for your application. Update messages arrive just like other messages sent to a window thread, including the interface messages that report events. Although they take precedence over other messages, update messages must wait their turn. The window thread can respond to only one message at a time; it will get the update message only after it finishes with the current one.

Therefore, if your application alters a view and calls **Invalidate()** while responding to an interface message, the view won't be updated until the response is finished and the window thread is free to turn to the next message. Usually, this is soon enough. But if it's not, if the response to the interface message includes some time-consuming operations, the application can request an immediate update by calling BWindow's **UpdateIfNeeded()** function.

Erasing the Clipping Region

Just before sending an update message, the Application Server prepares the clipping region of each BView that is about to draw by erasing it to the view background color. Note that only the clipping region is erased, not the entire view, and perhaps not the entire area where the BView will, in fact, draw.

The Server foregoes this step only if the BView's background color is set to the magical **B_TRANSPARENT_32_BIT** color.

Drawing during an Update

While drawing, a BView may set and reset its graphics parameters any number of times—for example, the pen position and high color might be repeatedly reset so that whatever is drawn next is in the right place and has the right color. These settings are temporary. When the update is over, all graphics parameters are reset to their initial values.

If, for example, **Draw()** sets the high color to a shade of light blue, as shown below,

```
SetHighColor(152, 203, 255);
```

it doesn't mean that the high color will be blue when **Draw()** is called next. If this line of code is executed during an update, light blue would remain the high color only until the update ends or **SetHighColor()** is called again, whichever comes first. When the update ends, the previous graphics state, including the previous high color, is restored.

Although you can change most graphics parameters during an update—move the pen around, reset the font, change the high color, and so on—the coordinate system can't be touched; a view can't be scrolled while it's being updated. Since scrolling causes a view to be updated, scrolling during an update would, in effect, be an attempt to nest one update in another, something that can't logically be done (since updates happen sequentially through messages). If the view's coordinate system were to change, it would alter the current clipping region and confuse the update mechanism.

Drawing outside of an Update

Graphics parameters that are set outside the context of an update are not limited; they remain in effect until they're explicitly changed. For example, if application code calls **Draw()**, perhaps in response to an interface message, the parameter values that **Draw()** last sets would persist even after the function returns. They would become the default values for the view and would be assumed the next time **Draw()** is called.

Default graphics parameters are typically set as part of initializing the BView once it's attached to a window—in an **AttachedToWindow()** function. If you want a **Draw()** function to assume the values set by **AttachedToWindow()**, it's important to restore those values after any drawing the BView does that's not the result of an update. For example, if a BView invokes **SetHighColor()** while drawing in response to an interface message, it will need to restore the default high color when done.

If **Draw()** is called outside of an update, it can't assume that the clipping region will have been erased to the view color, nor can it assume that default graphics parameters will be restored when it's finished.

Picking Pixels to Stroke and Fill

This section discusses how the various BView **Stroke...()** and **Fill...()** functions pick specific pixels to color. Pixels are chosen after the pen size and all coordinate values have been translated to device-specific units. Device-specific values measure distances by counting pixels; one unit equals one pixel on the device.

A device-specific value can be derived from a coordinate value using a formula that takes the size of a coordinate unit and the resolution of the device into account. For example:

$$device\_value = coordinate\_value \times (dpi / 72)$$

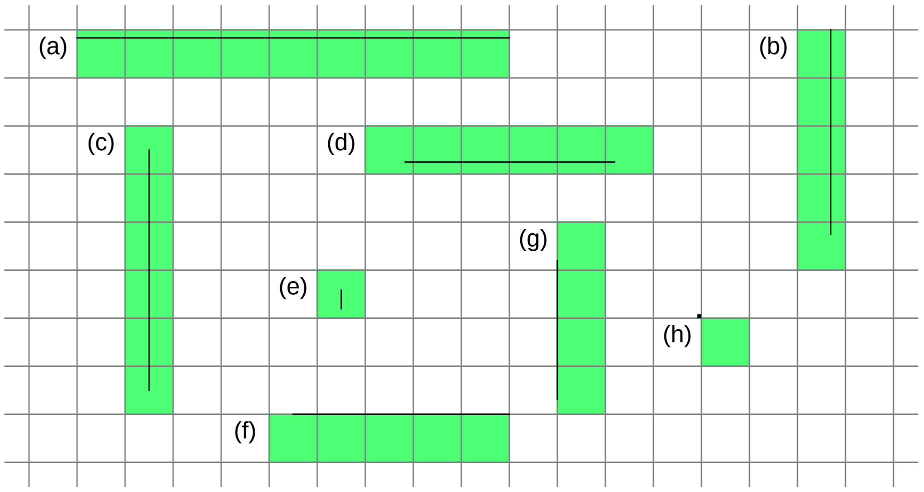
dpi is the resolution of the device in dots (pixels) per inch, 72 is the number of coordinate units in an inch, and *device_value* is rounded to the closest integer.

To describe where lines and shapes fall on the pixel grid, this section mostly talks about pixel units rather than coordinate units. The accompanying illustrations magnify the grid so that pixel boundaries are clear. As a consequence, they can show only very short lines and small shapes. By blowing up the image, they exaggerate the phenomena they illustrate.

Stroking Thin Lines

The thinnest possible line is drawn when the pen size translates to 1 pixel on the device. Setting the size to 0.0 coordinate units guarantees a one-pixel pen on all devices.

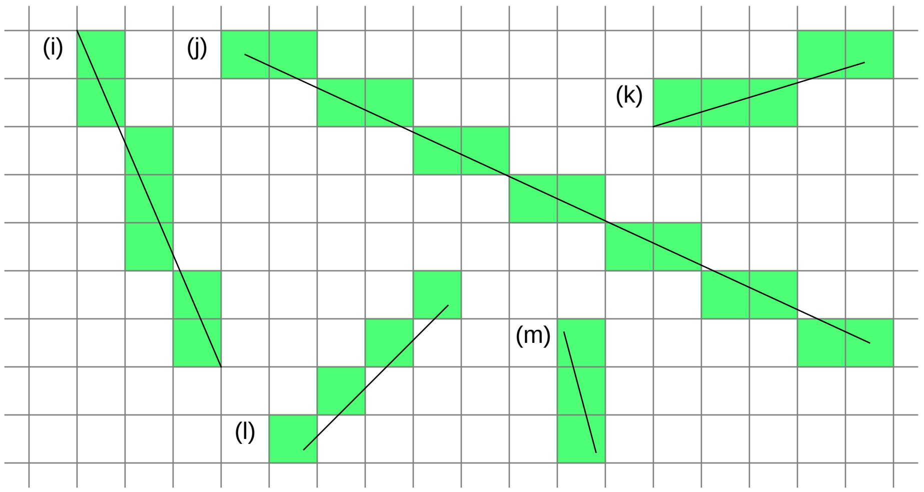
A one-pixel pen follows the path of the line it strokes and makes the line exactly one pixel thick at all points. If the line is perfectly horizontal or vertical, it touches just one row or one column of pixels, as illustrated below. (The grid of broken lines shows the separation of the display surface into pixels.)



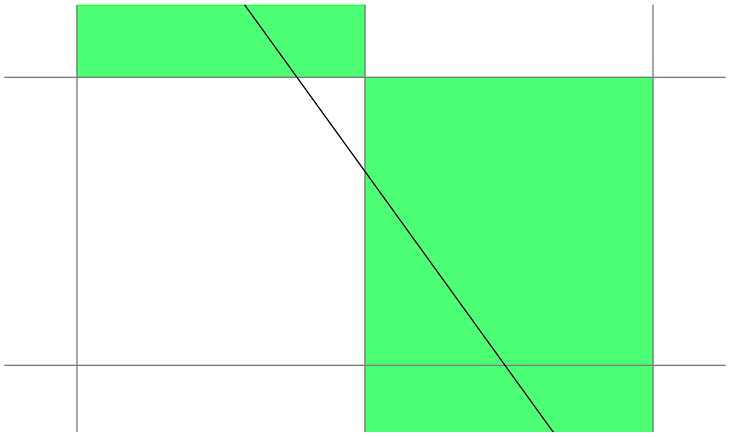
Only pixels that the line path actually passes through are colored to display the line. If a path begins or ends on a pixel boundary, as it does for examples (a) and (b) above, the pixels at the boundary aren't colored unless the path crosses into the pixel. The pen touches the fewest possible number of pixels.

A line path that doesn't enter any pixels, but lies entirely on the boundaries between pixels, colors the pixel row beneath it or the pixel column to its right, as illustrated by (f) and (g) above. A path that reduces to a single point lying on the corner of four pixels, as does (h) above, colors the pixel at its lower right. < However, currently, it's indeterminate which column or row of adjacent pixels would be used to display vertical and horizontal lines like (f) and (g) above. Point (h) would not be visible. >

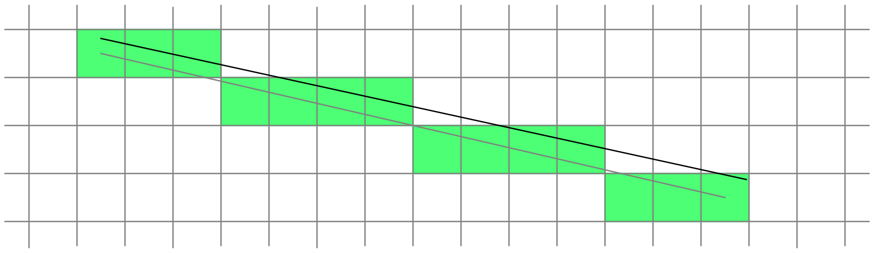
One-pixel lines that aren't exactly vertical or horizontal touch just one pixel per row or one per column. If the line is more vertical than horizontal, only one pixel in each row is used to color the line. If the line is more horizontal than vertical, only one pixel in each column is used. Some illustrations of slanted one-pixel thick lines are given below:



Although a one-pixel pen touches only pixels that lie on the path it strokes, it won't touch every pixel that the path crosses if that would mean making the line thicker than specified. When the path cuts though two pixels in a column or row, but only one of those pixels can be colored, the one that contains more of the path (the one that contains the midpoint of the segment cut by the column or row) is chosen. This is illustrated in the close-up below, which shows where a mostly vertical line crosses one row of pixels:



However, before a choice is made as to which pixel in a row or column to color, the line path is normalized for the device. For example, if a line is defined by two endpoints, it's first determined which pixels correspond to those endpoints. The line path is then treated as if it connected the centers of those pixels. This may alter which pixels get colored, as is illustrated below. In this illustration, the solid black line is the line path as originally specified and the broken line is its normalized version:



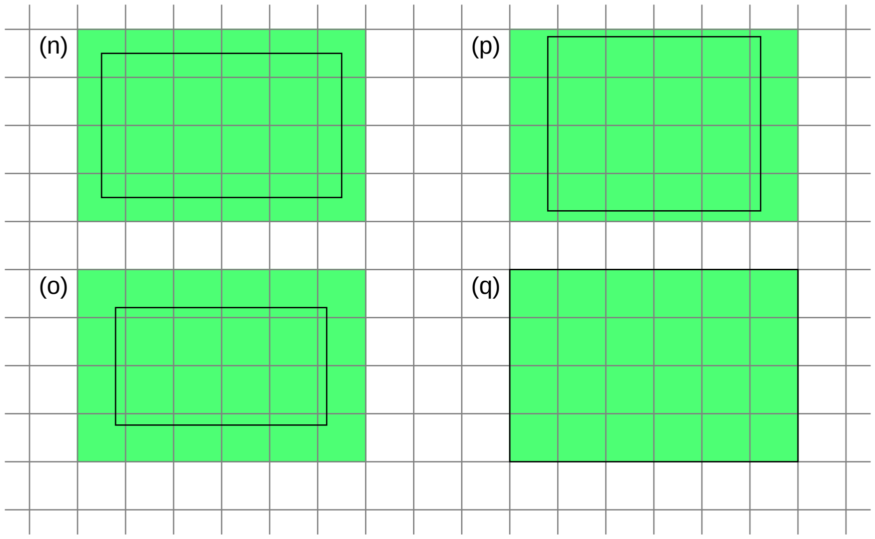
This normalization is nothing more than the natural consequence of the rounding that occurs when coordinate values are translated to device-specific pixel values.

Stroking Curved Lines

Although all the diagrams above show straight lines, the principles they illustrate apply equally to curved line paths. A curved path can be treated as if it were made up of a large number of short straight segments.

Filling and Stroking Rectangles

The following illustration shows how some rectangles, represented by the solid black line, would be filled with a solid color.

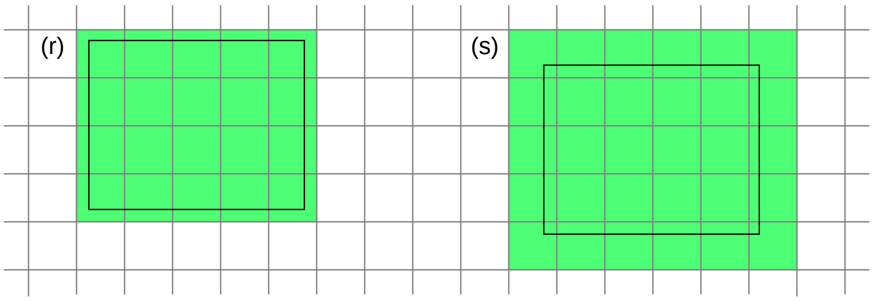


A rectangle includes every pixel that it encloses and every pixel that its sides pass through. However, as rectangle (q) illustrates, it doesn’t include pixels that its sides merely touch at the boundary.

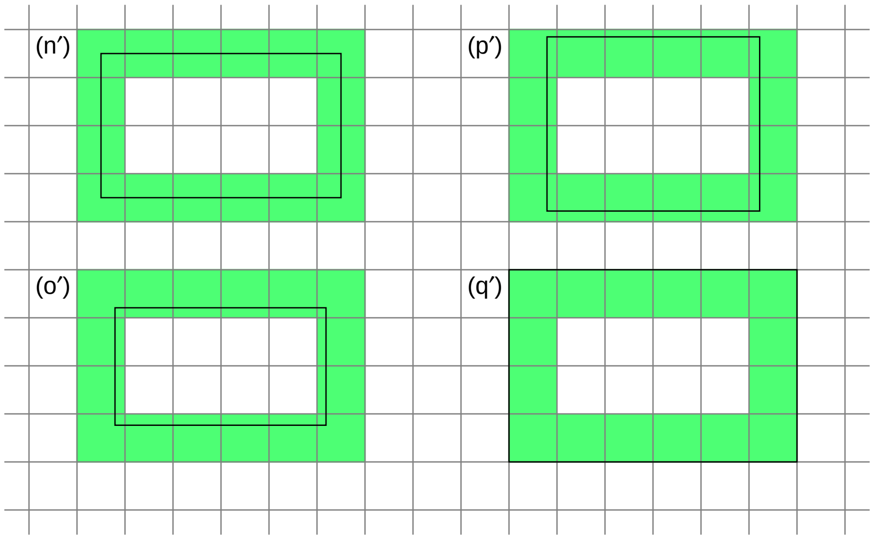
If the pixel grid in this illustration represents the screen, rectangle (q) would have left, top, right, and bottom coordinates with fractional values of .5. Rectangle (n), on the other hand, would have coordinates without any fractional parts. Nonfractional coordinates lie at the center of screen pixels.

Rectangle (n), in fact, is the normalized version of all four of the illustrated rectangles. It shows how the sides of the four rectangles would be translated to pixel values. Note that for a rectangle like (q), with edges that fall on pixel boundaries, normalization means rounding the left and top sides upward and rounding the right and bottom sides downward. This follows from the principal that the fewest possible number of pixels should be colored.

Although the four rectangles above differ in size and shape, when filled they all cover a 6 × 4 pixel area. You can't predict this area from the dimensions of the rectangle. Because the coordinate space is continuous and *x* and *y* values can be located anywhere, rectangles with different dimensions might have the same rendered size, as shown above, and rectangles with the same dimensions might have different rendered sizes, as shown below:



If a one-pixel pen strokes a rectangular path, it touches only pixels that would be included if the rectangle were filled. The illustration below shows the same rectangles that were presented above, but strokes them rather than fills them:

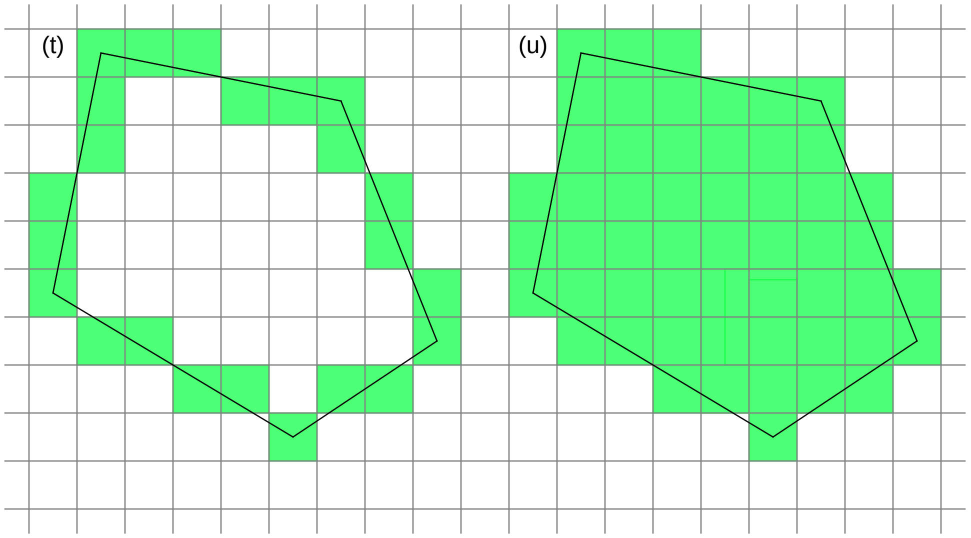


Each of the rectangles still covers a 6 × 4 pixel area. Note that even though the path of rectangle (q′) lies entirely on pixel boundaries, pixels below it and to its right are not touched by the pen. The pen touches only pixels that lie within the rectangle.

If a rectangle collapses to a straight line or to a single point, it no longer contains any area. Stroking or filling such a rectangle is equivalent to stroking the line path with a one-pixel pen, as was discussed in the previous section.

Filling and Stroking Polygons

The figure below shows a polygon as it would be stroked by a one-pixel pen and as it would be filled:

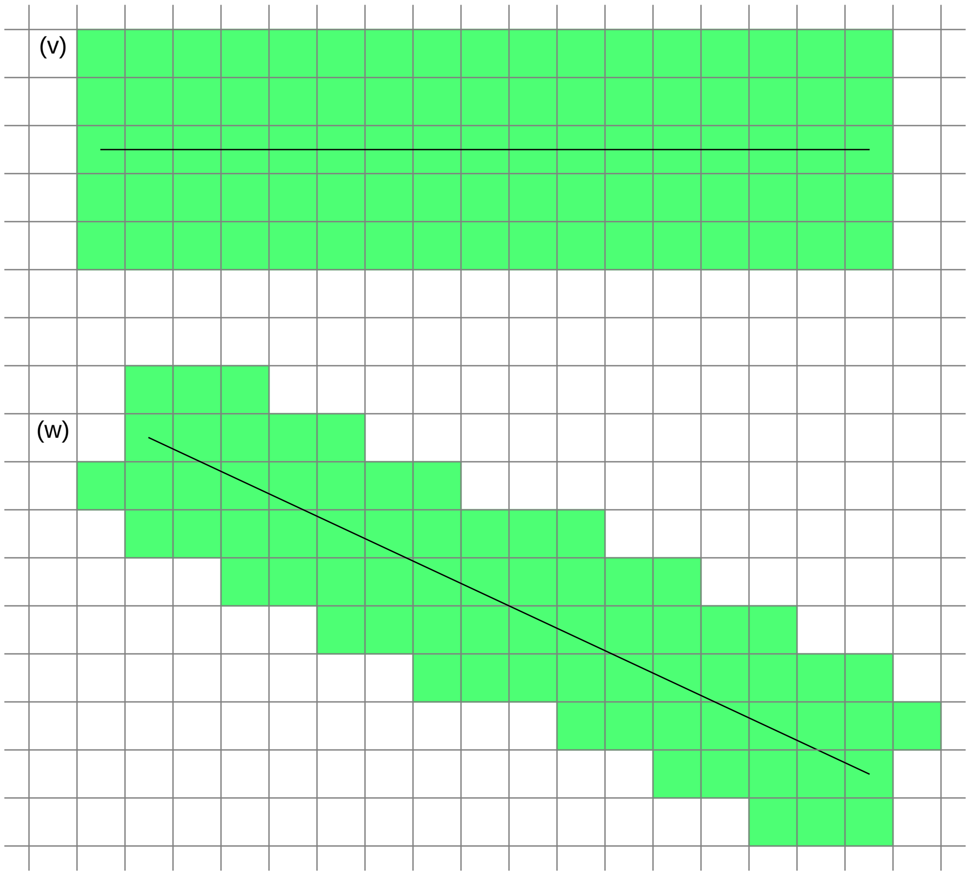


The same rules apply when stroking each segment of a polygon as would apply if that segment were an independent line. Therefore, the pen may not touch every pixel the segment passes through.

When the polygon is filled, no additional pixels around its border are colored. As is the case for a rectangle, the displayed shape of filled polygon is identical to the shape of the polygon when stroked with a one-pixel pen. The pen doesn't touch any pixels when stroking the polygon that aren't colored when the polygon is filled. Conversely, filling doesn't color any pixels at the border of the polygon that aren't touched by a one-pixel pen.

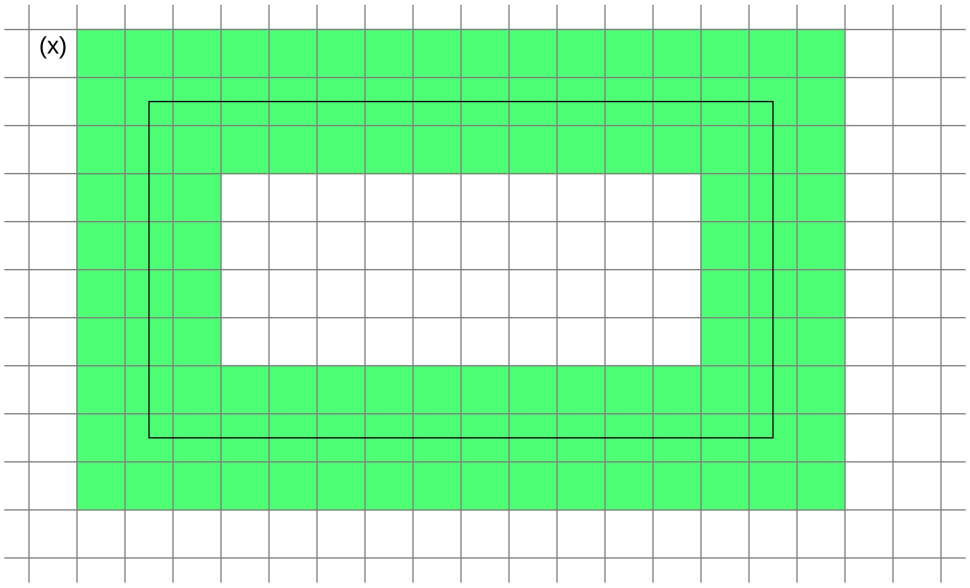
Stroking Thick Lines

A pen that's thicker than one pixel touches the same pixels that a one-pixel pen does, but it adds extra columns and rows adjacent to the line path. A thick pen tip is, in effect, a linear brush that's held perpendicular to the line path and kept centered on the line. The illustration below shows two short lines, each five pixels thick:



The thickness or a vertical or horizontal line can be measured in an exact number of pixels. When the line is slanted, as it is for (t) above, the stroking algorithm tries to make the line visually approximate the thickness of a vertical or horizontal line. In this way, lines retain their shape even when rotated.

When a rectangle is stroked with a thick pen, the corners of the rectangle are filled in, as shown in the example below:



Responding to the User

The BWindow and BView classes together define a structure for responding to user actions on the keyboard and mouse. These actions generate *interface messages* that are delivered to BWindow objects. The BWindow distributes responsibility for the messages it receives to other objects, typically BViews.

This section describes the messages that report user actions, and the way that BWindow and BView objects are structured to respond to them.

Interface Messages

Twenty interface messages are currently defined. Two of them command the window to do something in particular:

- A **B_ZOOM** instruction tells the window to zoom to a larger size—or to return to its normal size having previously been zoomed larger. The message is typically caused by the user operating the zoom button in the window's title tab.
- A **B_MINIMIZE** instruction tells the window to replace itself on-screen with a token representation—or to restore itself having been previously minimized. This message is typically caused by the user double-clicking the window tab (or the window token).

All other interface messages report *events*—something that happened, rather than something that the application must do. In most cases, the message merely reports what the user did on the keyboard or mouse. However, in some cases, the event may reflect the way the Application Server interpreted or handled a user action. The Server might respond directly to the user and pass along an message that indicates what it did—moved a window or changed a value, for example. In a few cases, the event may even reflect what the application thinks the user intended—that is, an application might interpret one or more generic user actions as a more specific event.

The following five messages report atomic user actions on the keyboard and mouse:

- A **B_KEY_DOWN** message reports a single key-down event. Key-down events occur when the user presses a character key on the keyboard. After the initial event (and a brief threshold), most keys generate repeated key-down events—as long as the user continues to hold the key down and doesn't press another key. Only character keys produce keyboard events. The modifier keys—Shift, Control, Caps Lock, and so on—don't produce events of any kind but may affect the character that's reported for another key.

- A **B_KEY_UP** message reports the event that occurs when the user releases the character key. < Although defined, this message is currently not used. Key-up events are unreported. >

- A **B_MOUSE_DOWN** message reports a single mouse-down event. A mouse-down event occurs when the user presses one of the mouse buttons while the cursor is over the content area of a window. The event is recognized (the message is generated) only for the first button the user presses—that is, only if no other mouse buttons are down at the time.

- A **B_MOUSE_UP** message reports the event that occurs when the user releases the mouse button. The event is recognized only for the last button the user releases—that is, only if no other mouse button remains down.

- A **B_MOUSE_MOVED** message captures some small portion of the cursor's movement into, within, or out of a window. If the cursor isn't over a window, it's movement isn't reported; it doesn't create mouse-moved events. (All interface events are associated with windows.) Repeated mouse-moved events occur as the user moves the mouse.

The five messages above are all directed at particular views—the view where the cursor is located or where typed input appears. Three others also concern views:

- A **B_VIEW_MOVED** message is sent when a view is moved within its parent's coordinate system. This can be a consequence of a programmatic action or of the parent view being automatically resized. If the parent view is being continuously resized because the user is resizing the window, repeated mouse-moved events may be reported.

- A **B_VIEW_RESIZED** message is delivered when a view is resized, perhaps because the program resized it or possibly as an automatic consequence of the window being resized. If the resizing is continuous, because the user is resizing the window, repeated view-resized events are reported.

- A **B_VALUE_CHANGED** message reports that the Application Server changed a value associated with an object. Currently, a value-changed event occurs only for BScrollBar objects. Repeated events are reported as the user manipulates a scroll bar.

A few messages concern events that affect the window itself:

- A **B_WINDOW_ACTIVATED** message reports an activation event. This event occurs when a window becomes the active window and again when it gives up that status. The single action of clicking a window to make it active might result in two activation events—one for the window that gains active-window status and one for the window that relinquishes it—plus a mouse-down and a mouse-up event.

- A **B_QUIT_REQUESTED** message is interpreted by a BWindow object as a request to close the window. Quit-requested events occur when the user clicks a window's close button, or when the system perceives some other reason to request the window to quit.

- A **B_WINDOW_MOVED** message records the new location of a window that has been moved, either programmatically or by the user. When the user drags a window, repeated messages are generated, each one capturing a small portion of the window's continuous movement. Only one window-moved event is reported when the program moves a window.

- A **B_WINDOW_RESIZED** message reports that a window has been resized, again either programmatically or by the user. The message is generated repeatedly as the user resizes the window, but only once each time the application resizes it.

A few messages report changes to the on-screen environment for a window:

- A **B_SCREEN_CHANGED** message reports that the configuration of the screen—the size of the pixel grid it displays or the color space of the frame buffer—has changed. Such changes may require the window to take compensatory measures.

- A **B_WORKSPACE_ACTIVATED** message reports that the active workspace (the one displayed on-screen) has changed. All windows that live in the previously active workspace and in the one that has been newly activated are notified of the change.

- A **B_WORKSPACES_CHANGED** message notifies the window that the set of workspaces in which it can be displayed has changed.

Two messages are produced by the save panel:

- A **B_SAVE_REQUESTED** message is sent when the user operates the panel to request that a document be saved.

- A **B_PANEL_CLOSED** message is sent when the application or the user closes the panel.

Finally, there's one message that doesn't derive from a user action:

- Periodic **B_PULSE** messages are posted at regularly spaced intervals, like a steady heartbeat. Pulses don't involve any communication between the application and the Server. They're generated as long as no other events are pending, but only if the application asks for them.

An application doesn't have to wait for a message to discover what the user is doing on the keyboard and mouse. Two BView functions, **GetKeys()** and **GetMouse()**, can provide an immediate check on the state of these devices.

Hook Functions for Interface Messages

Interface messages are generated and delivered to the application as the user acts. The Application Server determines which window an action affects and notifies the appropriate window thread. Messages for keyboard events are delivered to the current active window; messages announcing mouse events are sent to the window where the cursor is located.

However, the message is just an intermediary. As soon as it arrives, the BWindow dispatches it to initiate action within the window thread. Typically, one of the BViews associated with the window is asked to respond to the message—usually the BView that drew the image that elicited the user action. But some messages are handled by the BWindow itself.

Interface messages are dispatched by calling a virtual function that's matched to the message. If the message delivers an instruction, the function is named for the action that should be taken. For example, a zoom instruction is dispatched by calling the **Zoom()** function. If the message reports an event, the function is named for the event. For example, the BView where a mouse-down event occurs is notified with a **MouseDown()** function call. When the user clicks the close box of a window, generating a quit-requested event, the BWindow's **QuitRequested()** function is called.

The chart below lists the virtual functions that are called to initiate the application's response to interface messages, and the base classes where the functions are declared. Each application can implement these message-specific functions in a way that's appropriate to its purposes.

Message type	Virtual function	Class
B_ZOOM	**Zoom()**	BWindow
B_MINIMIZE	**Minimize()**	BWindow
B_KEY_DOWN	**KeyDown()**	BView
B_KEY_UP	*none*	
B_MOUSE_DOWN	**MouseDown()**	BView
B_MOUSE_UP	*none*	
B_MOUSE_MOVED	**MouseMoved()**	BView

B_VIEW_MOVED	**FrameMoved()**	BView
B_VIEW_RESIZED	**FrameResized()**	BView
B_VALUE_CHANGED	**ValueChanged()**	BScrollBar
B_WINDOW_ACTIVATED	**WindowActivated()**	BWindow and BView
B_QUIT_REQUESTED	**QuitRequested()**	BLooper
B_WINDOW_MOVED	**FrameMoved()**	BWindow
B_WINDOW_RESIZED	**FrameResized()**	BWindow
B_SCREEN_CHANGED	**ScreenChanged()**	BWindow
B_WORKSPACE_ACTIVATED	**WorkspaceActivated()**	BWindow
B_WORKSPACES_CHANGED	**WorkspacesChanged()**	BWindow
B_SAVE_REQUESTED	**SaveRequested()**	BWindow
B_PANEL_CLOSED	**SavePanelClosed()**	BWindow
B_PULSE	**Pulse()**	BView

< **B_KEY_UP** messages are currently not produced. > **B_MOUSE_UP** messages are produced, but they aren't dispatched by calling a virtual function. A BView can determine when a mouse button goes up by calling **GetMouse()** from within its **MouseDown()** function. As it reports information about the location of the cursor and the state of the mouse buttons, **GetMouse()** removes mouse messages from the BWindow's message queue, so the same information won't be reported twice.

A BWindow reinterprets a **B_QUIT_REQUESTED** message, originally defined for the BLooper class in the Application Kit, to mean a user request to close the window. However, it doesn't redeclare the **QuitRequested()** hook function that it inherits from BLooper.

Dispatching

Notice, from the chart above, that the BWindow class declares the functions that handle instructions and events directed at the window itself. **FrameMoved()** is called when the user moves the window, **FrameResized()** when the user resizes it, **WindowActivated()** when it becomes, or ceases to be, the active window, **Zoom()** when it should zoom larger, and so on.

Although the BWindow handles some interface messages, the most common ones—those reporting direct user actions on the keyboard and mouse—are handled by BViews. When the BWindow receives a keyboard or mouse message, it must decide which view is responsible.

This decision is relatively easy for messages reporting mouse events. The cursor points to the affected view. For example, when the user presses a mouse button, the BWindow calls the **MouseDown()** virtual function of the view under the cursor. When the user moves the mouse, it calls the **MouseMoved()** function of each view the cursor travels through.

However, there's no cursor attached to the keyboard, so the BWindow object must keep track of the view that's responsible for messages reporting key-down events. That view is known as the *focus view*.

The Focus View

The focus view is whatever view happens to be displaying the current selection (possibly an insertion point) within the window, or whatever check box, button, or other gadget is currently marked to show that it can be operated from the keyboard.

The focus view is expected to respond to the user's keyboard actions when the window is the active window. When the user presses a key on the keyboard, the BWindow calls the focus view's **KeyDown()** function. If the focus view displays editable data, it's also expected to handle commands that target the current selection, such as commands to cut, copy, or paste data.

The focus typically doesn't stay on one view all the time; it shifts from view to view. It may change as the user changes the current selection in the window—from text field to text field, for example. Or it changes when the user navigates from one view to another by pressing the Tab key. Only one view in the window can be in focus at a time.

Views put themselves in focus when they're selected by a user action of some kind. For example, when a BView's **MouseDown()** function is called, notifying it that the user has selected the view, it can grab the focus by calling **MakeFocus()**. When a BView makes itself the focus view, the previous focus view is notified that it has lost that status.

A view should become the focus view if:

- It has a **KeyDown()** function to display typed characters,
- It has a **KeyDown()** function so that the user can operate it from the keyboard, or
- It can show the current selection, whether or not it has a **KeyDown()** function.

A view should highlight the current selection only while it's in focus.

BViews make themselves the focus view (with the **MakeFocus()** function), but BWindows report which view is currently in focus (with the **CurrentFocus()** function).

Kinds of Keyboard Messages

The focus view gets most keyboard messages, but not all. Three kinds of **B_KEY_DOWN** messages are conscripted for special tasks:

- If the user holds a Command key down while pressing a character key, the Command-character combination is interpreted as a keyboard shortcut (typically for a menu item, but possibly for some other control device). Instead of assigning the message to a view, the BWindow tries to issue the command associated with the shortcut.

- If the user holds an Option key down while pressing the Tab key, the Option-Tab combination is interpreted as an instruction to change the focus view. Instead of assigning the message to a view, the BWindow forces the change. This is done to enable keyboard navigation in all circumstances.
- If the window has a default button and the user presses the Enter key, the window assigns the message to the button, so that it can respond to the key-down event as it would to a click. A "default button" is simply a button that can be operated from the Enter key on the keyboard.

In all other cases, the BWindow assigns the message to the current focus view.

Message Protocols

The BMessage objects that convey interface messages typically contain various kinds of data describing the events they report or clarifying the instructions they give. In most cases, the message contains more information than is passed to the function that starts the application's response. For example, a **MouseDown()** function is passed the point where the cursor was located when the user pressed the mouse button. But a **B_MOUSE_DOWN** BMessage also includes information about when the event occurred, what modifier keys the user was holding down at the time, which mouse button was pressed, whether the event counts as a solitary mouse-down, the second event of a double-click, or the third of a triple-click, and so on.

A **MouseDown()** function can get this information by taking it directly from the BMessage. The BMessage that the window thread is currently responding to can be obtained by calling the **CurrentMessage()** function, which the BWindow inherits from BLooper. For example, a **MouseDown()** function might check whether the event is a single-click or the second of a double-click as follows:

```
void MyView::MouseDown(BPoint point)
{
    long num = Window()->CurrentMessage()->FindLong("clicks");
    if ( num == 1 ) {
        . . .
    }
    else if ( num == 2 ) {
        . . .
    }
    . . .
}
```

The *Message Protocols* appendix lists the contents of all interface messages.

Keyboard Information

Most information about what the user is doing on the keyboard comes to applications by way of messages reporting key-down events. The application can usually determine what the user's intent was in pressing a key by looking at the character recorded in the message. But, as discussed under "**B_KEY_DOWN**" on page 7 of the *Message Protocols* appendix, the message carries other keyboard information in addition to the character—the key the user pressed, the modifier states that were in effect at the time, and the current state of all keys on the keyboard.

Some of this information can be obtained in the absence of key-down messages:

- The Interface Kit has a global **modifiers()** function that returns the current modifier states, and

- The BView class has a **GetKeys()** function that can provide the current state of all the keys and modifiers on the keyboard.

This section discusses in detail the kinds of information that you can get about the keyboard through interface messages and these functions.

Key Codes

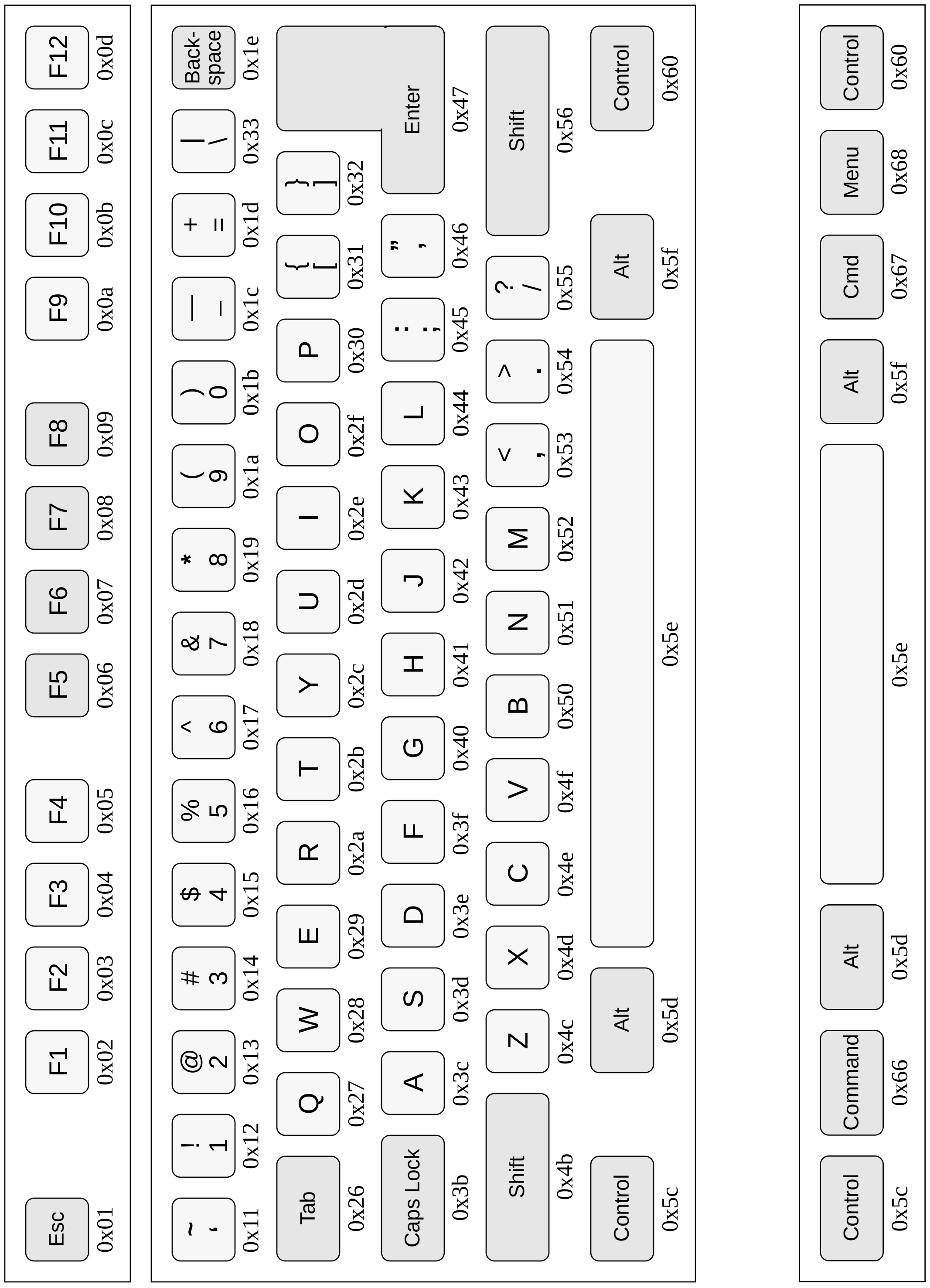
To talk about the keys on the keyboard, it's necessary first to have a standard way of identifying them. For this purpose, each key is arbitrarily assigned a numerical code.

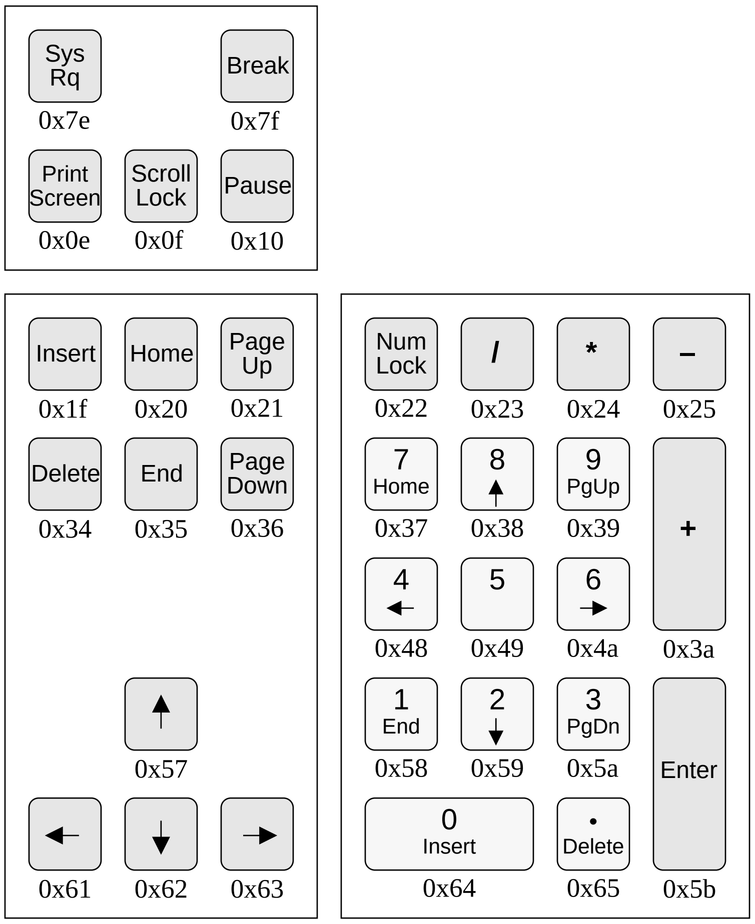
The illustrations on the next two pages show the key identifiers for a typical keyboard. The codes for the main keyboard are shown on page 49. This diagram shows a standard 101-key keyboard and an alternate version of the bottom row of keys—one that adds a Menu key and left and right Command keys.

The codes for the numerical keypad and for the keys between it and the main keyboard are shown on page 50.

Different keyboards locate keys in slightly different positions. The function keys may be to the left of the main keyboard, for example, rather than along the top. The backslash key (0x33) shows up in various places—sometimes above the Enter key, sometimes next to Shift, and sometimes in the top row (as shown here). No matter where these keys are located, they have the codes indicated in the illustrations.

The BMessage that reports a key-down event contains an entry named "key" for the code of the key that was pressed.


Esc 0x01
F1 0x02
F2 0x03
F3 0x04
F4 0x05
F5 0x06
F6 0x07
F7 0x08
F8 0x09
F9 0x0a
F10 0x0b
F11 0x0c
F12 0x0d
~ ` 0x11
! 1 0x12
@ 2 0x13
# 3 0x14
$ 4 0x15
% 5 0x16
^ 6 0x17
& 7 0x18
* 8 0x19
(9 0x1a
) 0 0x1b
— – 0x1c
+ = 0x1d
| \ 0x33
Back-space 0x1e
Tab 0x26
Q 0x27
W 0x28
E 0x29
R 0x2a
T 0x2b
Y 0x2c
U 0x2d
I 0x2e
O 0x2f
P 0x30
{ [0x31
}] 0x32
Enter 0x47
Caps Lock 0x3b
A 0x3c
S 0x3d
D 0x3e
F 0x3f
G 0x40
H 0x41
J 0x42
K 0x43
L 0x44
: ; 0x45
” ’ 0x46
Shift 0x4b
Z 0x4c
X 0x4d
C 0x4e
V 0x4f
B 0x50
N 0x51
M 0x52
< , 0x53
> . 0x54
? / 0x55
Shift 0x56
Control 0x5c
Alt 0x5d
0x5e
Alt 0x5f
Control 0x60
Control 0x5c
Command 0x66
Alt 0x5d
0x5e
Alt 0x5f
Cmd 0x67
Menu 0x68
Control 0x60



Kinds of Keys

Keys on the keyboard can be distinguished by the way they behave and by the kinds of information they provide. A principal distinction is between *character keys* and *modifier keys*:

- *Character keys* are mapped to particular characters; they generate key-down events when pressed. Keys not mapped to characters don't generate events.
- *Modifier keys* set states that can be discerned independently of key-down events (through the **modifiers()** function). Some modifier keys—like Caps Lock and Num Lock—toggle in and out of a locked modifier state. Others—like Shift and Control—set the state only while the key is being held down.

If a key doesn't fall into one of these categories or the other, there's nothing for it to do; it has no role to play in the interface. For most keys, the categories are mutually exclusive. Modifier keys are typically not mapped to characters, and character keys don't set modifier states. However, the Scroll Lock key is an exception. It both sets a modifier state and generates a character.

Keys can be distinguished on two other grounds as well:

- *Repeating keys* produce a continuous series of key-down events, as long as the user holds the key down and doesn't press another key. After the initial event, there's a slight delay before the key begins repeating, but then events are generated in rapid succession.

  All keys are repeating keys except for Pause, Break, and the three that set locks (Caps Lock, Num Lock, and Scroll Lock). Even modifier keys like Shift and Control would repeat if they were mapped to characters (but, since they're not, they don't produce any key-down events at all).

- *Dead keys* are keys that don't produce characters until the user strikes another key (or the key repeats). If the key the user strikes after the dead key belongs to a particular set, the two keys together produce one character (one key-down event). If not, each produces a separate character. The key-down event for the dead key is delayed until it can be determined whether it will be combined with another key to produce just one event.

  Dead keys are dead only when the Option key is held down. They're most appropriate for situations where the user can imagine a character being composed of two distinguishable parts—such as 'a' and 'e' combining to form 'æ'.

  The system permits up to five dead keys. By default, they're reserved for combining diacritical marks with other characters. The diacritical marks are the acute (´) and grave (`) accents, dieresis (¨), circumflex (ˆ), and tilde (˜).

There's a system key map that determines the role that each key plays—whether it's a character key or a modifier key, which modifier states it sets, which characters it produces, whether it's dead or not, how it combines with other keys, and so on. The map is shared by all applications.

Users can modify the key map with the Keyboard utility. Applications can look at it (and perhaps modify it) by calling the **system_key_map()** global function. See that function on page 327 for details on the structure of the map. The discussion here assumes the default key map that comes with the computer.

Modifier Keys

The role of a modifier key is to set a temporary, modal state. There are eight modifier states—eight different kinds of modifier key—defined functionally. Three of them affect the character that's reported in a key-down event:

- The *Shift key* maps alphabetic keys to the uppercase version of the character, and other keys to alternative symbols.

- The *Control key* maps alphabetic keys to Control characters—those with ASCII values (character codes) below 0x20.

- The *Option key* maps keys to alternative characters, typically characters in an extended set—those with ASCII values above 0x7f.

Two modifier keys permit users to give the application instructions from the keyboard:

- When the *Command key* is held down, the character keys perform keyboard shortcuts.
- The *Menu key* initiates keyboard navigation of menus. Pressing and releasing a Command key (without touching another key) accomplishes the same thing.

Three modifiers toggle in and out of locked states:

- The *Caps Lock* key reverses the effect of the Shift key for alphabetic characters. With Caps Lock on, the uppercase version of the character is produced without the Shift key, and the lowercase version with the Shift key.
- The *Num Lock* key similarly reverses the effect of the Shift key for keys on the numeric keypad.
- The *Scroll Lock* key temporarily prevents the display from updating. (It's up to applications to implement this behavior.)

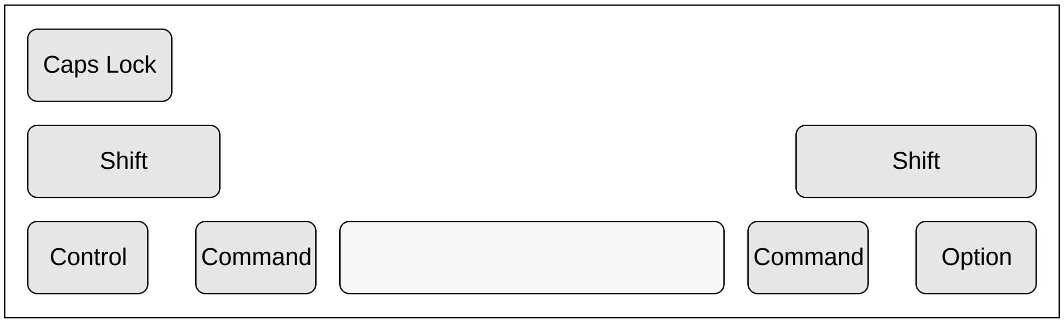
There are two things to note about these eight modifier states. First, since applications can read the modifiers directly from the messages that report key-down events and obtain them at other times by calling the **modifiers()** and **GetKeys()** functions, they are free to interpret the modifier states in any way they desire. They're not tied to the narrow interpretation of, say, the Control key given above. Control, Option, and Shift, for example, often modify the meaning of a mouse event or are used to set other temporary modes of behavior.

Second, the set of modifier states listed above doesn't quite match the keys that are marked on a typical keyboard. A standard 101-key keyboard has left and right "Alt(ernate)" keys, but lacks those labeled "Command," "Option," or "Menu."

The key map must, therefore, bend the standard keyboard to the required modifier states. The default key map does this in three ways:

- Because the "Alt(ernate)" keys are close to the space bar and are easily accessible, the default key map assigns them the role of Command keys.
- It turns the right "Control" key into an Option key. Therefore, there's just one functional Control key (on the left) and one Option key (on the right).
- It leaves the Menu key unmapped. It relies on the Command key as an adequate alternative for initiating keyboard navigation of menus.

The illustration below shows the modifier keys on the main keyboard, with labels that match their functional roles. Users can, of course, remap these keys with the Keyboard utility. Applications can remap them by calling **set_modifier_key()** or **system_key_map()**.



Current modifier states are reported in a mask that can be tested against these constants:

B_SHIFT_KEY	**B_COMMAND_KEY**	**B_CAPS_LOCK**
B_CONTROL_KEY	**B_MENU_KEY**	**B_NUM_LOCK**
B_OPTION_KEY		**B_SCROLL_LOCK**

The ...**_KEY** modifiers are set if the user is holding the key down. The ...**_LOCK** modifiers are set only if the lock is on—regardless of whether the key that sets the lock happens to be up or down at the time.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be more specifically tested against these constants:

B_LEFT_SHIFT_KEY	**B_RIGHT_SHIFT_KEY**
B_LEFT_CONTROL_KEY	**B_RIGHT_CONTROL_KEY**
B_LEFT_OPTION_KEY	**B_RIGHT_OPTION_KEY**
B_LEFT_COMMAND_KEY	**B_RIGHT_COMMAND_KEY**

If no keyboard locks are on and the user isn't holding a modifier key down, the modifiers mask will be 0.

The modifiers mask is returned by the **modifiers()** function and, along with other keyboard information, by BView's **GetKeys()**. It's also included as a "modifiers" entry in every BMessage that reports a keyboard or mouse event.

Character Mapping

Most keys are mapped to more than one character. The precise character that the key produces depends on which modifier keys are being held down and which lock states the keyboard is in at the time the key is pressed.

A few examples are given in the table below:

Key	No modifiers	Shift alone	Option alone	Shift & Option	Control
0x15	'4'	'$'	'¢'		'4'
0x18	'7'	'&'	'¶'	'§'	'7'
0x26	**B_TAB**	**B_TAB**	**B_TAB**	**B_TAB**	**B_TAB**
0x2e	'i'	'I'			**B_TAB**
0x40	'g'	'G'	'©'		0x07
0x43	'k'	'K'	'◊'		**B_PAGE_UP**
0x51	'n'	'N'	'ñ'	'Ñ'	0x0e
0x55	'/'	'?'	'÷'	'¿'	'/'
0x64	**B_INSERT**	'0'	**B_INSERT**	'0'	**B_INSERT**

The mapping follows some fixed rules, including these:

- If a Command key is held down, the Control keys are ignored. Command trumps Control. Otherwise, Command doesn't affect the character that's reported for the key. If only Command is held down, the character that's reported is the same as if no modifiers were down; if Command and Option are held down, the character that's reported is the same as for Option alone; and so on.

- If a Control key is held down (without a Command key), Shift, Option, and all keyboard locks are ignored. Control trumps the other modifiers (except for Command).

- Num Lock applies only to keys on the numerical keypad. While this lock is on, the effect of the Shift key is inverted. Num Lock alone yields the same character that's produced when a Shift key is down (and Num Lock is off). Num Lock plus Shift yields the same character that's produced without either Shift or the lock.

- Menu and Scroll Lock play no role in determining how keys are mapped to characters.

The default key map also follows the conventional rules for Caps Lock and Control:

- Caps Lock applies only to the 26 alphabetic keys on the main keyboard. It serves to map the key to the same character as Shift. Using Shift while the lock is on undoes the effect of the lock; the character that's reported is the same as if neither Shift nor Caps Lock applied. For example, Shift-*G* and Caps Lock-*G* both are mapped to uppercase 'G', but Shift-Caps Lock-*G* is mapped to lowercase 'g'.

  However, if the lock doesn't affect the character, Shift plus the lock is the same as Shift alone. For example, Caps Lock-*7* produces '7' (the lock is ignored) and Shift-*7* produces '&' (Shift has an effect), so Shift-Caps Lock-*7* also produces '&' (only Shift has an effect).

- When Control is used with a key that otherwise produces an alphabetic character, the character that's reported has an ASCII value 0x40 less than the value of the uppercase version of the character (0x60 less than the lowercase version of the character). This often results in a character that is produced independently by another key. For example, Control-*I* produces the **B_TAB** character and Control-*L* produces **B_PAGE_DOWN**.

When Control is used with a key that doesn't produce an alphabetic character, the character that's reported is the same as if no modifiers were on. For example, Control-*7* produces a '7'.

The Interface Kit defines constants for characters that aren't normally represented by a visible symbol. This includes the usual space and backspace characters, but most invisible characters are produced by the function keys and the navigation keys located between the main keyboard and the numeric keypad. The character values associated with these keys are more or less arbitrary, so you should always use the constant in your code rather than the actual character value. Many of these characters are also produced by alphabetic keys when a Control key is held down.

The table below lists all the character constants defined in the Kit and the keys they're associated with.

Key label	Key code	Character reported
Backspace	0x1e	**B_BACKSPACE**
Tab	0x26	**B_TAB**
Enter	0x47	**B_ENTER**
(space bar)	0x5e	**B_SPACE**
Escape	0x01	**B_ESCAPE**
F1 – F12	0x02 through 0x0d	**B_FUNCTION_KEY**
Print Screen	0x0e	**B_FUNCTION_KEY**
Scroll Lock	0x0f	**B_FUNCTION_KEY**
Pause	0x10	**B_FUNCTION_KEY**
System Request	0x7e	0xc8
Break	0x7f	0xca
Insert	0x1f	**B_INSERT**
Home	0x20	**B_HOME**
Page Up	0x21	**B_PAGE_UP**
Delete	0x34	**B_DELETE**
End	0x35	**B_END**
Page Down	0x36	**B_PAGE_DOWN**
(up arrow)	0x57	**B_UP_ARROW**
(left arrow)	0x61	**B_LEFT_ARROW**
(down arrow)	0x62	**B_DOWN_ARROW**
(right arrow)	0x63	**B_RIGHT_ARROW**

Several keys are mapped to the **B_FUNCTION_KEY** character. An application can determine which function key was pressed to produce the character by testing the key code against these constants:

B_F1_KEY	**B_F6_KEY**	**B_F11_KEY**
B_F2_KEY	**B_F7_KEY**	**B_F12_KEY**
B_F3_KEY	**B_F8_KEY**	**B_PRINT_KEY** (the "Print Screen" key)
B_F4_KEY	**B_F9_KEY**	**B_SCROLL_KEY** (the "Scroll Lock" key)
B_F5_KEY	**B_F10_KEY**	**B_PAUSE_KEY**

Note that key 0x30 (*P*) is also mapped to **B_FUNCTION_KEY** when the Control key is held down.

Key States

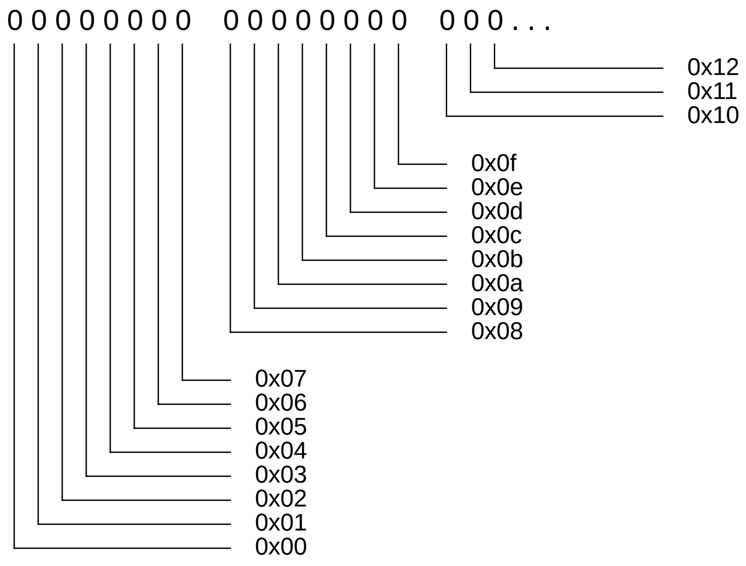
You can look at the state of all keys on the keyboard at a given moment in time. This information is captured and reported in two ways:

- As the "states" entry in every **B_KEY_DOWN** message, and
- As the **key_states** bitfield reported by BView's **GetKeys()** function.

In both cases, the bitfield is an array of 16 bytes,

```
uchar states[16];
```

with one bit standing for each key on the keyboard. Bits are numbered from left to right, beginning with the first byte in the array, as illustrated below:



Bit numbers start with 0 and match key codes. For example, bit 0x3c corresponds to the *A* key, 0x3d to the *S* key, 0x3e to the *D* key, and so on. The first bit is 0x00, which doesn't correspond to any key. The first meaningful bit is 0x01, which corresponds to the Escape key.

When a key is down, the bit corresponding to its key code is set to 1. Otherwise, the bit is set to 0. However, for the three keys that toggle keyboard locks—Caps Lock (key 0x3b), Num Lock (key 0x22), and Scroll Lock (key 0x0f)—the bit is set to 1 if the lock is on and set to 0 if the lock is off, regardless of the state of the key itself.

To test the bitfield against a particular key,

- Select the byte in the **states** array that contains the bit for that key,
- Form a mask for the key that can be compared to that byte, and
- Compare the byte to the mask.

For example:

```
if ( states[keyCode>>3] & (1 << (7 - (keyCode%8))) )
    . . .
```

Here, the key code is divided by 8 to obtain an index into the **states** array. This selects the byte (the **uchar**) in the array that contains the bit for that key. Then, the part of the key code that remains after dividing by 8 is used to calculate how far a bit needs to be shifted to the left so that it's in the same position as the bit corresponding to the key. This mask is compared to the **states** byte with the bitwise **&** operator.

Class Descriptions

The classes in the Interface Kit work together to define a program structure for drawing and responding to the user. The two classes at the core of the structure—BWindow and BView—have been discussed extensively above. Other Kit classes either derive from BWindow and BView or support the work of those that do. The Kit defines several different kinds of BViews that you can use in your application. But every application does some unique drawing and has some application-specific responses to messages, so it must also invent some BViews of its own.

To learn about the Interface Kit for the first time, it's recommended that you first read this introduction, then look at the BView and BWindow class descriptions, followed by the descriptions of other classes as they interest you. It also might be useful to look at supporting classes—like BPoint and BRect—early.

The class overview should help you determine which specific functions you need to turn to in order to get more information about a class. The class constructor is often a good place to start, as it contains general information on how instances of the class are initialized.

If you haven't already read about the BApplication object and the messaging classes in the Application Kit, be sure to do so. A program must have a BApplication object before it can use the Interface Kit.

A reference to the Interface Kit follows. The classes are presented in alphabetical order, beginning with BAlert.

BAlert

Derived from:	public BWindow
Declared in:	<interface/Alert.h>

Overview

A BAlert places a modal window on-screen in front of other windows and keeps it there until the user dismisses it. The window is an *alert panel* that has a message for the user to read and one or more buttons along the bottom that present various options for the user to choose among. Operating a button with the keyboard or mouse selects a course of action and dismisses the panel (closes the window). The message in the alert panel might warn the user of something or convey some information that the application doesn't want the user to overlook. Typically, it asks a question that the user must answer (by operating the appropriate button).

The alert panel stays on-screen only temporarily, until the user operates one of the buttons. As long as it's on-screen, other parts of the application's user interface are disabled. However, the user can continue to move windows around and work in other applications.

It's possible to design such a panel using a BWindow object, some BButtons, and other views. However, the BAlert class provides a simple way to do it. There's no need to construct views and arrange them, or call functions to show the window and then get rid of it. All you do is:

- Construct the object.
- Call **SetShortcut()** if you want the user to be able to operate window buttons from the keyboard. (The button on the right is automatically made the default button and can be operated by the Enter key.)
- Call **Go()** to put the window on-screen.

For example:

```
BAlert *alert;
long result;

alert = new BAlert("", "Time’s up!  Do you want to continue?",
                       "Cancel", "Continue", NULL,
                       B_WIDTH_FROM_WIDEST, B_WARNING_ALERT);
alert->SetShortcut(0, B_ESCAPE);
result = alert->Go();
```

Go() doesn't return until the user operates a button to dismiss the panel. When it returns, the window will have been closed, the window thread will have been killed, and the BAlert object will have been deleted.

The value **Go()** returns indicates which button dismissed the panel. If the user clicked the "Cancel" button in this example or pressed the Escape key, the return result would be 0. If the user clicked "Continue", the result would be 1. Since the BAlert sets up the rightmost button as the default button for the window, the user could also operate the "Continue" button by pressing the Enter key.

Constructor

BAlert()

BAlert(const char **title*, const char **text*,
const char **firstButton*,
const char **secondButton* = NULL,
const char **thirdButton* = NULL,
button_width *width* = B_WIDTH_AS_USUAL,
alert_type *type* = B_INFO_ALERT**)**

Creates an alert panel as a modal window. The window displays some *text* for the user to read, and can have up to three buttons. There must be at least a *firstButton*; the others are optional. The BAlert must also have a *title*, even though the panel doesn't have a title tab to display it. The title can be **NULL** or an empty string.

The buttons are arranged in a row at the bottom of the panel so that one is always in the right bottom corner. They're placed from left to right in the order specified to the constructor. If labels for three buttons are provided, *firstButton* will be on the left, *secondButton* in the middle, and *thirdButton* on the right. If only two labels are provided, *firstButton* will come first and *secondButton* will be in the right bottom corner. If there's just one label (*firstButton*), it will be at the right bottom location.

By default, the user can operate the rightmost button by pressing the Enter key. If a "Cancel" button is included, it should be assigned the **B_ESCAPE** character as a keyboard shortcut. Other buttons can be assigned other shortcut characters. Use BAlert's **SetShortcut()** function to set up the shortcuts, rather than BWindow's **AddShortcut()**. Shortcuts added by a BWindow require the user to hold down a Command key, while those set by a BAlert don't.

By default, all the buttons have a standard, minimal width (**B_WIDTH_AS_USUAL**). This is adequate for most buttons, but may not be wide enough to accommodate an especially long label. To let the width of each button adjust to the width of its label, set the *width* parameter to **B_WIDTH_FROM_LABEL**. To ensure that the buttons are all the same width, yet wide enough to display the widest label, set the *width* parameter to **B_WIDTH_FROM_WIDEST**.

For more hands-on manipulation of the buttons, you can get the BButton objects that the BAlert creates by calling the **ButtonAt()** function. To get the BTextView object that displays the *text* string, you can call **TextView()**.

There are various kinds of alert panels, depending on the content of the textual message and the nature of the options presented to the user. The *type* parameter should classify the BAlert object as one of the following:

B_EMPTY_ALERT
B_INFO_ALERT
B_IDEA_ALERT
B_WARNING_ALERT
B_STOP_ALERT

Currently, the alert *type* is used only to select a representative icon that's displayed at the left top corner of the window. A **B_EMPTY_ALERT** doesn't have an icon.

After the BAlert is constructed, **Go()** must be called to place it on-screen. Before returning, **Go()** destroys the object. You don't need to write code to delete it.

See also: **Go()**, **SetShortcut()**

Member Functions

ButtonAt()

inline BButton ***ButtonAt(**long *index***)** const

Returns a pointer to the BButton object for the button at *index*. Indices begin at 0 and count buttons from left to right. The BButton belongs to the BAlert object and should not be freed.

See also: **TextView()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Overrides the BView function to adjust the layout within the panel when its dimensions change. This function is called as the panel is being resized; there's no need to call it or override it in application code.

See also: **BWindow::FrameResized()**

Go()

long **Go(**void**)**

Calls the **Show()** virtual function to place the alert panel on-screen, sets the modal loop for the BAlert in motion, and returns when the loop has quit and the window has been closed. The value returned is the index of the button that the user operated to dismiss the window. Buttons are numbered from left to right, beginning with 0.

To put an alert panel on-screen, simply construct a BAlert object, set its keyboard shortcuts, if any, and call this function. See the example code in the “Overview” section above.

Before returning, this function deletes the BAlert object, and all the objects it created.

See also: the BAlert constructor

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Closes the window in response to messages posted from the window’s buttons. There’s no need for your application to call or override this function.

SetShortcut()

void **SetShortcut(**long *index*, char *shortcut***)**

Sets a *shortcut* character that the user can type to operate the button at *index*. Buttons are indexed from left to right beginning with 0. By default, **B_ENTER** is the shortcut for the rightmost button.

A “Cancel” button should be assigned the **B_ESCAPE** character as a shortcut.

The shortcut doesn’t require the user to hold down a Command key or other modifier (except for any modifiers that would normally be required to produce the *shortcut* character).

The shortcut is valid only while the window is on-screen.

TextView()

inline BTextView ***TextView(**void**)** const

Returns a pointer to the BTextView object that contains the textual information that’s displayed in the panel. The object is created and the text is set when the BAlert is constructed. The BTextView object belongs to the BAlert and should not be freed.

See also: the BAlert constructor, **ButtonAt()**

BBitmap

Derived from: public BObject

Declared in: <interface/Bitmap.h>

Overview

A BBitmap object is a container for an image bitmap; it stores pixel data—data that describes an image pixel by pixel. The class provides a way of specifying a bitmap from raw data, and also a way of creating the data from scratch using the Interface Kit graphics mechanism.

BBitmap functions manage the bitmap data and provide information about it. However, they don't do anything with the data. Placing the image somewhere so that it can be seen is the province of BView functions—such as **DrawBitmap()** and **DragMessage()**—not this class.

Bitmap Data

An image bitmap records the color values of pixels within a rectangular area. The pixels in the rectangle, as on the screen, are arranged in rows and columns. The data is specified in rows, beginning with the top row of pixels in the image and working downward to the bottom row. Each row of data is aligned on a long word boundary and is read from left to right.

New BBitmap objects are constructed with two pieces of information that prepare them to store bitmap data—a bounds rectangle and a color space. For example, this code

```
BRect rect(0.0, 0.0, 39.0, 79.0);
BBitmap *image = new BBitmap(rect, B_COLOR_8_BIT);
```

constructs a bitmap of 40 rows and 80 pixels per row. Each pixel is specified by an 8-bit color value.

The Bounds Rectangle

A BBitmap's bounds rectangle serves two purposes:

- It sets the size of the image. A bitmap covers as many pixels as its bounds rectangle encloses—under the assumption that one coordinate unit equals one pixel, as it does when the display device is the screen.

Since a bitmap can't contain a fraction of a pixel, the bounds rectangle shouldn't contain any fractional coordinates. Without fractional coordinates, each side of the bounds rectangle will be aligned with a column or a row of pixels. The pixels around the edge of the rectangle are included in the image, so the bitmap will contain one more column of pixels than the width of the rectangle and one more row than the rectangle's height. (See the BRect class "Overview" on page 175 for an illustration.)

- It establishes a coordinate system that can be used later by drawing functions, such as **DrawBitmap()** and **DragMessage()**, to designate particular points or portions of the image.

  For example, if one BBitmap was constructed with this bounds rectangle,

  ```
  BRect firstRect(0.0, 0.0, 60.0, 100.0);
  ```

  and another with this rectangle,

  ```
  BRect secondRect(60.0, 100.0, 120.0, 200.0);
  ```

  they would both have the same size and shape. However, the coordinates (60.0, 100.0) would designate the right bottom corner of the first bitmap, but the left top corner of the second.

< If a BBitmap object enlists BViews to create the bitmap data, it must have a bounds rectangle with (0.0, 0.0) at the left top corner. >

The Color Space

The color space of a bitmap determines its depth (how many bits of information are stored for each pixel) and its interpretation (what the data values mean). These five color spaces are currently defined:

B_MONOCHROME_1_BIT
B_GRAYSCALE_8_BIT
B_COLOR_8_BIT
B_RGB_16_BIT
B_RGB_32_BIT

< Currently, bitmap data is stored only in the **B_RGB_32_BIT**, **B_COLOR_8_BIT**, and **B_MONOCHROME_1_BIT** color spaces. The **B_GRAYSCALE_8_BIT** and **B_RGB_16_BIT** color spaces are not used at the present time. >

In the **B_RGB_32_BIT** color space, the color of each pixel is specified by its red, green, and blue components. In the **B_COLOR_8_BIT** color space, colors are specified as byte indices into the color map. In the **B_MONOCHROME_1_BIT** color space, a value of 1 means black and 0 means white. (A more complete description of the five color spaces can be found under "Colors" on page 25 of the introduction to this chapter.)

Specifying the Image

BBitmap objects begin life empty. When constructed, they allocate sufficient memory to store an image of the size and color space specified. However, the memory isn't initialized. The actual image must be set after construction. This can be done by explicitly assigning pixel values with the **SetBits()** function:

```
image->SetBits(rawData, numBytes, 0, COLOR_8_BIT);
```

In addition to this function, BView objects can be enlisted to produce the bitmap. Views are assigned to a BBitmap object just as they are to a BWindow (by calling the **AddChild()** function). In reality, the BBitmap sets up a private, off-screen window for the views. When the views draw, the window renders their output into the bitmap buffer. The rendered image has the same format as the data captured by the **SetBits()** function. **SetBits()** and BViews can be used in combination to create a bitmap.

The BViews that construct a bitmap behave a bit differently than the BViews that draw in regular windows:

- In contrast to BViews attached to an ordinary window, the BViews assigned to a BBitmap can create an image off-screen. When an ordinary window is hidden, it doesn't render images; its BViews may draw, but they don't produce image data. However, the BViews assigned to a BBitmap produce an off-screen bitmap.

- Because they never appear on-screen, the BViews that produce a bitmap image never handle events and never get update messages telling them to draw. You must call their drawing functions directly in your own code.

  This is typically done just once, to create the bitmap. After that, the BViews can be discarded; they'll never be called upon to update the image. However, if the bitmap will change—perhaps to reflect decisions the user makes as the program runs—the BViews can be retained to make the changes.

- Because there are no update messages, the output buffer to the Application Server isn't automatically flushed. You must flush it explicitly in application code. This is best done by calling **Sync()**, rather than **Flush()**, so that you can be sure the entire image has been rendered before the bitmap is used.

- A BBitmap has no background color against which images are drawn. Your code must color every pixel within the bounds rectangle.

- Views that are attached to a BWindow normally draw in the window's thread. However, views attached to a BBitmap don't draw in a separate thread; the BBitmap doesn't set up an independent thread for its private window.

So that you can manage the BViews that are assigned to a BBitmap, the BBitmap class duplicates a number of BWindow functions—such as **AddChild()**, **FindView()**, and **ChildAt()**.

A BBitmap that enlists views to produce the bitmap consumes more system resources than one that relies solely on **SetBits()**. Therefore, by default, BBitmaps refuse to accept

BViews. If BViews will be used to create bitmap data, the BBitmap constructor must be informed so that it can set up the off-screen window and prepare the rendering mechanism.

Transparency

Color bitmaps can have transparent pixels. When the bitmap is imaged in a drawing mode other than **B_OP_COPY**, its transparent pixels won't be transferred to the destination view. The destination image will show through wherever the bitmap is transparent.

To introduce transparency into a **B_COLOR_8_BIT** bitmap, a pixel can be assigned a value of **B_TRANSPARENT_8_BIT**. In a **B_RGB_32_BIT** bitmap, a pixel can be assigned the special value of **B_TRANSPARENT_32_BIT**. (Or **B_TRANSPARENT_32_BIT** can be made the high or low color of the BView drawing the bitmap.)

Transparency is covered in more detail under "Drawing Modes" on page 27 of the chapter introduction.

See also: **system_colors()**

Constructor and Destructor

BBitmap()

BBitmap(BRect *bounds*, color_space *mode*, bool *acceptsViews* = FALSE**)**

Initializes the BBitmap to the size and internal coordinate system implied by the *bounds* rectangle and to the depth and color interpretation specified by the *mode* color space.

This function allocates enough memory to store data for an image the size of *bounds* at the depth required by *mode*, but does not initialize any of it. All pixel data should be explicitly set using the **SetBits()** function, or by enlisting BViews to produce the bitmap. If BViews are to be used, the constructor must be informed by setting the *acceptsViews* flag to **TRUE**. This permits it to set up the mechanisms for rendering the image, including an off-screen window to contain the views.

< Currently, only **B_RGB_32_BIT**, **B_COLOR_8_BIT**, and **B_MONOCHROME_1_BIT** are acceptable as the color space *mode*. **B_RGB_16_BIT** is not supported for the present release and **B_GRAYSCALE_8_BIT** is reinterpreted as **B_COLOR_8_BIT**. >

< If the BBitmap accepts BViews, the left and top sides of its *bounds* rectangle must be located at 0.0. >

~BBitmap()

virtual **~BBitmap(**void**)**

Frees all memory allocated to hold image data, deletes any BViews used to create the image, gets rid of the off-screen window that held the views, and severs the BBitmap's connection to the Application Server.

Member Functions

AddChild()

virtual void **AddChild(**BView **aView***)**

Adds *aView* to the hierarchy of views associated with the BBitmap, attaching it to an off-screen window (one created by the BBitmap for just this purpose) by making it a child of the window's top view. If *aView* already has a parent, it's removed from that view hierarchy and adopted into this one. A view can serve only one window at a time.

Like **AddChild()** in the BWindow class, this function calls the BView's **AttachedToWindow()** function to inform it that it now belongs to a view hierarchy. Every view that descends from *aView* also becomes attached to the BBitmap's off-screen window and receives its own **AttachedToWindow()** notification.

AddChild() fails if the BBitmap was not constructed to accept views.

See also: **BWindow::AddChild()**, **BView::AttachedToWindow()**, **RemoveChild()**, the BBitmap constructor

Bits()

inline void ***Bits(**void**)** const

Returns a pointer to the bitmap data. The data lies in memory shared by the application and the Application Server. The length of the data can be obtained by calling **BitsLength()**—or it can be calculated from the height of the bitmap (the number of rows) and the number of bytes per row.

A **B_RGB_32_BIT** bitmap holds the data in an internal format that's most natural for screen display devices. In this format, the color components are ordered BGRA (blue, green, red, alpha).

See also: **Bounds()**, **BytesPerRow()**, **BitsLength()**

BitsLength()

inline long **BitsLength(**void**)** const

Returns the number of bytes that were allocated to store the bitmap data.

See also: **Bits()**, **BytesPerRow()**

Bounds()

inline BRect **Bounds(**void**)** const

Returns the bounds rectangle that defines the size and coordinate system of the bitmap. This should be identical to the rectangle used in constructing the object.

See also: the BBitmap constructor

BytesPerRow()

inline long **BytesPerRow(**void**)** const

Returns how many bytes of data are required to specify a row of pixels. For example, a monochrome bitmap (one bit per pixel) 80 pixels wide would require twelve bytes per row (96 bits). The extra sixteen bits at the end of the twelve bytes are ignored. Every row of bitmap data is aligned on a long word boundary.

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

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

ChildAt() returns the child BView at *index*, or **NULL** if there's no child at *index*. Indices begin at 0 and count only BViews that were added to the BBitmap (added as children of the top view of the BBitmap's off-screen window) and not subsequently removed.

CountChildren() returns the number of BViews the BBitmap currently has. (It counts only BViews that were added directly to the BBitmap, not BViews farther down the view hierarchy.)

These functions fail if the BBitmap wasn't constructed to accept views.

See also: **BWindow::ChildAt()**, **BView::Parent()**

ColorSpace()

inline color_space **ColorSpace(**void**)** const

Returns the color space of the data being stored (not necessarily the color space of the data passed to the **SetBits()** function). Once set by the BBitmap constructor, the color space doesn't change.

The **color_space** data type is defined in **interface/InterfaceDefs.h** and is explained on page 25 of the introduction to this chapter.

See also: the BBitmap constructor

CountChildren() *see* ChildAt()

FindView()

BView ***FindView(**BPoint *point***)** const
BView ***FindView(**const char **name***)** const

Returns the BView located at *point* within the bitmap, or the BView tagged with *name*. The point must be somewhere within the BBitmap's bounds rectangle, which must have the coordinate origin, (0.0, 0.0), at its left top corner.

If the BBitmap doesn't accept views, this function fails. If no view draws at the *point* given, or no view associated with the BBitmap has the *name* given, it returns **NULL**.

See also: **BView::FindView()**

Lock(), Unlock()

bool **Lock(**void**)**

void **Unlock(**void**)**

These functions lock and unlock the off-screen window where BViews associated with the BBitmap draw. Locking works for this window and its views just as it does for ordinary on-screen windows.

Lock() returns **FALSE** if the BBitmap doesn't accept views or if its off-screen window is unlockable (and therefore unusable) for some reason. Otherwise, it doesn't return until it has the window locked and can return **TRUE**.

See also: **BLooper::Lock()** in the Application Kit

RemoveChild()

virtual bool **RemoveChild(**BView **aView***)**

Removes *aView* from the hierarchy of views associated with the BBitmap, but only if *aView* was added to the hierarchy by calling BBitmap's version of the **AddChild()** function.

If *aView* is successfully removed, **RemoveChild()** returns **TRUE**. If not, it returns **FALSE**.

See also: **AddChild()**

SetBits()

void **SetBits(**const void **data*, long *length*, long *offset*, color_space *mode***)**

Assigns *length* bytes of *data* to the BBitmap object. The new data is copied into the bitmap beginning *offset* bytes (*not* pixels) from the start of allocated memory. To set data beginning with the first (left top) pixel in the image, the *offset* should be 0; to set data beginning with, for example, the sixth pixel in the first row of a **B_RGB_32_BIT** image, the offset should be 20. The offset counts any padding required to align rows of data.

The source data is specified in the *mode* color space, which may or may not be the same as the color space that the BBitmap uses to store the data. If not, the following conversions are automatically made:

- **B_MONOCHROME_1_BIT** and **B_RGB_32_BIT** to **B_COLOR_8_BIT**.
- **B_COLOR_8_BIT** and **B_GRAYSCALE_8_BIT** to **B_RGB_32_BIT**.

Colors may be dithered in the conversion to **B_COLOR_8_BIT**, so that the resulting image will match the original as closely as possible, despite the lost information.

If the color space *mode* is **B_RGB_32_BIT**, the *data* should be triplets of three 8-bit components—red, green, and blue, in that order—without an alpha component. Although stored as 32-bit quantities, the input data is only 24 bits. Rows of source data do not need to be aligned.

However, if the source data is in any *mode* other than **B_RGB_32_BIT**, padding must be added so that each row is aligned on a **long** word boundary.

This function works for all BBitmaps, whether or not BViews are also enlisted to produce the image.

See also: **Bits()**

BBox

Derived from:	public BView
Declared in:	<interface/Box.h>

Overview

A BBox draws a labeled border around other views. It serves only to label those views and organize them visually. It doesn't respond to messages.

The border is drawn around the edge of the view's frame rectangle. If the BBox has a label, the border at the top of box is broken where the label appears (and the border is inset from the top somewhat to make room for the label).

The current pen size of the view determines the width of the border, which by default is 1.0 coordinate unit. If you make the border thicker, it will be inset somewhat so that none of it is clipped by the BBox's frame rectangle. The label is drawn in the current font, which by default is the Erich bitmap font. Both the border and the label are drawn in the current high color; the default high color is black.

The views that the box encloses should be made children of the BBox object.

Constructor and Destructor

BBox()

BBox(BRect *frame*, const char **name* = NULL,
    ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
    ulong *flags* = B_WILL_DRAW**)**

Initializes the BBox by passing all arguments to the BView constructor, and sets the font for displaying the label to the 9.0-point Erich bitmap font. However, the new object doesn't have a label; call **SetLabel()** to assign it one.

See also: **SetLabel()**

~BBox()

virtual **~BBox(**void**)**

Frees the label, if the BBox has one.

Member Functions

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the box and its label. This function is called automatically in response to update messages.

See also: **BView::Draw()**

SetLabel(), Label()

void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

These functions set and return the label that's displayed along the top edge of the box. **SetLabel()** copies *string* and makes it the BBox's label, freeing the previous label, if any. If *string* is **NULL**, it removes the current label and frees it.

Label() returns a pointer to the BBox's current label, or **NULL** if it doesn't have one.

BButton

Derived from:	public BControl
Declared in:	<interface/Button.h>

Overview

A BButton object draws a labeled button on-screen and responds when the button is clicked or when it's operated from the keyboard. If the BButton is the *default button* for its window and the window is the active window, the user can operate it by pressing the Enter key.

BButtons have a single state. Unlike check boxes and radio buttons, the user can't toggle a button on and off. However, the button's value changes while it's being operated. During a click (while the user holds the mouse button down and the cursor points to the button on-screen), the BButton's value is set to 1 (**B_CONTROL_ON**). Otherwise, the value is 0 (**B_CONTROL_OFF**).

This class depends on the control framework defined in the BControl class. In particular, it calls these BControl functions:

- **SetValue()** to make each change in the BControl's value. This is a hook function that you can override to take collateral action when the value changes.
- **Invoke()** to post a message each time the button is clicked or operated from the keyboard. You can designate the object that should handle the message by calling BControl's **SetTarget()** function. A model for the message is set by the BButton constructor (or by BControl's **SetMessage()** function).
- **IsEnabled()** to determine how the button should be drawn and whether it's enabled to post a message. You can call BControl's **SetEnabled()** to enable and disable the button.

A BButton is an appropriate control device for initiating an action. Use a BCheckBox, BPictureButton, or BRadioButtons to set a state.

Hook Functions

MakeDefault()	Makes the BButton the default button for its window or removes that status; can be augmented by derived classes to take note when the status of the button changes.

Constructor

BButton()

```
BButton(BRect frame, const char *name,
        const char *label,
        BMessage *message,
        ulong resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
        ulong flags = B_WILL_DRAW | B_NAVIGABLE)
```

Initializes the BButton by passing all arguments to the BControl constructor. BControl initializes the button's *label* and assigns it a model *message* that identifies the action that should be carried out when the button is invoked.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change.

When the button is attached to a window, it will be resized so that the height of its *frame* rectangle exactly accommodates the height of its label, given the BButton's current font.

See also: the BControl and BView constructors, **BControl::Invoke()**

Member Functions

AttachedToWindow()

```
virtual void AttachedToWindow(void)
```

Augments the BControl version of this function to set the background color of the button to match the background color of its parent. This function also resizes the button vertically so that its height is just adequate to display the label and the button border. The height of the label depends on the BView's font.

Finally, it makes sure that the BButton does not consider itself the default button for the window to which it has just become attached—even if it may have been the default button for the window to which it was previously attached.

See also: **BView::AttachedToWindow()**, **BControl::AttachedToWindow()**, **MakeDefault()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the button and labels it. If the BButton's value is anything but 0, the button is highlighted. If it's disabled, it drawn in muted shades of gray. Otherwise, it's drawn in its ordinary, enabled, unhighlighted state.

See also: **BView::Draw()**

IsDefault() *see* MakeDefault

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the inherited version of **KeyDown()** to respond to messages reporting that the user pressed the Enter key or the space bar. Its response is to:

- Momentarily highlight the button and change its value, and
- Post a copy of the model BMessage to the target receiver.

The BButton can expect **KeyDown()** function calls when it's the focus view for the active window (which results when the user navigates to it) and also when it's the default button for the window and *aChar* is **B_ENTER**.

See also: **BControl::Invoke()**, **BView::KeyDown()**, **MakeDefault()**

MakeDefault(), IsDefault()

virtual void **MakeDefault(**bool *flag***)**

bool **IsDefault(**void**)** const

MakeDefault() makes the BButton the default button for its window when *flag* is **TRUE**, and removes that status when *flag* is **FALSE**. The default button is the button the user can operate by striking the Enter key when the window is the active window. **IsDefault()** returns whether the BButton is currently the default button.

A window can have only one default button at a time. Setting a new default button, therefore, may deprive another button of that status. When **MakeDefault()** is called with an argument of **TRUE**, it generates a **MakeDefault()** call with an argument of **FALSE** for previous default button. Both buttons are redisplayed so that the user can see which one is currently the default.

The default button can also be set by calling BWindow's **SetDefaultButton()** function. That function makes sure that the button that's forced to give up default status and the button that obtains it are both notified through **MakeDefault()** function calls.

MakeDefault() is therefore a hook function that can be augmented to take note each time the default status of the button changes. It's called once for each change in status, no matter which function initiated the change.

See also: **BWindow::SetDefaultButton()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **MouseDown()** to track the cursor while the user holds the mouse button down. As the cursor moves in and out of the button, the BButton's value is reset accordingly. The **SetValue()** virtual function is called to make the change each time.

If the cursor is inside the BButton's bounds rectangle when the user releases the mouse button, this function posts a copy of the model message so that it will be dispatched to the target object.

See also: **BView::MouseDown()**, **BControl::Invoke()**, **BControl::SetTarget()**

BCheckBox

Derived from:	public BControl
Declared in:	<interface/CheckBox.h>

Overview

A BCheckBox object draws a labeled check box on-screen and responds to a keyboard action or a click by changing the state of the device. A check box has two states: An “X” is displayed in the box when the object’s value is 1 (**B_CONTROL_ON**), and is absent when the value is 0 (**B_CONTROL_OFF**). The BCheckBox is invoked (it posts a message to the target receiver) whenever its value changes in either direction—when it’s turned on *and* when it’s turned off.

A check box is an appropriate control device for setting a state—turning a value on and off. Use menu items or buttons to initiate actions within the application.

Constructor

BCheckBox()

BCheckBox(BRect *frame*, const char **name*,
const char **label*,
BMessage **message*,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = B_WILL_DRAW | B_NAVIGABLE**)**

Initializes the BCheckBox by passing all arguments to the BControl constructor. BControl initializes the *label* of the check box and assigns it a model *message* that encapsulates the action that should be taken when the state of the check box changes.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed unchanged to the BView constructor.

When the BCheckBox is attached to a window, the height of its *frame* rectangle will be adjusted so that it has exactly the right amount of room to display the check box icon and the label, given its current font. The object draws at the vertical center of its frame rectangle beginning at the left side.

See also: the BControl and BView constructors, **AttachedToWindow()**

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of **AttachedToWindow()** to set the view and low colors of the BCheckbox to the match its parent's view color, and to resize the view vertically to fit the height of the label it displays. The height of the label depends on the BCheckBox's font, which the BControl constructor sets to the Emily bitmap font.

See also: **BControl::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the check box and its label. If the current value of the BCheckBox is 1 (**B_CONTROL_ON**), it's marked with an "X". If the value is 0 (**B_CONTROL_OFF**), it's empty.

See also: **BView::Draw()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event within the check box by tracking the cursor while the user holds the mouse button down. If the cursor is inside the bounds rectangle when the user releases the mouse button, this function toggles the value of the BCheckBox and calls **Draw()** to redisplay it. If the box was empty before the mouse-down event, it will be marked afterward; if marked before, it will be empty afterwards.

When the value of the BCheckBox changes, a copy of the model BMessage is posted so that it can be delivered to the object's target handler. See BControl's **Invoke()** and **SetTarget()** functions for more information. The message is dispatched by calling the target's **MessageReceived()** virtual function.

The target object can get a pointer to the BCheckBox from the message, and use it to discover the object's new value. For example:

```
void MyHandler::MessageReceived(BMessage *msg)
{
    . . .
    BCheckBox *box = (BCheckBox *)msg->FindObject("source");
    if ( message->Error() == B_NO_ERROR ) {
        long value = box->Value();
        . . .
    }
    . . .
}
```

See also: **BControl::Invoke()**, **BControl::SetTarget()**, and **BControl::SetValue()**

BColorControl

Derived from: public BControl

Declared in: <interface/ColorControl.h>

Overview

A BColorControl object displays an on-screen device that permits users to pick a color. It reports the color as its current value—an **rgb_color** data structure stored as a **long** integer. If a model message is provided, it announces each change in value by sending a copy of the message to a designated target.

When the screen is 8 bits deep, the BColorControl object presents users with a matrix of the 256 available colors. The user chooses a color by pressing the primary mouse button while the cursor is over one of the cells in the matrix. Dragging from cell to cell changes the selected color. The arrow keys can similarly change the selection when the object is the focus view. The BColorControl's value changes each time the selection does.

When the screen is 32 bits deep, the BColorControl object displays ramps for each color component. The user changes the current color by modifying a red, green, or blue component value.

Constructor and Destructor

BColorControl()

BColorControl(BPoint *leftTop*, color_control_layout *matrix*, long *cellSide*,
const char **name*, BMessage **message* = NULL,
bool *bufferedDrawing* = FALSE**)**

Initializes the BColorControl so that the left top corner of its frame rectangle will be located at the stated *leftTop* point in the coordinate system of its parent view. The frame rectangle will be large enough to display 256 color cells arranged in the specified *matrix*, which can be any of the following constants:

B_CELLS_4x64
B_CELLS_8x32
B_CELLS_16x16
B_CELLS_32x8
B_CELLS_64x4

For example, **B_CELLS_4x64** lays out a matrix with four cell columns and 64 rows; **B_CELLS_32x8** specifies 32 columns and 8 rows. Each cell is a square *cellSide* coordinate units on a side.

When the screen is 32 bits deep, the same frame rectangle will display four color ramps, one each for the red, green, and blue components, plus a disabled ramp for the alpha component. You might choose *matrix* and *cellSize* values with a view toward how the resulting bounds rectangle would be divided into four horizontal rows.

The *name* argument assigns a name to the object as a BHandler. It's the same as the argument declared by the BView constructor.

If a model *message* is supplied, the BColorControl will announce every change in color value by calling **Invoke()** (defined in the BControl class) to post a copy of the message to a designated target.

If the *bufferedDrawing* flag is **TRUE**, all changes to the on-screen display will first be made in an off-screen bitmap and then copied to the screen. This makes the drawing smoother, but it requires more memory.

The initial value of the new object is 0, which when translated to an **rgb_color** structure, means black.

See also: **BHandler::SetName()**, **BControl::Invoke()**

~BColorControl()

virtual **~BColorControl(**void**)**

Gets rid of the off-screen bitmap, if one was requested when the object was constructed.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of this function to set the BColorControl's view color and low color to be the same as its parent's view color.

See also: **BControl::AttachedToWindow()**, **BView::SetViewColor()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Overrides the BView version of this function to draw the color control.

See also: **BView::Draw()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the BControl version of **KeyDown()** to allow the user to navigate within the color control using the arrow keys.

See also: **BControl::KeyDown()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of this function to allow the user to operate the color control with the mouse.

See also: **BView::MouseDown()**

SetValue(), ValueAsColor()

virtual void **SetValue(**long *color***)**
virtual void **SetValue(**rgb_color *color***)**

rgb_color **ValueAsColor(**void**)**

These functions set and return the BColorControl's current value—the last color that the user selected.

The version of **SetValue()** that takes a **long** argument is essentially the same as the BControl version of the function, which it augments only to take care of class-internal housekeeping details. The version that takes an **rgb_color** argument packs the information from that structure into a **long** integer and passes it to the other version of the function. Like all other objects that derive from BControl, a BColorControl stores its current value as a **long**; no information is lost in the translation from an **rgb_color** structure to an integer.

ValueAsColor() is an alternative to the **Value()** function inherited from the BControl class. It returns the object's current value as an **rgb_color**; **Value()** returns it as a **long**.

See also: **BControl::SetValue()**

BControl

Derived from:	public BView
Declared in:	<interface/Control.h>

Overview

BControl is an abstract class for views that draw control devices on the screen. Objects that inherit from BControl emulate, in software, real-world control devices—like the switches and levers on a machine, the check lists and blank lines on a form to fill out, or the dials and knobs on a home appliance.

Controls translate the messages that report generic mouse and keyboard events into other messages with more specific instructions for the application. A BControl object can be customized by setting the message it posts when invoked and the target object that should handle the message.

Controls also register a current value, stored as a **long** integer that's typically set to **B_CONTROL_ON** or **B_CONTROL_OFF**. The value is changed only by calling **SetValue()**, a virtual function that derived classes can implement to be notified of the change.

The Interface Kit currently includes six classes derived from BControl—BButton, BPictureButton, BRadioButton, BCheckBox, BColorControl, and BTextControl. In addition, it has two classes—BListView and BMenuItem—that implement control devices but are not derived from this class. BListView shares an interface with the BList class (of the Support Kit) and BMenuItem is designed to work with the other classes in the menu system.

As BListView and BMenuItem demonstrate, it's possible to implement a control device that's not a BControl. However, it's simpler to take advantage of the code that's already provided by the BControl class. That way you can keep a simple programming interface and avoid reimplementing functions that BControl has defined for you. If your application defines its own control devices—dials, sliders, selection lists, and the like—they should be derived from BControl.

Hook Functions

SetEnabled()	Enables and disables the control device; can be augmented by derived classes to note when the state of the object has changed.
SetValue()	Changes the value of the control device; can be augmented to take collateral action when the change is made.

Constructor and Destructor

BControl()

BControl(BRect *frame*, const char **name*,
const char **label*, BMessage **message*,
ulong *resizingMode*, ulong *flags***)**

Initializes the BControl by setting its initial value to 0 (**B_CONTROL_OFF**), assigning it a *label*, and registering a model *message* that captures what the control does—the command it gives when it's invoked and the information that accompanies the command. The *label* and the *message* can each be **NULL**.

The *label* is copied, but the *message* is not. The BMessage object becomes the property of the BControl; it should not be deleted, posted, assigned to another object, or otherwise used in application code. The label and message can be altered after construction with the **SetLabel()** and **SetMessage()** functions.

The BControl class doesn't define a **Draw()** function to draw the label or a **MouseDown()** function to post the message. (It does define **KeyDown()**, but only to enable keyboard navigation between controls.) It's up to derived classes to determine how the *label* is drawn and how the *message* is to be used. Typically, when a BControl object needs to take action (in response to a click, for example), it calls the **Invoke()** function, which copies the model message and posts the copy so that it will be dispatched to the designated target. By default, the target is the window where the control is located, but **SetTarget()** can designate another handler.

Before posting a copy of the model message, **Invoke()** adds two data entries to it, under the names "when" and "source". These names should not be used for data items in the model.

The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The BControl begins life enabled, and the Emily bitmap font is made the default font for all control devices.

See also: the BView constructor, **BLooper::PostMessage()** in the Application Kit, **SetLabel()**, **SetMessage()**, **SetTarget()**, **Invoke()**

~BControl()

virtual **~BControl(**void**)**

Frees the model message and all memory allocated by the BControl.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Overrides BView's version of this function to make the BWindow to which the BControl has become attached the default target for the **Invoke()** function, provided that another target hasn't already been set. To designate the target, it calls **SetTarget()**, a virtual function.

AttachedToWindow() is called for you when the BControl becomes a child of a view already associated with the window.

See also: **BView::AttachedToWindow()**, **BView::SetFontName()**, **Invoke()**, **SetTarget()**

Command() *see* SetMessage()

Invoke()

protected:

void **Invoke(**void**)**

Copies the BControl's model BMessage and posts the copy so that it will be dispatched to the designated target. The following two pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the control was invoked, as measured in microseconds from the time the machine was last booted.
"source"	**B_OBJECT_TYPE**	A pointer to the BControl object. This permits the message handler to request more information from the source of the message.

These two names shouldn't be used for data entries in the model.

If the control doesn't have a target BHandler, but it does have a designated BLooper where it can post the message, it will ask the BLooper for its preferred handler and name it as the target. Since the preferred handler for a BWindow object is the current focus view, this option allows control devices to be targeted to whatever view happens to be in focus at the time. See the **SetTarget()** function for information on how to designate a target BHandler and BLooper for the control.

Invoke() is designed to be called from the **MouseDown()** and **KeyDown()** functions defined for derived classes; it's not called for you in BControl code. It's up to each derived class to define what user actions trigger the call to **Invoke()**—what activity constitutes "invoking" the control.

This function doesn't check to make sure the BControl is currently enabled. Derived classes should make that determination before calling **Invoke()**.

See also: **SetTarget()**, **SetMessage()**, **SetEnabled()**

IsEnabled() *see* SetEnabled()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the BView version of **KeyDown()** to toggle the BControl's value and call **Invoke()** when *aChar* is the **B_SPACE** character or **B_ENTER**. This is done to facilitate keyboard navigation and make all derived control devices operable from the keyboard. Some derived classes—BCheckBox in particular—find this version of the function to be adequate. Others, like BRadioButton, reimplement it.

KeyDown() is called only when the BControl is the focus view in the active window. (However, if the window has a default button, **B_ENTER** events will be passed to that object and won't be dispatched to the focus view.)

See also: **BView::KeyDown()**, **MakeFocus()**

Label() *see* SetLabel()

MakeFocus()

virtual void **MakeFocus(**bool *focused* = TRUE**)**

Augments the BView version of this function to call the BControl's **Draw()** function when the focus changes. This is done to aid keyboard navigation among control devices. If the **Draw()** function of a derived class has a section of code that checks whether the object is in focus and marks the on-screen display to show that it is (and removes any such marking when it isn't), the visual part of keyboard navigation will be taken care of. The derived class doesn’t have to reimplement **MakeFocus()**. Most of the derived classes implemented in the Interface Kit depend on this version of the function.

See also: **BView::MakeFocus()**, **KeyDown()**

SetEnabled(), IsEnabled()

> virtual void **SetEnabled(**bool *enabled***)**
>
> bool **IsEnabled(**void**)** const

SetEnabled() enables the BControl if the *enabled* flag is **TRUE**, and disables it if *enabled* is **FALSE**. **IsEnabled()** returns whether or not the object is currently enabled. BControls are enabled by default.

While disabled, a BControl won’t let the user navigate to it; the **B_NAVIGABLE** flag is turned off if *enabled* is **FALSE** and turned on again if *enabled* is **TRUE**.

Typically, a disabled BControl also won’t post messages or respond visually to mouse and keyboard manipulation. To indicate this nonfunctional state, the control device is displayed on-screen in subdued colors. However, it’s left to each derived class to carry out this strategy in a way that’s appropriate for the kind of control it implements. The BControl class merely marks an object as being enabled or disabled; none of its functions take the enabled state of the device into account.

Derived classes can augment **SetEnabled()** (override it) to take action when the control device becomes enabled or disabled. To be sure that **SetEnabled()** has been called to actually make a change, its current state should be checked before calling the inherited version of the function. For example:

```
void MyControl::SetEnabled(bool enabled)
{
    if ( enabled == IsEnabled() )
        return;
    BControl::SetEnabled(enabled);
    /* Code that responds to the change in state goes here. */
}
```

Note, however, that you don’t have to override **SetEnabled()** just to update the on-screen display when the control becomes enabled or disabled. If the BControl is attached to a window, the Kit’s version of **SetEnabled()** always calls the **Draw()** function. Therefore, the device on-screen will be updated automatically—as long as **Draw()** has been implemented to take the enabled state into account.

See also: the BControl constructor

SetLabel(), Label()

virtual void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

These functions set and return the label on a control device—the text that's displayed, for example, on top of a button or alongside a check box or radio button. The label is a null-terminated string.

SetLabel() makes a copy of *string*, replaces the current label with it, frees the old label, and updates the control on-screen so the new label will be displayed to the user—but only if the *string* that's passed differs from the current label. The label is first set by the constructor and can be modified thereafter by this function.

Label() returns the current label. The string it returns belongs to the BControl and may be altered or freed without notice.

See also: the BControl constructor, **BView::AttachedToWindow()**, **BView::SetFontName()**

SetMessage(), Message(), Command()

virtual void **SetMessage(**BMessage **message***)**

BMessage ***Message(**void**)** const

ulong **Command(**void**)** const

SetMessage() sets the model BMessage that defines what the BControl does, and frees the message that was previously set. **Message()** returns a pointer to the BMessage that's the current model, and **Command()** returns its **what** data member. The message is first set by the BControl constructor.

Because **Invoke()** adds "when" and "source" entries to the messages it posts, these two names shouldn't be used for any data entries in the model BMessage.

The model message passed to **SetMessage()** and returned by **Message()** belongs to the BControl object; it can be modified in application code, but it shouldn't be deleted (except by passing **NULL** to **SetMessage()**), posted, or put to any other use.

See also: the BControl constructor, **Invoke()**, **SetTarget()**

SetTarget(), Target()

virtual long **SetTarget(**BHandler **target***)**
virtual long **SetTarget(**BLooper **looper*, bool *targetsPreferredHandler***)**

BHandler ***Target(**BLooper ***looper* = NULL**)** const

These functions set and return the object that's targeted to handle the messages that the BControl posts (through its **Invoke()** function).

The version of **SetTarget()** that takes a single argument sets the *target* BHandler object. It's successful only if the *target* can reveal, through its **Looper()** function, a BLooper object where **Invoke()** can post messages so that they will be dispatched to that target. Therefore, the *target* BHandler must either:

- Be a BLooper itself (such as a BWindow), so that it can fulfill the roles of both BLooper and BHandler, or
- Have been added to a BLooper (as BViews are attached to BWindows).

Armed with both the BLooper and the target BHandler, **Invoke()** calls the BLooper's **PostMessage()** function and names the *target* as the object that should handle the message:

```
theLooper->PostMessage(theMessage, target);
```

After being set as the control's *target*, the BHandler must maintain its association with the BLooper. If it moves to another BLooper, **PostMessage()** will fail.

The version of **SetTarget()** that takes two arguments sets the BLooper object where the BControl's **Invoke()** function should post messages. If the *targetsPreferredHandler* flag is **FALSE**, messages will be targeted to the *looper* object itself—it will also act as the handler. In other words, passing a BLooper and **FALSE** to the version of **SetTarget()** that takes two arguments accomplishes the same thing as simply passing the BLooper alone to the version that takes one argument. These two lines of code accomplish the same thing:

```
myControl->SetTarget(someLooper, FALSE);
myControl->SetTarget(someLooper);
```

The two-argument version of **SetTarget()** becomes interesting only if the *targetsPreferredHandler* flag is **TRUE**. In this case, messages are targeted to the *looper*'s preferred handler (the object returned by its **PreferredHandler()** function). This permits the targeting decision to be made dynamically, at the time **Invoke()** is called:

```
looper->PostMessage(theMessage, looper->PreferredHandler());
```

For example, the preferred handler of a BWindow object is the current focus view. Therefore, by passing a BWindow *looper* and **TRUE** to **SetTarget()**,

```
myControl->SetTarget(someWindow, TRUE);
```

the control device can be targeted to whatever BView happens to be in focus at the time the control is invoked. This is useful for controls that act on the current selection. (Note, however, that if the **PreferredHandler()** is **NULL**, the *looper* itself becomes the target, just as it would if the *targetsPreferredHandler* flag were **FALSE**.)

When successful, **SetTarget()** returns **B_NO_ERROR**. It fails and returns **B_BAD_VALUE** if the proposed *target* or *looper* is **NULL**. The one-argument version also returns **B_BAD_VALUE** if it can't discover a BLooper from the target handler.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where **Invoke()** will post messages. If the target BHandler is the preferred handler of the *looper*, **Target()** returns **NULL**. In other words, passing a BLooper and **TRUE** to **SetTarget()** causes **Target()** to report that there is a *looper*, but a **NULL** target—the BLooper is known, but the BHandler is not. Passing a BLooper and **FALSE** to **SetTarget()** causes **Target()** to report that the same object is both *looper* and target.

By default (established by **AttachedToWindow()**), both roles—BLooper and BHandler—are filled by the BWindow where the control device is located.

See also: **BHandler::Looper()** and **BLooper::PreferredHandler()** in the Application Kit, **BWindow::PreferredHandler()**, **Invoke()**, **AttachedToWindow()**

SetValue(), Value()

virtual void **SetValue(**long *value***)**

long **Value(**void**)** const

These functions set and return the value of the BControl object.

SetValue() assigns the object a new value. If the *value* passed is in fact different from the BControl's current value, this function calls the object's **Draw()** function so that the new value will be reflected in what the user sees on-screen; otherwise it does nothing.

Value() returns the current value.

Classes derived from BControl should call **SetValue()** to change the value of the control device in response to user actions. The derived classes defined in the Be software kits change values only by calling this function.

Since **SetValue()** is a virtual function, you can override it to take note whenever a control's value changes. However, if you want your code to act only when the value actually changes, you must check to be sure the new value doesn't match the old before calling the inherited version of the function. For example:

```
void MyControl::SetValue(long value)
{
    if ( value != Value() ) {
        BControl::SetValue(value);
        /* MyControl's additions to SetValue() go here */
    }
}
```

Remember that the BControl version of **SetValue()** does nothing unless the new value differs from the old.

Target() *see* SetTarget()

Value() *see* SetValue()

BListView

Derived from:	public BView
Declared in:	<interface/ListView.h>

Overview

A BListView is a view that displays a list of items the user can select and invoke. This class is based on the BList class of the Support Kit. Every member function of the BList class is replicated by BListView, so you can treat a BListView object just like a BList. BListView simply makes the list visible.

Displaying the List

In both classes, the list keeps track of data pointers. Adding an item to the list adds only the pointer; the data itself isn't copied. Neither class imposes a type restriction on the data (both declare items to be type **void ***). However, by default, BListView assumes they're pointers to strings (type **char ***). Its functions can display the strings, highlight them when selected, and so on. As long as only string pointers are placed in the list, a BListView object can be used as is. However, if the list is to contain another kind of data, it's necessary to derive a class from BListView and reimplement some of its hook functions.

When the contents of the list change, the BListView makes sure the visible list on-screen is updated. However, it can know that something changed only when a data pointer changes, since pointers are all that the list records. If any pointed-to data is altered, but the pointer remains the same, you must force the list to be redrawn (by calling the **InvalidateItem()** function or BView's **Invalidate()**).

Selecting and Invoking Items

The user can click an item in the list to select it and double-click an item to both select and invoke it. The user can also select and invoke items from the keyboard. The navigation keys (such as Down Arrow, Home, and Page Up) select items; Enter invokes the item that's currently selected.

The BListView highlights the selected item, but otherwise it doesn't define what, if anything, should take place when an item is selected. You can determine that yourself by registering a "selection message" (a BMessage object) that should be delivered to a target destination whenever the user selects an item.

Similarly, the BListView doesn't define what it means to "invoke" an item. You can register a separate "invocation message" that's posted whenever the user double-clicks an item or presses Enter while an item is selected. For example, if the user double-clicks an item in a list of file names, a message might be posted telling the BApplication object to open that file.

A BListView doesn't have a default selection message or invocation message. Messages are posted only if registered with the **SetSelectionMessage()** and **SetInvocationMessage()** functions. The registered message is only a model. When an item is selected or invoked, the BListView makes a copy of the model, adds information to the copy about itself and the item, then posts the copy. See the function descriptions for information on the data that automatically gets added to the message.

See also: the BList class in the Support Kit

Hook Functions

DrawItem()	Draws the character string that the item points to; can be reimplemented to draw from another kind of data.
HighlightItem()	Highlights the item by inverting all the colors in its frame rectangle; can be reimplemented to highlight in a different way.
Invoke()	Posts the invocation message, if one has been registered for the BListView; can be augmented to do whatever else may be necessary when a item is invoked.
ItemHeight()	Returns the height of a single item, assuming that it's a character string and is to be drawn in the current font; can be reimplemented to return the height required to draw a different kind of item. All items are taken to have the same height.
Select()	Highlights the selected item and posts the selection message, if one has been registered for the BListView; can be augmented to take any collateral action that may be required when the selection changes.

Constructor and Destructor

BListView()

> **BListView(**BRect *frame*, const char **name*,
> ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
> ulong *flags* =
> B_WILL_DRAW | B_NAVIGABLE | B_FRAME_EVENTS**)**

Initializes the new BListView. The *frame*, *name*, *resizingMode*, and *flags* arguments are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The list begins life empty. Call **AddItem()** or **AddList()** (documented for the BList class) to put items in the list. Call **Select()** (documented below) to select one of the items so that it's highlighted when the list is initially displayed to the user.

See also: the BView constructor, **BList::AddItem()**

~BListView()

> virtual **~BListView(**void**)**

Frees the model messages, if any, and all memory allocated to hold the list of items.

Member Functions

The BListView class reimplements *all* of the member functions of the BList class in the Support Kit. BListView's versions of these functions work identically to the BList versions, except that a BListView makes sure that the on-screen display is properly updated whenever the list changes.

Consequently, this section excludes all functions that BList and BListView have in common. It concentrates instead on those member functions that deal with the BListView's behavior as a view, not as a list. See the BList class for information on the functions that you can use to manipulate the BListView's list.

AttachedToWindow()

> virtual void **AttachedToWindow(**void**)**

Sets up the BListView so that it's prepared to draw character strings for items, and makes the BWindow to which the object has become attached the target for messages posted by the **Select()** and **Invoke()** functions—provided another target hasn't already been set.

This function is called for you when the BListView becomes part of a window's view hierarchy.

See also: **BView::AttachedToWindow()**, **SetTarget()**

BaselineOffset()

protected:

float **BaselineOffset(**void**)**

Returns the distance from the bottom of an item's frame rectangle to the baseline where the item, assuming it is a character string, is drawn. The string is drawn beginning at a point that's offset 2.0 coordinate units from the left of the frame rectangle and **BaselineOffset()** units from the bottom. The offsets are the same for all items.

This function will give unreliable results unless the BListView is attached to a window.

CurrentSelection()

inline long **CurrentSelection(**void**)** const

Returns the index of the currently selected item, or a negative number if no item is selected.

See also: **Select()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Calls the **DrawItem()** hook function to draw each visible item in the *updateRect* area of the view and highlights the currently selected item by calling the **HighlightItem()** hook function.

Draw() is called for you whenever the list view is to be updated or redisplayed; you don't need to call it yourself. You also don't need to reimplement it, even if you're defining a list that displays something other than character strings. You should implement data-specific versions of **DrawItem()** and **HighlightItem()** instead.

See also: **BView::Draw()**, **DrawItem()**, **HighlightItem()**

DrawItem()

protected:

virtual void **DrawItem(**BRect *updateRect*, long *index***)**

Draws the item at *index*. The default version of this function assumes that the item is a character string. It can be reimplemented by derived classes to draw differently, based on other kinds of data.

The *updateRect* rectangle is stated in the BListView's coordinate system. It's the portion of the item's frame rectangle that needs to be updated. The full frame rectangle of the item is returned by the **ItemFrame()** function.

The **Draw()** function determines which items in the BListView need to be updated and calls **DrawItem()** for each one.

See also: **ItemHeight()**, **ItemFrame()**, **HighlightItem()**, **BaselineOffset()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Updates the on-screen display in response to a notification that the BListView's frame rectangle has been resized. In particular, this function looks for a vertical scroll bar that's a sibling of the BListView. It adjusts this scroll bar to reflect the way the list view was resized, under the assumption that it must have the BListView as its target.

FrameResized() is called automatically at the appropriate times; you shouldn't call it yourself.

See also: **BView::FrameResized()**

HighlightItem()

protected:

virtual void **HighlightItem(**bool *flag*, long *index***)**

Highlights the item at *index* if *flag* is **TRUE**, and removes the highlighting if *flag* is **FALSE**. Items are highlighted by inverting all colors in their frame rectangles.

This function is called (by **Draw()**) to highlight the selected item and (by **Select()**) to change the item that's highlighted whenever the selection changes. It can be reimplemented in a derived class to highlight in a different way.

See also: **Select()**, **Draw()**

InvalidateItem()

void **InvalidateItem(**long *index***)**

Invalidates the item at *index* so that an update message will be sent forcing the BListView to redraw it.

See also: **BView::Invalidate()**

Invoke()

virtual void **Invoke(**long *index***)**

Invokes the item at *index*, provided that the *index* isn't out-of-range.

This function is called whenever the user double-clicks an item in the list, or presses the Enter key while the BListView is the current focus view for the window and there's a selected item. It can also be called from application code to invoke a particular item; usually **Select()** would first be called to select the item.

To invoke an item that's identified by a pointer, first call **IndexOf()** to find where it's located in the list:

```
long i = myList->IndexOf(someItem);
myList->Select(i);
myList->Invoke(i);
```

If a model "invocation message" has been registered with the BListView (through **SetInvocationMessage()**), **Invoke()** makes a copy of the message, adds information to the copy identifying the BListView and the invoked item, and posts the copy so that it will be handled by the designated target. The default target (established by **AttachedToWindow()**) is the BWindow where the BListView is located. **SetTarget()** can be called to name another BHandler for the message. It can also be called to set a particular BLooper where the message should be posted, but to let that BLooper's preferred handler respond to the message. In this case, the exact target will be picked when **Invoke()** is called.

What it means to "invoke" an item depends entirely on the BMessage that's posted and the receiver's response when it gets the message. This function does nothing but post the message.

See also: **Select()**, **SetInvocationMessage()**, **SetTarget()**

IsItemSelected()

inline bool **IsItemSelected(**long *index***)** const

Returns **TRUE** if the item at *index* is currently selected, and **FALSE** if it's not.

See also: **CurrentSelection()**

ItemFrame()

protected:

  BRect **ItemFrame(**long *index***)** const

Returns the frame rectangle of the item at *index*. The rectangle defines the area where the item is drawn; it's stated in the coordinate system of the BListView. The rectangle is calculated from the ordinal position of the item in the list and the value returned by **ItemHeight()**.

It's expected that you'd need to find an item's frame rectangle only if you're implementing a **DrawItem()** function.

< This function currently doesn't check to be sure that the index is in range. >

See also: **DrawItem()**

ItemHeight()

protected:

  virtual float **ItemHeight(**void**)** const

Returns how much vertical room is required to draw a single item in the list—how high each item's frame rectangle should be. The BListView calls **ItemHeight()** extensively to determine where items are located and where to draw them. By default, it returns a height sufficient to draw a character string in the current font.

A derived class that draws items other than character strings should reimplement **ItemHeight()** so that it returns the height required to draw one of its items.

See also: **DrawItem()**

KeyDown()

  virtual void **KeyDown(**ulong *aChar***)**

Permits the user to operate the list using the following keys:

Keys	Perform Action
Up Arrow and Down Arrow	Select the items that are immediately before and immediately after the currently selected item.
Page Up and Page Down	Select the items that are one viewful above and below the currently selected item—or the first and last items if there's no item a viewful away.
Home and End	Select the first and last items in the list.
Enter and the space bar	Invoke the currently selected item.

This function also incorporates the inherited BView version so that the Tab key can navigate to another view.

KeyDown() is called to report **B_KEY_DOWN** messages when the BListView is the focus view of the active window; you shouldn't call it yourself.

See also: **BView::KeyDown()**, **Select()**, **Invoke()**

MakeFocus()

virtual void **MakeFocus(**bool *focused* = TRUE**)**

Overrides the BView version of **MakeFocus()** to draw an indication that the BListView has become the focus for keyboard events when the *focused* flag is **TRUE**, and to remove that indication when the flag is **FALSE**.

See also: **BView::MakeFocus()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Determines which item is located at *point* and calls **Select()** to select it (for a single-click or the first event in a series) and **Invoke()** to invoke it (for a double-click or the second in a series).

This function also makes the BListView the focus view so the user can operate the list from the keyboard.

MouseDown() is called to notify the BListView of a mouse-down event; you don't need to call it yourself.

See also: **BView::MouseDown()**, **Select()**, **Invoke()**

Select()

virtual void **Select(**long *index***)**

Selects the item located at *index*, provided that the *index* isn't out-of-range. This function removes the highlighting from the previously selected item and highlights the new selection, scrolling the list so the item is visible if necessary. Selecting an item also marks it as the item that **CurrentSelection()** returns and that the Enter key can invoke.

Select() is called whenever the user selects an item, using either the keyboard or the mouse. It can also be called from application code to set an initial selection in the list or change the current selection.

If a model "selection message" has been registered with the BListView, **Select()** copies the message, adds information to the copy identifying the list and the item that was selected, and posts the copy so that it will be dispatched to the target BHandler. If a message hasn't been registered, "selecting" an item simply means to highlight it and mark is as the selected item.

Typically, BListViews are set up to post a message when an item is invoked, but not when one is selected.

See also: **SetSelectionMessage()**, **Invoke()**

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

SetFontName(), **SetFontSize()**, and **SetFontShear()** augment their BView counterparts to recalculate the layout of items in the list when the font changes. However, the list is not automatically redisplayed in the new font.

SetFontRotation() is disabled; a rotated font is incompatible with a list horizontal items.

See also: **BView::SetFontName()**

SetInvocationMessage(), InvocationMessage(), InvocationCommand()

virtual void **SetInvocationMessage(**BMessage **message***)**

BMessage ***InvocationMessage(**void**)** const

ulong **InvocationCommand(**void**)** const

These functions set and return information about the BMessage that the BListView posts when an item is invoked.

SetInvocationMessage() assigns *message* to the BListView, freeing any message previously assigned. The message becomes the responsibility of the BListView object and will be freed only when it's replaced by another message or the BListView is freed; you shouldn't free it yourself. Passing a **NULL** pointer to this function deletes the current message without replacing it.

The BListView treats the BMessage as its "invocation message," a model for the message it posts when an item in the list is invoked. The **Invoke()** function makes a copy of the model and adds two pieces of relevant information. It then posts the copy, not the original.

The added information identifies the BListView and the invoked item:

Data name	Type code	Description
"source"	**B_OBJECT_TYPE**	A pointer to the BListView object.
"index"	**B_LONG_TYPE**	The index of the item that was invoked.

These names should not be used for any data that you add to the model *message*.

Given this information, the message receiver can get a pointer to item data. For example:

```
void myWindow::MessageReceived(BMessage *message)
{
    BListView *theList;
    long theIndex;
    char *theItem;
    . . .
    theList = (BListView *)message->FindObject("source");
    if ( message->Error() == B_NO_ERROR ) {
        theIndex = message->FindLong("index");
        if ( message->Error() == B_NO_ERROR ) {
            theItem = (char *)theList->ItemAt(theIndex);
            . . .
        }
    }
    . . .
 }
```

(Although not shown in this example, you might also want to use the **cast_as()** macro to make sure that it's safe to cast the "source" object pointer to the BListView class.)

InvocationMessage() returns a pointer to the model BMessage and **InvocationCommand()** returns its **what** data member. The message belongs to the BListView; it can be altered by adding or removing data, but it shouldn't be deleted. Nor should it be posted or sent anywhere, since that would eventually free it. To get rid of the current message, pass a **NULL** pointer to **SetInvocationMessage()**.

See also: **Invoke()**, the BMessage class

SetSelectionMessage(), SelectionMessage(), SelectionCommand()

virtual void **SetSelectionMessage(**BMessage **message***)**

BMessage ***SelectionMessage(**void**)** const

ulong **SelectionCommand(**void**)** const

These functions set, and return information about, the message that a BListView posts whenever one of its items is selected. They're exact counterparts to the invocation message functions described above under **SetInvocationMessage()**, except that the "selection message" is posted whenever an item in the list is selected, rather than when invoked. It's more common to take action (to post a message) on invoking an item than on selecting one.

The *message* that **SetSelectionMessage()** assigns to the BListView is a model for the messages that the **Select()** function posts. **Select()** copies the model and posts the copy. It adds the same two pieces of information to the copy as are added to the invocation message:

Data name	Type code	Description
"source"	**B_OBJECT_TYPE**	A pointer to the BListView object.
"index"	**B_LONG_TYPE**	The index of the item that was selected.

You should not use these names for data you add to the model *message*.

See also: **Select()**, **SetInvocationMessage()**, the BMessage class

SetSymbolSet()

virtual void **SetSymbolSet(**const char **name***)**

Augments its BView counterpart to recalculate the layout of the list when the symbol set changes.

See also: **BView::SetSymbolSet()**

SetTarget(), Target()

virtual long **SetTarget(**BHandler **target***)**
virtual long **SetTarget(**BLooper **target*, bool *targetsPreferredHandler***)**

BHandler ***Target(**BLooper ***looper* = NULL**)** const

These functions set and return the object that's expected to handle messages the BListView posts (through its **Select()** and **Invoke()** functions).

The version of **SetTarget()** that takes a single argument sets the *target* BHandler object. It's successful only if it can also discern a BLooper object where the BListView can post messages so that they will be dispatched to that target. To post a message, the BListView calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
theLooper->PostMessage(theMessage, target);
```

Therefore, the *target* BHandler must either:

- Have been added to a BLooper, or
- Be a BLooper itself, so that it can fulfill the roles of both BLooper and BHandler.

Once it's set as the BListView's *target*, the BHandler must continue its association with the BLooper. If it moves to another BLooper, **PostMessage()** will fail.

The version of **SetTarget()** that takes two arguments sets the BLooper object where the BListView function should post messages. If the *targetsPreferredHandler* flag is **FALSE**, messages will be targeted to the *looper* object itself—it will also act as the handler. In other words, passing a BLooper and **FALSE** to the version of **SetTarget()** that takes two arguments accomplishes the same thing as simply passing the BLooper alone to the version that takes one argument. These two lines of code are equivalent:

```
myListView->SetTarget(someLooper, FALSE);
myListView->SetTarget(someLooper);
```

However, if the *targetsPreferredHandler* flag is **TRUE**, messages are targeted to the *looper*'s preferred handler (the object returned by its **PreferredHandler()** function). This permits the targeting decision to be made dynamically:

```
looper->PostMessage(theMessage, looper->PreferredHandler());
```

For a BWindow, the preferred handler is the current focus view. Therefore, by passing a BWindow *looper* and **TRUE** to **SetTarget()**,

```
myListView->SetTarget(someWindow, TRUE);
```

the BListView can be targeted to whatever BView happens to be in focus at the time an item is invoked. (Note, however, that if the *looper*'s **PreferredHandler()** is **NULL**, the BLooper itself becomes the target, just as it would if the *targetsPreferredHandler* flag were **FALSE**.)

When successful, **SetTarget()** returns **B_NO_ERROR**. It fails and returns **B_BAD_VALUE** if the proposed *target* or *looper* is **NULL**. The one-argument version also returns **B_BAD_VALUE** if it can't discover a BLooper from the target handler.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where the BListView will post messages. If the target BHandler is the preferred handler of the *looper*, **Target()** returns **NULL**. In other words, passing a BLooper and **TRUE** to **SetTarget()** causes **Target()** to report that there is a *looper*, but a **NULL** target; the BLooper is known, but the target BHandler is not. Passing a BLooper and **FALSE** to **SetTarget()** causes **Target()** to report that the same object is both *looper* and target.

By default (established by **AttachedToWindow()**), the BWindow where the list is located acts as both BLooper and BHandler.

See also: **BView::Looper()**, **BWindow::PreferredHandler()**, **Invoke()**, **AttachedToWindow()**

BMenu

Derived from:	public BView
Declared in:	<interface/Menu.h>

Overview

A BMenu object displays a pull-down or pop-up list of menu items. Menus organize the features of an application—the common ones as well as the more obscure—and provide users with points of entry for most everything the application can do.

Menus categorize the features of the application—all formatting possibilities might be grouped in one menu, a list of documents in another, graphics choices in a third, and so on. The arrangement of menus presents an outline of how the various parts of the application fit together.

Menu Hierarchy

Menus are hierarchically arranged; an item in one menu can control another menu. The controlled menu is a *submenu*; the menu that contains the item that controls it is its *supermenu*. A submenu remains hidden until the user operates the item that controls it; it becomes hidden again when the user is finished with it. A submenu can have its own submenus, and those submenus can have submenus of their own, and so on—although it becomes hard for users to find their way around in a menu hierarchy that becomes too deep.

The menu at the root of the hierarchy is displayed in a window as a list—perhaps a list of just one item. Since it, unlike other menus, doesn't have a controlling item, it must remain visible. A root menu is therefore a special kind of menu in that it behaves more like an ordinary view than do other menus, which stay hidden. Root menus should belong to the BMenuBar class, which is derived from BMenu. The typical root menu is a menu bar displayed across the top of a window (hence the name of the class).

Menu Items

Each item in a menu is a kind of BMenuItem object. An item can be marked (displayed with a check mark to its left), assigned a keyboard shortcut, enabled and disabled, and given a "trigger" character that the user can type to invoke the item when its menu is open on-screen.

Every item has a particular job to do. If an item controls a submenu, its job is to show the submenu on-screen and hide it again. All other items give instructions to the application. When invoked by the user, they post a BMessage object to a target BHandler. What the item does depends on the content of the BMessage and the BHandler's response to it.

The BMenu and BMenuItem classes share some functions that accomplish the same thing when called for a submenu or for the supermenu item that controls the submenu. For example, setting the target for a BMenu (**SetTarget()**) sets the target for each of its items. Disabling a submenu (**SetEnabled()**) is the same as disabling the item that controls it; the user will be able to bring the submenu to the screen, but none of its items will work. This, in effect, disables all items and menus in the branch of the menu hierarchy under the superitem.

Hook Functions

ScreenLocation()	Can be implemented to have the menu appear on-screen at some location other than the default.

Constructor and Destructor

BMenu()

public:

BMenu(const char **name*, menu_layout *layout* = B_ITEMS_IN_COLUMN**)**
BMenu(const char **name*, float *width*, float *height***)**

protected:

BMenu(BRect *frame*, const char **name*, ulong *resizingMode*, ulong *flags*,
menu_layout *layout*, bool *resizeToFit***)**

Initializes the BMenu object. The *name* of the object becomes the initial label of the supermenu item that controls the menu and brings it to the screen. (It's also the name that can be passed to BView's **FindView()** function.)

A new BMenu object doesn't contain any items; you need to call **AddItem()** to set up its contents.

A menu can arrange its items in any of three ways:

B_ITEMS_IN_COLUMN	The items are stacked vertically in a column, one on top of the other, as in a typical menu.
B_ITEMS_IN_ROW	The items are laid out horizontally in a row, from end to end, as in a typical menu bar.
B_ITEMS_IN_MATRIX	The items are arranged in a custom fashion, such as a matrix.

Either **B_ITEMS_IN_ROW** or the default **B_ITEMS_IN_COLUMN** can be passed as the *layout* argument to the public constructor. (A column is the default for ordinary menus; a row is the default for BMenuBars.) This version of the constructor isn't designed for **B_ITEMS_IN_MATRIX** layouts.

A BMenu object can arrange items that are laid out in a column or a row entirely on its own. The menu will be resized to exactly fit the items that are added to it.

However, when items are laid out in a custom matrix, the menu needs more help. First, the constructor must be informed of the exact *width* and *height* of the menu rectangle. The version of the constructor that takes these two parameters is designed just for matrix menus—it sets the layout to **B_ITEMS_IN_MATRIX**. Then, when items are added to the menu, the BMenu object expects to be informed of their precise positions within the specified area. The menu is *not* resized to fit the items that are added. Finally, when items in the matrix change, you must take care of any required adjustments in the layout yourself.

The protected version of the constructor is supplied for derived classes that don't simply devise different sorts of menu items or arrange them in a different way, but invent a different kind of menu. If the *resizeToFit* flag is **TRUE**, it's expected that the *layout* will be **B_ITEMS_IN_COLUMN** or **B_ITEMS_IN_ROW**. The menu will resize itself to fit the items that are added to it. If the layout is **B_ITEMS_IN_MATRIX**, the *resizeToFit* flag should be **FALSE**.

~BMenu()

virtual **~BMenu(**void**)**

Deletes all the items that were added to the menu and frees all memory allocated by the BMenu object. Deleting the items serves also to delete any submenus those items control and, thus, the whole branch of the menu hierarchy.

Member Functions

AddItem()

> bool **AddItem(**BMenuItem **item***)**
> bool **AddItem(**BMenuItem **item*, long *index***)**
> bool **AddItem(**BMenuItem **item*, BRect *frame***)**
> bool **AddItem(**BMenu **submenu***)**
> bool **AddItem(**BMenu **submenu*, long *index***)**
> bool **AddItem(**BMenu **submenu*, BRect *frame***)**

Adds an item to the menu list at *index*—or, if no *index* is mentioned, to the end of the list. If items are arranged in a matrix rather than a list, it's necessary to specify the item's *frame* rectangle—the exact position where it should be located in the menu view. Assume a coordinate system for the menu that has the origin, (0.0, 0.0), at the left top corner of the view rectangle. The rectangle will have the width and height that were specified when the menu was constructed.

The versions of this function that take an *index* (even an implicit one) can be used only if the menu arranges items in a column or row (**B_ITEMS_IN_COLUMN** or **B_ITEMS_IN_ROW**); it's an error to use them for items arranged in a matrix. Conversely, the versions of this function that take a *frame* rectangle can be used only if the menu arranges items in a matrix (**B_ITEMS_IN_MATRIX**); it's an error to use them for items arranged in a list.

If a *submenu* is specified rather than an *item*, **AddItem()** constructs a controlling BMenuItem for the submenu and adds the item to the menu.

If it's unable to add the item to the menu—for example, if the *index* is out-of-range or the wrong version of the function has been called—**AddItem()** returns **FALSE**. If successful, it returns **TRUE**.

See also: the BMenu constructor, the BMenuItem class, **RemoveItem()**

AddSeparatorItem()

> bool **AddSeparatorItem(**void**)**

Creates an instance of the BSeparatorItem class and adds it to the end of the menu list, returning **TRUE** if successful and **FALSE** if not (a very unlikely possibility). This function is a shorthand for:

```
BSeparatorItem *separator = new BSeparatorItem;
AddItem(separator);
```

A separator serves only to separate other items in the list. It counts as an item and has an indexed position in the list, but it doesn't do anything. It's drawn as a horizontal line across the menu. Therefore, it's appropriately added only to menus where the items are laid out in a column.

See also: **AddItem()**, the BSeparatorItem class

AreTriggersEnabled() *see* SetTriggersEnabled()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Finishes initializing the BMenu object by setting graphics parameters and laying out items. This function is called for you each time the BMenu is assigned to a window. For a submenu, that means each time the menu is shown on-screen.

See also: **BView::AttachedToWindow()**

CountItems()

long **CountItems(**void**)** const

Returns the total number of items in the menu, including separator items.

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the menu. This function is called for you whenever the menu is placed on-screen or is updated while on-screen. It's not a function you need to call yourself.

See also: **BView::Draw()**

FindItem()

BMenuItem ***FindItem(**const char **label***)** const
BMenuItem ***FindItem(**ulong *command***)** const

Returns the item with the specified *label*—or the one that posts a message with the specified *command*. If there's more than one item in the menu hierarchy with that particular *label* or associated with that particular *command*, this function returns the first one it finds. It recursively searches the menu by working down the list of items in order. If an item controls a submenu, it searches the submenu before returning to check any remaining items in the menu.

If none of the items in the menu hierarchy meet the stated criterion, **FindItem()** returns **NULL**.

FindMarked()

BMenuItem ***FindMarked(**void**)**

Returns the first marked item in the menu list (the one with the lowest index), or **NULL** if no item is marked.

See also: **SetRadioMode()**, **BMenuItem::SetMarked()**

Hide(), Show()

protected:

void **Hide(**void**)**

void **Show(**bool *selectFirst***)**
virtual void **Show(**void**)**

These functions hide the menu (remove the BMenu view from the window it's in and remove the window from the screen) and show it (attach the BMenu to a window and place the window on-screen). If the *selectFirst* flag passed to **Show()** is **TRUE**, the first item in the menu will be selected when it's shown. If *selectFirst* is **FALSE**, the menu is shown without a selected item.

The version of **Show()** that doesn't take an argument simply calls the version that does and passes it a *selectFirst* value of **FALSE**.

These functions are not ones that you'd ordinarily call, even when implementing a derived class. You'd need them only if you're implementing a nonstandard menu of some kind and want to control when the menu appears on-screen.

See also: **BView::Show()**, **Track()**

IndexOf()

long **IndexOf(**BMenuItem **item***)** const
long **IndexOf(**BMenu **submenu***)** const

Returns the index of the specified menu *item*—or the item that controls the specified *submenu*. Indices record the position of the item in the menu list. They begin at 0 for the item at the top of a column or at the left of a row and include separator items.

If the menu doesn't contain the specified *item*, or the item that controls *submenu*, the return value will be **B_ERROR**.

See also: **AddItem()**

InvalidateLayout()

void **InvalidateLayout(**void**)**

Forces the BMenu to recalculate the layout of all menu items and, consequently, its own size. It can do this only if the items are arranged in a row or a column. If the items are arranged in a matrix, it's up to you to keep their layout up-to-date.

All BMenu and BMenuItem functions that change an item in a way that might affect the overall menu automatically invalidate the menu's layout so it will be recalculated. For example, changing the label of an item might cause the menu to become wider (if it needs more room to accommodate the longer label) or narrower (if it no longer needs as much room as before).

Therefore, you don't need to call **InvalidateLayout()** after using a Kit function to change a menu or menu item; it's called for you. You'd call it only when making some other change to a menu.

See also: the BMenu constructor

IsEnabled() *see* SetEnabled()

IsLabelFromMarked() *see* SetLabelFromMarked()

IsRadioMode() *see* SetRadioMode()

ItemAt(), SubmenuAt()

BMenuItem ***ItemAt(**long *index***)** const

BMenu ***SubmenuAt(**long *index***)** const

These functions return the item at *index*—or the submenu controlled by the item at *index*. If there's no item at the index, they return **NULL**. **SubmenuAt()** is a shorthand for:

```
ItemAt(index)->Submenu()
```

It returns **NULL** if the item at *index* doesn't control a submenu.

See also: **AddItem()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Handles keyboard navigation through the menu. This function is called to respond to messages reporting key-down events. It should not be called from application code.

See also: **BView::KeyDown()**

Layout()

protected:
menu_layout **Layout(**void**)** const

Returns **B_ITEMS_IN_COLUMN** if the items in the menu are stacked in a column from top to bottom, **B_ITEMS_IN_ROW** if they're stretched out in a row from left to right, or **B_ITEMS_IN_MATRIX** if they're arranged in some custom fashion. By default BMenu items are arranged in a column and BMenuBar items in a row.

The layout is established by the constructor.

See also: the BMenu and BMenuBar constructors

RemoveItem()

BMenuItem ***RemoveItem(**long *index***)**
bool **RemoveItem(**BMenuItem **item***)**
bool **RemoveItem(**BMenu **submenu***)**

Removes the item at *index*, or the specified *item*, or the item that controls the specified *submenu*. Removing the item doesn't free it.

- If passed an *index*, this function returns a pointer to the item so you can free it. It returns a **NULL** pointer if the item couldn't be removed (for example, if the *index* is out-of-range).
- If passed an *item*, it returns **TRUE** if the item was in the list and could be removed, and **FALSE** if not.
- If passed a *submenu*, it returns **TRUE** if the submenu is controlled by an item in the menu and that item could be removed, and **FALSE** otherwise.

When an item is removed from a menu, it loses its target; the cached value is set to **NULL**. If the item controls a submenu, it remains attached to the submenu even after being removed.

See also: **AddItem()**

ScreenLocation()

protected:

virtual BPoint **ScreenLocation(**void**)**

Returns the point where the left top corner of the menu should appear when the menu is shown on-screen. The point is specified in the screen coordinate system.

This function is called each time a hidden menu (a submenu of another menu) is brought to the screen. It can be overridden in a derived class to change where the menu appears. For example, the BPopUpMenu class overrides it so that a pop-up menu pops up over the controlling item.

See also: the BPopUpMenu class

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenu if the *enabled* flag is **TRUE**, and disables it if *enabled* is **FALSE**. If the menu is a submenu, this enables or disables its controlling item, just as if **SetEnabled()** were called for that item. The controlling item is updated so that it displays its new state, if it happens to be visible on-screen.

Disabling a menu disables its entire branch of the menu hierarchy. All items in the menu, including those that control other menus, are disabled.

IsEnabled() returns **TRUE** if the BMenu, and every BMenu above it in the menu hierarchy, is enabled. It returns **FALSE** if the BMenu, or any BMenu above it in the menu hierarchy, is disabled.

See also: **BMenuItem::SetEnabled()**

SetLabelFromMarked(), IsLabelFromMarked()

protected:

void **SetLabelFromMarked(**bool *flag***)**

bool **IsLabelFromMarked(**void**)**

SetLabelFromMarked() determines whether the label of the item that controls the menu (the label of the superitem) should be taken from the currently marked item within the menu. If *flag* is **TRUE**, the menu is placed in radio mode and the superitem's label is reset each time the user selects a different item. If *flag* is **FALSE**, the setting for radio mode doesn't change and the label of the superitem isn't automatically reset.

IsLabelFromMarked() returns whether the superitem's label is taken from the marked item (but not necessarily whether the BMenu is in radio mode).

See also: **SetRadioMode()**

SetRadioMode(), IsRadioMode()

virtual void **SetRadioMode(**bool *flag***)**

bool **IsRadioMode(**void**)**

SetRadioMode() puts the BMenu in radio mode if *flag* is **TRUE** and takes it out of radio mode if *flag* is **FALSE**. In radio mode, only one item in the menu can be marked at a time. If the user selects an item, a check mark is placed in front of it automatically (you don't need to call BMenuItem's **SetMarked()** function; it's called for you). If another item was marked at the time, its mark is removed. Selecting a currently marked item retains the mark.

IsRadioMode() returns whether the BMenu is currently in radio mode. The default radio mode is **FALSE** for ordinary BMenus, but **TRUE** for BPopUpMenus.

SetRadioMode() doesn't change any of the items in the menu. If you want an initial item to be marked when the menu is put into radio mode, you must mark it yourself.

When **SetRadioMode()** turns radio mode off, it calls **SetLabelFromMarked()** and passes it an argument of **FALSE**—turning off the feature that changes the label of the menu's superitem each time the marked item changes. Similarly, when **SetLabelFromMarked()** turns on this feature, it calls **SetRadioMode()** and passes it an argument of **TRUE**—turning radio mode on.

See also: **BMenuItem::SetMarked()**, **SetLabelFromMarked()**

SetTargetForItems()

virtual long **SetTargetForItems(**BHandler **target***)**

This function is a convenience for assigning the same *target* BHandler to all the items in the menu. It works through the list of items in order, calling BMenuItem's **SetTarget()** virtual function for each one. If it's unable to set the target of any item, it aborts and returns the error it encountered. If successful in setting the *target* of all items, it returns **B_NO_ERROR**. See BMenuItem's **SetTarget()** for information on acceptable *target* values.

This function doesn't work recursively; it acts only on items currently in the BMenu, not on items that might be added later nor on items in submenus.

See also: **BMenuItem::SetTarget()**

SetTriggersEnabled(), AreTriggersEnabled()

virtual void **SetTriggersEnabled(**bool *flag***)**

bool **AreTriggersEnabled(**void**)** const

SetTriggersEnabled() enables the triggers for all items in the menu if *flag* is **TRUE** and disables them if *flag* is **FALSE**. **AreTriggersEnabled()** returns whether the triggers are currently enabled or disabled. They're enabled by default.

Triggers are displayed to the user only if they're enabled, and only when keyboard actions can operate the menu.

Triggers are appropriate for some menus, but not for others. **SetTriggersEnabled()** is typically called to initialize the BMenu when it's constructed, not to enable and disable triggers as the application is running. If triggers are ever enabled for a menu, they should always be enabled; if they're ever disabled, they should always be disabled.

See also: **BMenuItem::SetTrigger()**

Show() *see* Hide()

SubmenuAt() *see* ItemAt()

Superitem(), Supermenu()

BMenuItem ***Superitem(**void**)** const

BMenu ***Supermenu(**void**)** const

These functions return the supermenu item that controls the BMenu and the supermenu where that item is located. The supermenu could be a BMenuBar object. If the BMenu hasn't been made the submenu of another menu, both functions return **NULL**.

See also: **AddItem()**

Track()

protected:

BMenuItem ***Track(**bool *openAnyway* = FALSE, BRect **clickToOpenRect* = NULL**)**

Initiates tracking of the cursor within the menu. This function passes tracking control to submenus (and submenus of submenus) depending on where the user moves the mouse. If the user ends tracking by invoking an item, **Track()** returns the item. If the user didn't invoke any item, it returns **NULL**. The item doesn't have to be located in the BMenu; it could, for example, belong to a submenu of the BMenu.

If the *openAnyway* flag is **TRUE**, **Track()** opens the menu and leaves it open even though a mouse button isn't held down. This enables menu navigation from the keyboard. If a *clickToOpenRect* is specified and the user has set the click-to-open preference, **Track()** will leave the menu open if the user releases the mouse button while the cursor is inside the rectangle. The rectangle should be stated in the screen coordinate system.

Track() is called by the BMenu to initiate tracking in the menu hierarchy. You would need to call it yourself only if you're implementing a different kind of menu that starts to track the cursor under nonstandard circumstances.

BMenuBar

Derived from:	public BMenu
Declared in:	<interface/MenuBar.h>

Overview

A BMenuBar is a menu that can stand at the root of a menu hierarchy. Rather than appear on-screen when commanded to do so by a user action, a BMenuBar object has a settled location in a window's view hierarchy, just like other views. Typically, the root menu is the menu bar that's drawn across the top of the window. It's from this use that the class gets its name.

However, instances of this class can also be used in other ways. A BMenuBar might simply display a list of items arranged in a column somewhere in a window. Or it might contain just one item, where that item controls a pop-up menu (a BPopUpMenu object). Rather than look like a "menu bar," the BMenuBar object would look something like a button.

The Key Menu Bar

The "real" menu bar at the top of the window usually represents an extensive menu hierarchy; each of its items typically controls a submenu.

The user should be able to operate this menu bar from the keyboard (using the arrow keys and Enter). There are two ways that the user can put the BMenuBar and its hierarchy in focus for keyboard events:

- Clicking an item in the menu bar. If the "click to open" preference is not turned off, this opens the submenu the item controls so that it stays visible on-screen and puts the submenu in focus.
- Pressing the Menu key, or pressing and releasing a Command key. This puts the BMenuBar in focus and selects its first item.

Either method opens the entire menu hierarchy to keyboard navigation.

If a window's view hierarchy includes more than one BMenuBar object, the Menu key (or Command) must choose one of them to put in focus. By default, it picks the last one that was attached to the window. However, the **SetKeyMenuBar()** function defined in the BWindow class can be called to designate a different BMenuBar object as the "key" menu bar for the window.

A Kind of BMenu

BMenuBar inherits most of its functions from the BMenu class. It reimplements the **AttachedToWindow()**, **Draw()**, and **MouseDown()** functions that set up the object and respond to messages, but these aren't functions that you'd call from application code; they're called for you.

The only real function (other than the constructor) that the BMenuBar class adds to those it inherits is **SetBorder()**, which determines how the list of items is bordered.

Therefore, for most BMenuBar operations—adding submenus, finding items, temporarily disabling the menu bar, and so on—you must call inherited functions and treat the object like the BMenu that it is.

See also: the BMenu class

Constructor and Destructor

BMenuBar()

BMenuBar(BRect *frame*, const char **name*,
    ulong *resizingMode* =
        B_FOLLOW_LEFT_RIGHT | B_FOLLOW_TOP,
    menu_layout *layout* = B_ITEMS_IN_ROW,
    bool *resizeToFit* = TRUE**)**

Initializes the BMenuBar by assigning it a *frame* rectangle, a *name*, and a *resizingMode*, just like other BViews. These values are passed up the inheritance hierarchy to the BView constructor. The default resizing mode (**B_FOLLOW_LEFT_RIGHT** | **B_FOLLOW_TOP**) is designed for a true menu bar (one that's displayed along the upper edge of a window). It permits the menu bar to adjust itself to changes in the window's width, while keeping it glued to the top of the window frame.

The *layout* argument determines how items are arranged in the menu bar. By default, they're arranged in a row as befits a true menu bar. If an instance of this class is being used to implement something other than a horizontal menu, items can be laid out in a column (**B_ITEMS_IN_COLUMN**) or in a matrix (**B_ITEMS_IN_MATRIX**).

If the *resizeToFit* flag is turned on, as it is by default, the frame rectangle of the BMenuBar will be automatically resized to fit the items it displays. This is generally a good idea, since it relieves you of the responsibility of testing user preferences to determine what size the menu bar should be. Because the font and font size for menu items are user preferences, items can vary in size from user to user.

When *resizeToFit* is **TRUE**, the *frame* rectangle determines only where the menu bar is located, not how large it will be. The rectangle's **left** and **top** data members are respected, but the **right** and **bottom** sides are adjusted to accommodate the items that are added to the menu bar.

Two kinds of adjustments are made if the *layout* is **B_ITEMS_IN_ROW**, as it typically is for a menu bar:

- The height of the menu bar is adjusted to the height of a single item.
- If the *resizingMode* includes **B_FOLLOW_LEFT_RIGHT**, the width of the menu bar is adjusted to match the width of its parent view. This means that a true menu bar (one that's a child of the window's top view) will always be as wide as the window.

Two similar adjustments are made if the menu bar *layout* is **B_ITEMS_IN_COLUMN**:

- The width of the menu bar is adjusted to the width of the widest item.
- If the *resizingMode* includes **B_FOLLOW_TOP_BOTTOM**, the height of the menu bar is adjusted to match the height of its parent view.

After setting up the key menu bar and adding items to it, you may want to set the minimum width of the window so that certain items won't be hidden when the window is resized smaller.

Change the *resizingMode*, the *layout*, and the *resizeToFit* flag as needed for BMenuBars that are used for a purpose other than to implement a true menu bar.

See also: the BMenu constructor, **BWindow::SetSizeLimits()**

~BMenuBar()

virtual **~BMenuBar(**void**)**

Frees all the items and submenus in the entire menu hierarchy, and all memory allocated by the BMenuBar.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Finishes the initialization of the BMenuBar by setting up its graphics environment, and by making the BWindow to which it has become attached the target handler for all items in the menu hierarchy, except for those items for which a target has already been set.

This function also makes the BMenuBar the key menu bar, the BMenuBar object whose menu hierarchy the user can navigate from the keyboard. If a window contains more than one BMenuBar in its view hierarchy, the last one that's added to the window gets to keep this designation. However, the key menu bar should always be the real menu bar at the top of the window. It can be explicitly set with BWindow's **SetKeyMenuBar()** function.

See also: **BWindow::SetKeyMenuBar()**

Border() *see* SetBorder()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the menu—whether as a true menu bar, as some other kind of menu list, or as a single item that controls a pop-up menu. This function is called as the result of update messages; you don't need to call it yourself.

See also: **BView::Draw()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Initiates mouse tracking and keyboard navigation of the menu hierarchy. This function is called to notify the BMenuBar of a mouse-down event.

See also: **BView::MouseDown()**

SetBorder(), Border()

void **SetBorder(**menu_bar_border *border***)**

menu_bar_border **Border(**void**)** const

SetBorder() determines how the menu list is bordered. The *border* argument can be:

B_BORDER_FRAME	The border is drawn around the entire frame rectangle.
B_BORDER_CONTENTS	The border is drawn around just the list of items.
B_BORDER_EACH_ITEM	A border is drawn around each item.

Border() returns the current setting. The default is **B_BORDER_FRAME**.

BMenuField

Derived from: public BView

Declared in: <interface/MenuField.h>

Overview

A BMenuField object displays a labeled pop-up menu. It's a simple object that employs a BMenuBar object to control a BMenu. All it adds to what a BMenuBar can do on its own is a label and a more control-like user interface that includes keyboard navigation.

The functions defined in this class resemble those of a BControl (**SetLabel()**, **IsEnabled()**), especially a BTextControl (**SetDivider()**, **Alignment()**). However, unlike a real BControl object, a BMenuField doesn't maintain a current value and it can't be invoked or post messages. All the control work is done by items in the BMenu.

Constructor and Destructor

BMenuField()

BMenuField(BRect *frame*, const char **name*,
const char **label*,
BMenu **menu*,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = B_WILL_DRAW | B_NAVIGABLE**)**

Initializes the BMenuField object with the specified *frame* rectangle, *name*, *resizingMode*, and *flags*. These arguments are the same as for any BView object and are passed unchanged to the BView constructor. When the object is attached to a window, the height of its frame rectangle will be adjusted to fit the height of the text it displays, which depends on the user's preferred font for menus.

By default, the frame rectangle is divided horizontally in half, with the *label* displayed on the left and the *menu* on the right. This division can be changed with the **SetDivider()** function. The *menu* is assigned to a BMenuBar object and will pop up under the user's control. For most uses, the *menu* should be a BPopUpMenu object.

~BMenuField()

virtual **~BMenuField(**void**)**

Frees the label, the BMenuBar object, and other memory allocated by the BMenuField.

Member Functions

Alignment() *see* SetAlignment()

AttachedToWindow(), AllAttached()

virtual void **AttachedToWindow(**void**)**

virtual void **AllAttached(**void**)**

These functions override their BView counterparts to make the BMenuField's background color match the color of its parent view and to adjust the height of the view to the height of the BMenuBar child it contains. The height of the child depends on the size of the user's preferred font for menus.

See also: **BView::AttachedToWindow()**

Divider() *see* SetDivider()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Overrides the BView version of this function to draw the view's border and label. The way the menu field is drawn depends on whether it's enabled or disabled and whether or not it's the current focus for keyboard actions.

See also: **BView::Draw()**

IsEnabled() *see* SetEnabled()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the BView version of **KeyDown()** to permit keyboard navigation to and from the view and to allow users to open the menu by pressing the space bar.

See also: **BView::KeyDown()**

Label() *see* SetLabel()

MakeFocus()

virtual void **MakeFocus(**bool *focused***)**

Augments the BView version of **MakeFocus()** to enable keyboard navigation. This function calls **Draw()** when the BMenuField becomes the focus view and when it loses that status.

See also: **BView::MakeFocus()**

Menu(), MenuBar()

BMenu ***Menu(**void**)** const

BMenuBar ***MenuBar(**void**)** const

Menu() returns the BMenu object that pops up when the user operates the BMenuField; **MenuBar()** returns the BMenuBar object that contains the menu. The BMenuBar is created by the BMenuField; the menu is assigned to it during construction.

See also: the BMenuField constructor

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Overrides the BView version of **MouseDown()** to enable users to pop up the menu using the mouse, even if the cursor isn't directly over the menu portion of the bounds rectangle.

See also: **BView::MouseDown()**

SetAlignment(), Alignment()

virtual void **SetAlignment(**alignment *label***)**

alignment **Alignment(**void**)** const

These functions set and return the alignment of the label in its portion of the frame rectangle.

B_ALIGN_LEFT	The label is aligned at the left side of the bounds rectangle.
B_ALIGN_RIGHT	The label is aligned at the right boundary of its portion of the bounds rectangle.
B_ALIGN_CENTER	The label is centered in its portion of the bounds rectangle.

The default is **B_ALIGN_LEFT**.

SetDivider(), Divider()

virtual void **SetDivider(**float *xCoordinate*)

float **Divider(**void**)** const

These functions set and return the *x* coordinate value that divides the bounds rectangle between the label's portion on the left and the portion that holds the menu on the right. The coordinate is expressed in the BMenuField's coordinate system.

The default divider splits the bounds rectangle in two equal sections. By resetting it, you can provide more or less room for the label or the menu.

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenuField if the *enabled* flag is **TRUE**, and disables it if the flag is **FALSE**. **IsEnabled()** returns whether or not the object is currently enabled. When disabled, the BMenuField doesn't respond to mouse and keyboard manipulations.

If the *enabled* flag changes the current state of the object, **SetEnabled()** causes the view to be redrawn, so that its new state can be displayed to the user.

SetLabel(), Label()

virtual void **SetLabel(**const char **string*)

const char ***Label(**void**)** const

SetLabel() frees the current label and, if the argument it's passed is not **NULL**, replaces it with a copy of *string*. **Label()** returns the current label. The string it returns belongs to the BMenuField object.

See also: the BMenuField constructor

BMenuItem

Derived from:	public BObject
Declared in:	<interface/MenuItem.h>

Overview

A BMenuItem is an object that contains and displays one item within a menu. By default, Menu items are displayed simply as textual labels, like “Options...” or “Save As”. Derived classes can be defined to draw something other than a label—or something in addition to the label.

Kinds of Items

Some menu items play a role in helping users navigate the menu hierarchy. They give the user access to submenus. A submenu remains hidden until the user operates the item that controls it.

Other items accomplish specific actions. When the user invokes the item, a message is posted so that it will be delivered to a target BHandler, usually the window where the menu at the root of the hierarchy (a BMenuBar object) is displayed. The action that the item initiates, or the state that it sets, depends entirely on the message and the handler’s response to it.

The target handler and the message can be customized for every item. Each BMenuItem retains a model for the BMessage it posts and can have a target that’s different from other items in the same menu.

Items can also have a visual presence, but do nothing. Instances of the BSeparatorItem class, which is derived from BMenuItem, serve only to visually separate groups of items in the menu.

Shortcuts and Triggers

Any menu item (except for those that control submenus) can be associated with a keyboard shortcut, a character the user can type in combination with a Command key (and possibly other modifiers) to invoke the item. The shortcut character is displayed in the menu item to the right of the label. All shortcuts for menu items require the user to hold down the Command key.

A shortcut works even when the item it invokes isn't visible on-screen. It, therefore, has to be unique within the window (within the entire menu hierarchy).

Every menu item is also associated with a *trigger*, a character that the user can type (without the Command key) to invoke the item. The trigger works only while the menu is both open on-screen and can be operated using the keyboard. It therefore must be unique only within a particular branch of the menu hierarchy (within the menu).

The trigger is one of the characters that's displayed within the item—either the keyboard shortcut or a character in the label. When it's possible for the trigger to invoke the item, the character is underlined. Like shortcuts, triggers are case-insensitive.

For an item to have a keyboard shortcut, the application must explicitly assign one. However, by default, the Interface Kit chooses and assigns triggers for all items. The default choice can be altered by the **SetTrigger()** function.

Marked Items

An item can also be marked (with a check mark drawn to the left of the label) in order to indicate that the state it sets is currently in effect. Items are marked by the **SetMarked()** function. A menu can be set up so that items are automatically marked when they're selected and exactly one item is marked at all times. (See **SetRadioMode()** in the BMenu class.)

Disabled Items

Items can also be enabled or disabled (by the **SetEnabled()** function). A disabled item is drawn in muted tones to indicate that it doesn't work. It can't be selected or invoked. If the item controls a specific action, it won't post the message that initiates the action. If it controls a submenu, it will still bring the submenu to the screen, but all the items in submenu will be disabled. If an item in the submenu brings its own submenu to the screen, items in that submenu will also be disabled. Disabling the superitem for a submenu in effect disables a whole branch of the menu hierarchy.

See also: the BMenu class, the BSeparatorItem class

Hook Functions

All BMenuItem hook functions are protected. They should be implemented only if you design a special type of menu item that displays something other than a textual label.

Draw()	Draws the entire item; can be reimplemented to draw the item in a different way.

DrawContents()	Draws the item label; can be reimplemented to draw something other than a label.
GetContentSize()	Provides the width and height of the item's content area, which is based on the length of the label and the current font; can be reimplemented to provide the size required to draw something other than a label.
Highlight()	Highlights the item when it's selected; can be reimplemented to do highlighting in some way other than the default.

Constructor and Destructor

BMenuItem()

BMenuItem(const char **label*, BMessage **message*,
char *shortcut* = NULL, ulong *modifiers* = NULL**)**
BMenuItem(BMenu **submenu*, BMessage **message* = NULL**)**

Initializes the BMenuItem to display *label* (which can be **NULL** if the item belongs to a derived class that's designed to display something other than text) and assigns it a model *message* (which also can be **NULL**).

Whenever the user invokes the item, the model message is copied and the copy is posted and marked for delivery to the target handler. Three pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	The time the item was invoked, as measured in microseconds since the machine was last booted.
"source"	**B_OBJECT_TYPE**	A pointer to the BMenuItem object.
"index"	**B_LONG_TYPE**	The index of the item, its ordinal position in the menu. Indices begin at 0.

These names should not be used for any data that you place in the *message*.

By default, the target of the message is the window associated with the item's menu hierarchy—the window where the BMenuBar at the root of the hierarchy is located. Another target can be designated by calling the **SetTarget()** function.

The constructor can also optionally set a keyboard shortcut for the item. The character that's passed as the *shortcut* parameter will be displayed to the right of the item's label. It's the accepted practice to display uppercase shortcut characters only, even though the actual character the user types may not be uppercase.

The *modifiers* mask, not the *shortcut* character, determines which modifier keys the user must hold down for the shortcut to work—including whether the Shift key must be down. The mask can be formed by combining any of the modifiers constants, especially these:

B_SHIFT_KEY
B_CONTROL_KEY
B_OPTION_KEY
B_COMMAND_KEY

However, **B_COMMAND_KEY** is required for all keyboard shortcuts; it doesn't have to be explicitly included in the mask. For example, setting the *shortcut* to 'U' with no *modifiers* would mean that the letter 'U' would be displayed alongside the item label and Command-*u* would invoke the item. The same *shortcut* with a **B_SHIFT_KEY** *modifiers* mask would mean that the uppercase character (Command-Shift-*U*) would invoke the item.

If the BMenuItem is constructed to control a *submenu*, it can't take a shortcut and it typically doesn't post messages—its role is to bring up the submenu. However, it can be assigned a model *message* if the application must take some collateral action when the submenu is opened. The item's initial label will be taken from the name of the submenu. It can be changed after construction by calling **SetLabel()**.

See also: **SetTarget()**, **SetMessage()**, **SetLabel()**

~BMenuItem()

virtual **~BMenuItem(**void**)**

Frees the item's label and its model BMessage object. If the item controls a submenu, that menu and all its items are also freed. Deleting a BMenuItem destroys the entire menu hierarchy under that item.

Member Functions

Command() *see* SetMessage()

ContentLocation()

protected:
BPoint **ContentLocation(**void**)** const

Returns the left top corner of the content area of the item, in the coordinate system of the BMenu to which it belongs. The content area of an item is the area where it displays its label (or whatever graphic substitutes for the label). It doesn't include the part of the item where a check mark or a keyboard shortcut could be displayed, nor the border and background around the content area.

You would need to call this function only if you're implementing a **DrawContent()** function to draw the contents of the menu item (likely something other than a label). The content rectangle can be calculated from the point returned by this function and the size specified by **GetContentSize()**.

If the item isn't part of a menu, the return value is indeterminate.

See also: **GetContentSize()**, **DrawContent()**

Draw(), DrawContent()

protected:

virtual void **Draw(**void**)**

virtual void **DrawContent(**void**)**

These functions draw the menu item and highlight it if it's currently selected. They're called by the **Draw()** function of the BMenu where the item is located whenever the menu is required to display itself; they don't need to be called from within application code.

However, they can both be overridden by derived classes that display something other than a textual label. The **Draw()** function is called first. It draws the background for the entire item, then calls **DrawContent()** to draw the label within the item's content area. After **DrawContent()** returns, it draws the check mark (if the item is currently marked) and the keyboard shortcut (if any). It finishes by calling **Highlight()** if the item is currently selected.

Both functions draw by calling functions of the BMenu in which the item is located. For example:

```
void MyItem::DrawContent()
{
    . . .
    Menu()->DrawBitmap(image);
    . . .
}
```

A derived class can override either **Draw()**, if it needs to draw the entire item, or **DrawContent()**, if it needs to draw only within the content area. A **Draw()** function can find the frame rectangle it should draw within by calling the BMenuItem's **Frame()** function; a **DrawContent()** function can calculate the content area from the point returned by **ContentLocation()** and the dimensions provided by **GetContentSize()**.

When **DrawContent()** is called, the pen is positioned to draw the item's label and the high color is appropriately set. The high color may be a shade of gray, if the item is disabled, or black if it's enabled. If some other distinction is used to distinguish disabled from enabled items, **DrawContent()** should check the item's current state by calling **IsEnabled()**.

Note: If a derived class implements its own **DrawContent()** function, but still wants to draw a textual string, it should do so by assigning the string as the BMenuItem's label and calling the inherited version of **DrawContent()**, not by calling **DrawString()**. This preserves the BMenuItem's ability to display a trigger character in the string.

See also: **Highlight()**, **Frame()**, **ContentLocation()**, **GetContentSize()**

Frame()

BRect **Frame(**void**)** const

Returns the rectangle that frames the entire menu item, in the coordinate system of the BMenu to which the item belongs. If the item hasn't been added to a menu, the return value is indeterminate.

See also: **BMenu::AddItem()**

GetContentSize()

protected:
virtual void **GetContentSize(**float **width*, float **height***)**

Writes the size of the item's content area into the variables referred to by *width* and *height*. The content area of an item is the area where its label (or whatever substitutes for the label) is drawn.

A BMenu calls **GetContentSize()** for each of its items as it arranges them in a column or a row; the function is not called for items in a matrix. The information it provides helps determine where each item is located and the overall size of the menu.

GetContentSize() must report a size that's large enough to display the content of the item (and separate one item from another). By default, it reports an area just large enough to display the item's label. This area is calculated from the label and the BMenu's current font.

If you design a class derived from BMenuItem and implement your own **Draw()** or **DrawContent()** function, you should also implement a **GetContentSize()** function to report how much room will be needed to draw the item's contents.

See also: **DrawContent()**, **ContentLocation()**

Highlight()

protected:
virtual void **Highlight(**bool *flag***)**

Highlights the menu item when *flag* is **TRUE**, and removes the highlighting when *flag* is **FALSE**. Highlighting simply inverts all the colors in the item's frame rectangle (except for the check mark).

This function is called by the **Draw()** function whenever the item is selected and needs to be drawn in its highlighted state. There's no reason to call it yourself, unless you define your own version of **Draw()**. However, it can be reimplemented in a derived class, if items belonging to that class need to be highlighted in some way other than simple inversion.

See also: **Draw()**

IsEnabled() *see* SetEnabled()

isMarked() *see* SetMarked()

IsSelected()

protected:

bool **IsSelected(**void**)** const

Returns **TRUE** if the menu item is currently selected, and **FALSE** if not. Selected items are highlighted.

Label() *see* SetLabel()

Menu()

BMenu ***Menu(**void**)** const

Returns the menu where the item is located, or **NULL** if the item hasn't yet been added to a menu.

See also: **BMenu::AddItem()**

Message() *see* SetMessage()

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *enabled***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenuItem if the *enabled* flag is **TRUE**, disables it if *enabled* is **FALSE**, and updates the item if it's visible on-screen. If the item controls a submenu, this function calls the submenu's **SetEnabled()** virtual function, passing it the same flag. This ensures that the submenu is enabled or disabled as well.

IsEnabled() returns **TRUE** if the BMenuItem is enabled, its menu is enabled, and all menus above it in the hierarchy are enabled. It returns **FALSE** if the item is disabled or any objects above it in the menu hierarchy are disabled.

Items and menus are enabled by default.

When using these functions, keep in mind that:

- Disabling a BMenuItem that controls a submenu serves to disable the entire menu hierarchy under the item.
- Passing an argument of **TRUE** to **SetEnabled()** is not sufficient to enable the item if it's located in a disabled branch of the menu hierarchy. It can only undo a previous **SetEnabled()** call (with an argument of **FALSE**) on the same item.

See also: **BMenu::SetEnabled()**

SetLabel(), Label()

virtual void **SetLabel(**const char **string***)**

const char ***Label(**void**)** const

SetLabel() frees the item's current label and copies *string* to replace it. If the menu is visible on-screen, it will be redisplayed with the item's new label. If necessary, the menu will become wider (or narrower) so that it fits the new label.

The Interface Kit calls this virtual function to:

- Set the initial label of an item that controls a submenu to the name of the submenu, and
- Subsequently set the item's label to match the marked item in the submenu, if the submenu was set up to have this feature.

Label() returns a pointer to the current label.

See also: **BMenu::SetLabelFromMarked()**, the BMenuItem constructor

SetMarked(), IsMarked()

virtual void **SetMarked(**bool *flag***)**

bool **IsMarked(**void**)** const

SetMarked() adds a check mark to the left of the item label if *flag* is **TRUE**, or removes an existing mark if *flag* is **FALSE**. If the menu is visible on-screen. it's redisplayed with or without the mark.

IsMarked() returns whether the item is currently marked.

See also: **BMenu::SetLabelFromMarked()**, **BMenu::FindMarked()**

SetMessage(), Message(), Command()

virtual void **SetMessage(**BMessage **message***)**

BMessage ***Message(**void**)** const

ulong **Command(**void**)** const

SetMessage() makes *message* the model BMessage for the menu item, deleting any previous message assigned to the item. The model message is first set by the BMenuItem constructor; **SetMessage()** allows you to change the message in midstream. You might need to change it, for example, when the item's label changes. Passing a **NULL** *message* frees the current model BMessage object without replacing it.

When a menu item is invoked, its model message is copied, relevant information is added to the copy, and the copy is posted so that it will be dispatched to the target BHandler. (The information that gets added to the copy is described under the BMenuItem constructor.)

Message() returns a pointer to the BMenuItem's model message and **Command()** returns its **what** data member. If the BMenuItem doesn't post a message, both functions return **NULL**.

The BMessage that **Message()** returns belongs to the BMenuItem. You can modify it by adding and removing data, but you shouldn't delete it or do anything that will cause it to be deleted. In particular, you shouldn't post or send the message anywhere, since that would transfer ownership to a message loop and subject the message to automatic deletion.

It's possible to set and return a model BMessage for a separator item. However, the message will never be used.

See also: the BMenuItem constructor, **SetTarget()**

SetShortcut(), Shortcut()

virtual void **SetShortcut(**char *shortcut*, ulong *modifiers***)**

char **Shortcut(**ulong **modifiers* = NULL**)** const

SetShortcut() sets the *shortcut* character that's displayed at the right edge of the menu item and the set of *modifiers* that are associated with the character. These two arguments work just like the arguments passed to the BMenuItem constructor. See the constructor for a more complete description.

Shortcut() returns the character that's used as the keyboard shortcut for invoking the item, and writes a mask of all the modifier keys the shortcut requires to the variable referred to by *modifiers*. Since the Command key is required to operate the keyboard shortcut for any menu item, **B_COMMAND_KEY** will always be part of the *modifiers* mask. The mask can also be tested against the **B_CONTROL_KEY**, **B_OPTION_KEY**, and **B_SHIFT_KEY** constants.

The shortcut is initially set by the BMenuItem constructor.

See also: the BMenuItem constructor

SetTarget(), Target()

virtual long **SetTarget(**BHandler **target***)**
virtual long **SetTarget(**BLooper **target*, bool *targetsPreferredHandler***)**

BHandler ***Target(**BLooper ***looper* = NULL**)** const

These functions set and return the object that's targeted to handle messages posted by the BMenuItem.

The version of **SetTarget()** that takes a single argument sets the *target* BHandler object. It's successful only if it can also discern a BLooper object where the BMenuItem can post messages so that they will be dispatched to that target. To post a message, the BMenuItem calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
theLooper->PostMessage(theMessage, target);
```

Therefore, the *target* BHandler must be able, through its **Looper()** function, to reveal the BLooper object with which it is associated. It can do so if:

- It's a BLooper itself (such as a BWindow), so that it can fulfill the roles of both BLooper and BHandler.
- It has been added to a BLooper (as BViews are added to BWindows).

Once it becomes the BMenuItem's *target*, the BHandler must maintain its association with the BLooper. If it moves to another BLooper, **PostMessage()** will fail.

The version of **SetTarget()** that takes two arguments sets the BLooper object where the BMenuItem should post messages. If the *targetsPreferredHandler* flag is **FALSE**, messages will be targeted to the *looper* object itself—it will act both as BLooper and BHandler. In other words, passing a BLooper and **FALSE** to the version of **SetTarget()** that takes two arguments accomplishes the same thing as simply passing the BLooper alone to the version that takes one argument. These two lines of code have the same result:

```
myItem->SetTarget(someLooper, FALSE);
myItem->SetTarget(someLooper);
```

The two-argument version of **SetTarget()** becomes interesting only if the *targetsPreferredHandler* flag is **TRUE**. In this case, messages are targeted to the *looper*'s preferred handler (the object returned by its **PreferredHandler()** function). This permits the targeting decision to be made dynamically, when the user invokes the item:

```
    looper->PostMessage(theMessage, looper->PreferredHandler());
```

For example, the preferred handler for a BWindow object is the current focus view. Therefore, by passing a BWindow *looper* and **TRUE** to **SetTarget()**,

```
    myItem->SetTarget(someWindow, TRUE);
```

the menu item can be targeted to whatever BView happens to be in focus at the time the user operates the menu. This is useful for items—like Cut, Copy, and Paste—that act on the current selection. (Note, however, that if the *looper*'s **PreferredHandler()** is **NULL**, the BLooper itself becomes the target, just as it would if the *targetsPreferredHandler* flag were **FALSE**.)

When successful, **SetTarget()** returns **B_NO_ERROR**. It fails and returns **B_BAD_VALUE** if the proposed *target* or *looper* is **NULL**. The one-argument version also returns **B_BAD_VALUE** if it can't discover a BLooper from the proposed *target*.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where the BMenuItem will post messages. If the target BHandler is the preferred handler of the *looper*, **Target()** returns **NULL**. In other words, passing a BLooper and **TRUE** to **SetTarget()** causes **Target()** to report that there is a *looper*, but a **NULL** target; the BLooper is known, but the target BHandler is not. Passing a BLooper and **FALSE** to **SetTarget()** causes **Target()** to report that the same object is both *looper* and target.

By default, the BLooper and BHandler roles are both filled by the BWindow at the root of the menu hierarchy (the BWindow where the menu bar is located). These defaults are established when the BMenuItem becomes part of a menu hierarchy that's rooted in a window, but only if another *target* (or *looper*) hasn't already been set. If a target hasn't been set and the BMenuItem isn't part of a rooted menu hierarchy, **Target()** returns **NULL**.

See also: **BView::Looper()**, **BWindow::PreferredHandler()**

SetTrigger(), Trigger()

virtual void **SetTrigger(**char *trigger***)**

char **Trigger(**void**)** const

SetTrigger() sets the *trigger* character that the user can type to invoke the item while the item's menu is open on-screen. If a *trigger* is not set, the Interface Kit will select one for the item, so it's not necessary to call **SetTrigger()**.

The character passed to this function has to match a character displayed in the item—either the keyboard shortcut or a character in the label. The case of the character doesn't matter; lowercase arguments will match uppercase characters in the item and uppercase arguments will match lowercase characters. When the item can be invoked by its trigger, the trigger character is underlined.

If more than one character in the item matches the character passed, **SetTrigger()** tries first to mark the keyboard shortcut. Failing that, it tries to mark an uppercase letter at the beginning of a word. Failing that, it marks the first instance of the character in the label.

If the *trigger* doesn't match any characters in the item, the item won't have a trigger, not even one selected by the system.

Trigger() returns the character set by **SetTrigger()**, or **NULL** if **SetTrigger()** didn't succeed or if **SetTrigger()** was never called and the trigger is selected automatically.

See also: **BMenu::SetTriggersEnabled()**

Shortcut() *see* SetShortcut()

Submenu()

BMenu ***Submenu(**void**)** const

Returns the BMenu object that the item controls, or **NULL** if the item doesn't control a submenu.

See also: the BMenuItem constructor, the BMenu class

Target() *see* SetTarget()

Trigger() *see* SetTrigger()

BPicture

Derived from:	public BObject
Declared in:	<interface/Picture.h>

Overview

A BPicture object holds a set of drawing instructions in the Application Server, where they can be reused over and over again simply by passing the object to BView's **DrawPicture()** function. Because it contains instructions for producing an image, not the rendered result of those instructions, a picture (unlike a bitmap) is independent of the resolution of the display device.

Recording a Picture

Drawing instructions are captured by bracketing them with calls to a BView's **BeginPicture()** and **EndPicture()** functions. An empty BPicture object is passed to **BeginPicture()**; **EndPicture()** returns the same object, fully initialized. For example:

```
BPicture *myPict;
someView->BeginPicture(new BPicture);
/* drawing code goes here */
myPict = someView->EndPicture();
```

The BPicture object records all of the drawing instructions given to the BView following the **BeginPicture()** call and preceding the **EndPicture()** call. Only the drawing that the BView does is recorded; drawing done by children and other views attached to the window is ignored, as is everything except drawing code.

If the BPicture object passed to **BeginPicture()** isn't empty, the new drawing is appended to the code that's already in place.

The Picture Definition

The picture captures everything that affects the image that's drawn. It takes a snapshot of the BView's graphics parameters—the pen size, high and low colors, font size, and so on—at the time **BeginPicture()** is called. It then captures all subsequent modifications to those parameters, such as calls to **MovePenTo()**, **SetLowColor()**, and **SetFontSize()**. However, changes to the coordinate system (**ScrollBy()** and **ScrollTo()**) are ignored.

The picture records all primitive drawing instructions—such as, **DrawBitmap()**, **StrokeEllipse()**, **FillRect()**, and **DrawString()**. It can even include a call to **DrawPicture()**; one picture can incorporate another.

The BPicture traces exactly what BView drew and reproduces it precisely. For example, whatever pen size happens to be in effect when a line is stroked will be the pen size that the picture records, whether it was explicitly set while the BPicture was being recorded or assumed from the BView's graphics environment.

The picture makes its own copy of any data that's passed during the recording session. For example, it copies the bitmap passed to **DrawBitmap()** and the picture passed to **DrawPicture()**. If that bitmap or picture later changes, it won't affect what was recorded.

See also: **BView::BeginPicture()**, **BView::DrawPicture()**, the BPictureButton class

Constructor and Destructor

BPicture()

BPicture(void**)**
BPicture(const BPicture &*picture***)**
BPicture(void **data*, long *size***)**

Initializes the BPicture object by ensuring that it's empty, by copying data from another *picture*, or by copying *size* bytes of picture *data*. The data should be taken, directly or indirectly, from another BPicture object.

~BPicture()

virtual **~BPicture(**void**)**

Destroys the Application Server's record of the BPicture object and deletes all its picture data.

Member Functions

Data()

void ***Data(**void**)** const

Returns a pointer to the data contained in the BPicture. The data can be copied from the object, stored on disk (perhaps as a resource), and later used to initialize another BPicture object.

See also: the BPicture constructor

DataSize()

long **DataSize(**void**)** const

Returns how many bytes of data the BPicture object contains.

See also: **Data()**

BPictureButton

Derived from:	public BControl
Declared in:	<interface/PictureButton.h>

Overview

A BPictureButton object draws a button with a graphic image on its face, rather than a textual label. The image is set by a BPicture object.

Like other BControl objects, BPictureButtons can have two values, **B_CONTROL_OFF** and **B_CONTROL_ON**. A separate BPicture object is associated with each value. How the BPictureButton displays these pictures depends on its behavior—whether it's set to remain in one state or to toggle between two states:

- A one-state BPictureButton usually has a value of 0 (**B_CONTROL_OFF**), and it displays the BPicture associated with that value. However, while it's being operated (while the cursor is over the button on-screen and the user keeps the mouse button down), its value is set to 1 (**B_CONTROL_ON**) and it displays the alternate picture. That picture should be a highlighted version of the picture that's normally shown.

  This behavior is exactly like an ordinary, labeled BButton object. Just as a BButton displays the same label, a one-state BPictureButton shows the same picture. Both kinds of objects are appropriate devices for initiating an action of some kind.

- A two-state BPictureButton toggles between the **B_CONTROL_OFF** and **B_CONTROL_ON** values. Each time the user operates the button, it's value changes. The picture that's displayed changes with the value. The two BPictures are alternatives to each other. The **B_CONTROL_ON** picture might be a highlighted version of the **B_CONTROL_OFF** picture, but it doesn't need to be. The value of the object changes only after it has been toggled to the other state, not while it's being operated.

  This behavior is exactly like a BCheckBox or an individual BRadioButton. Like those objects, a two-state BPictureButton is an appropriate device for setting a state.

Every BPictureButton must be assigned at least two BPictures. If it's a one-state button, one picture will be the one that's normally shown and another will be shown while the button is being operated. If it's a two-state button, one picture is shown when the button is turned on and one when it's off.

If a one-state button can be disabled, it also needs to be assigned an image that can be shown while it's disabled. If a two-state button can be disabled, it needs two additional images—one in case it's disabled while in the **B_CONTROL_OFF** state and another if it's disabled in the **B_CONTROL_ON** state.

Often the BPictures that are assigned to a BPictureButton simply wrap around a bitmap image. For example:

```
BPicture *myPict;
someView->BeginPicture(new BPicture);
someView->DrawBitmap(&buttonBitmap);
myPict = someView->EndPicture();
```

See also: the BPicture class

Constructor and Destructor

BPictureButton()

BPictureButton(BRect *frame*, const char* *name*,
BPicture **off*,
BPicture **on*,
BMessage **message*,
ulong *behavior* = B_ONE_STATE_BUTTON,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = B_WILL_DRAW | B_NAVIGABLE**)**

Initializes the BPictureButton by assigning it two images—an *off* picture that will be displayed when the object's value is **B_CONTROL_OFF** and an *on* picture that's displayed when the value is **B_CONTROL_ON**—and by setting its *behavior* to either **B_ONE_STATE_BUTTON** or **B_TWO_STATE_BUTTON**. A one-state button displays the *off* image normally and the *on* image to highlight the button as it's being operated by the user. A two-state button toggles between the *off* image and the *on* image (between the **B_CONTROL_OFF** and **B_CONTROL_ON** values). The initial value is set to **B_CONTROL_OFF**.

If the BPictureButton can be disabled, it will need additional BPicture images that indicate its disabled state. They can be set by calling **SetDisabledOff()** and **SetDisabledOn()**.

All the BPictures assigned to the BPictureButton object become its property. It takes responsibility for deleting them when they're no longer needed.

The *message* parameter is the same as the one declared for the BControl constructor. It establishes a model for the messages the BPictureButton sends to a target object each time it's invoked. See **SetMessage()**, **SetTarget()**, and **Invoke()** in the BControl class for more information.

The *frame*, *name*, *resizingMode*, and *flags* parameters are the same as those declared for the BView constructor. They're passed up the inheritance hierarchy to the BView class unchanged. See the BView constructor for details.

See also: the BControl and BView constructors, **SetEnabledOff()**, **BControl::Invoke()**, **BControl::SetMessage()**, **BControl::SetTarget()**

~BPictureButton()

virtual **~BPictureButton(**void**)**

Deletes the model message and the BPicture objects that have been assigned to the BPictureButton.

Member Functions

Behavior() see SetBehavior()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the BPictureButton. This function is called as the result of an update message to draw the button in its current appearance; it's also called from the **MouseDown()** function to draw the button in its highlighted state.

See also: **BView::Draw()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the inherited version of **KeyDown()** to respond when *aChar* is **B_ENTER** or **B_SPACE**, by:

- Momentarily highlighting the button,
- Temporarily changing its value while it's being highlighted, and
- Posting a copy of the model BMessage to the target receiver.

< Note that this function matches the BButton **KeyDown()** function. It regards all BPictureButtons as being one-state buttons. >

See also: **BView::KeyDown()**, **BControl::Invoke()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event in the button by tracking the cursor while the user holds the mouse button down. If the BPictureButton is a one-state object, this function resets its value as the cursor moves in and out of the button on-screen. The **SetValue()** virtual function is called to make the change each time. If it's a two-state object, the value is not reset. < However, the picture corresponding to the **B_CONTROL_ON** value is shown while the cursor is in the button on-screen and the mouse button remains down. >

If the cursor is inside the BPictureButton's bounds rectangle when the user releases the mouse button, this function posts a copy of the model message so that it will be dispatched to the target handler. If it's a one-state object, it's value is reset to **B_CONTROL_OFF**. If it's a two-state object, it's value is toggled on or off and the corresponding picture is displayed.

See also: **BView::MouseDown()**, **BControl::Invoke()**, **SetBehavior()**

SetBehavior(), Behavior()

virtual void **SetBehavior(**ulong *behavior***)**

ulong **Behavior(**void**)** const

These functions set and return whether the BPictureButton is a **B_ONE_STATE_BUTTON** or a **B_TWO_STATE_BUTTON**. If it's a one-state button, its value is normally set to **B_CONTROL_OFF** and it displays a fixed image (the *off* picture passed to the constructor or the one passed to **SetEnabledOff()**). Its value is reset as its being operated and it displays the alternate image (the *on* picture passed to the constructor or the one passed to **SetEnabledOn()**).

If it's a two-state button, its value toggles between **B_CONTROL_OFF** and **B_CONTROL_ON** each time the user operates it. The image the button displays similarly toggles between two pictures (the *off* and *on* images passed to the constructor or the ones passed to **SetEnabledOff()** and **SetEnabledOn()**).

See also: the BPictureButton constructor

SetEnabledOff(), SetEnabledOn(), SetDisabledOff(), SetDisabledOn(), EnabledOff(), EnabledOn(), DisabledOff(), DisabledOn

virtual void **SetEnabledOff(**BPicture **picture***)**

virtual void **SetEnabledOn(**BPicture **picture***)**

virtual void **SetDisabledOff(**BPicture **picture***)**

virtual void **SetDisabledOn(**BPicture **picture***)**

inline BPicture ***EnabledOff(**void**)** const

inline BPicture ***EnabledOn(**void**)** const

inline BPicture ***DisabledOff(**void**)** const

inline BPicture ***DisabledOn(**void**)** const

These functions set and return the images the BPictureButton displays. Each BPictureButton object needs to be assigned at least two BPicture objects—one corresponding to the **B_CONTROL_OFF** value and another corresponding to the **B_CONTROL_ON** value. These are the images that are displayed when the BPictureButton is enabled, as it is by default. They're initially set when the object is constructed and can be replaced by calling the **SetEnabledOff()** and **SetEnabledOn()** functions.

If a BPictureButton can be disabled, it needs to display an image that indicates its disabled condition. A two-state button might be disabled when its value is either **B_CONTROL_OFF** or **B_CONTROL_ON**, so it needs two BPictures to indicate disabling, one corresponding to each value. They can be set by calling **SetDisabledOff()** and **SetDisabledOn()**.

The value of a one-state button is always **B_CONTROL_OFF** (except when it's being operated), so it needs only a single BPicture to indicate disabling; you can set it by calling **SetDisabledOff()**.

All four of the **Set**...**()** functions free the image previously set, if any, and replace it with *picture*. The *picture* belongs to the BPictureButton; it should not be freed or assigned to any other object.

The last four functions listed above return the BPictureButton's four images, or **NULL** if it hasn't been assigned a BPicture object in the requested category.

See also: the BPictureButton constructor

BPoint

Derived from:	*none*
Declared in:	<interface/Point.h>

Overview

BPoint objects represent points on a two-dimensional coordinate grid. Each object holds an *x* coordinate value and a *y* coordinate value declared as public data members. These values locate a specific point, (*x*, *y*), relative to a given coordinate system.

Because the BPoint class defines a basic data type for graphic operations, its data members are publicly accessible and it declares no virtual functions. It's a simple class that doesn't inherit from BObject or any other class and doesn't retain class information that it can reveal at run time. In the Interface Kit, BPoint objects are typically passed and returned by value, not through pointers.

For an introduction to coordinate geometry on the BeBox, see "The Coordinate Space" on page 14.

Data Members

float **x**	The coordinate value measured horizontally along the *x*-axis.
float **y**	The coordinate value measured vertically along the *y*-axis.

Constructor

BPoint()

inline **BPoint(**float *x*, float *y***)**
inline **BPoint(**const BPoint& *point***)**
inline **BPoint(**void**)**

Initializes a new BPoint object to (*x*, *y*), or to the same values as *point*. For example:

```
BPoint somePoint(155.7, 336.0);
BPoint anotherPoint(somePoint);
```

Here, both *somePoint* and *anotherPoint* are initialized to (155.7, 336.0).

If no coordinate values are assigned to the BPoint when it's declared,

```
BPoint emptyPoint;
```

its initial values are indeterminate.

BPoint objects can also be initialized or modified using the **Set()** function,

```
emptyPoint.Set(155.7, 336.0);
anotherPoint.Set(221.5, 67.8);
```

or the assignment operator:

```
somePoint = anotherPoint;
```

See also: **Set()**, the assignment operator

Member Functions

ConstrainTo()

void **ConstrainTo(**BRect *rect***)**

Constrains the point so that it lies inside the *rect* rectangle. If the point is already contained in the rectangle, it remains unchanged. However, if it falls outside the rectangle, it's moved to the nearest edge. For example, this code

```
BPoint point(54.9, 76.3);
BRect rect(10.0, 20.0, 40.0, 80.0);
point.Constrain(rect);
```

modifies the point to (40.0, 76.3).

See also: **BRect::Contains()**

PrintToStream()

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

Prints the contents of the BPoint object to the standard output stream (**stdout**) in the form:

```
"BPoint(x, y)"
```

where *x* and *y* stand for the current values of the BPoint's data members.

Set()

inline void **Set(**float *x*, float *y***)**

Assigns the coordinate values *x* and *y* to the BPoint object. For example, this code

```
BPoint point;
point.Set(27.0, 53.4);
```

is equivalent to:

```
BPoint point;
point.x = 27.0;
point.y = 53.4;
```

See also: the BPoint constructor

Operators

= (assignment)

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

Assigns the *x* and *y* values of one BPoint object to another BPoint:

```
BPoint a, b;
a.Set(21.5, 17.0);
b = a;
```

Point *b*, like point *a*, is set to (21.5, 17.0).

== (equality)

bool **operator ==(**const BPoint&**)** const

Compares the data members of two BPoint objects and returns **TRUE** if each one exactly matches its counterpart in the other object, and **FALSE** if not. In the following example, the equality operator would return **FALSE**:

```
BPoint a(21.5, 17.0);
BPoint b(17.5, 21.0);
if ( a == b )
    . . .
```

!= (inequality)

bool **operator !=(**const BPoint&**)** const

Compares two BPoint objects and returns **TRUE** unless their data members match exactly (the two points are the same), in which case it returns **FALSE**. This operator is the inverse of the **==** (equality) operator.

+ (addition)

BPoint **operator +(**const BPoint&**)** const

Combines two BPoint objects by adding the *x* coordinate of the second to the *x* coordinate of the first and the *y* coordinate of the second to the *y* coordinate of the first, and returns a BPoint object that holds the result. For example:

```
BPoint a(77.0, 11.0);
BPoint b(55.0, 33.0);
BPoint c = a + b;
```

Point *c* is initialized to (132.0, 44.0).

+= (addition and assignment)

BPoint& **operator +=(**const BPoint&**)**

Modifies a BPoint object by adding another point to it. As in the case of the **+** (addition) operator, the members of the second point are added to their counterparts in the first point:

```
BPoint a(77.0, 11.0);
BPoint b(55.0, 33.0);
a += b;
```

Point *a* is modified to (132.0, 44.0).

– (subtraction)

BPoint **operator –(**const BPoint&**)** const

Subtracts one BPoint object from another by subtracting the *x* coordinate of the second from the *x* coordinate of the first and the *y* coordinate of the second from the *y* coordinate of the first, and returns a BPoint object that holds the result. For example:

```
BPoint a(99.0, 66.0);
BPoint b(44.0, 88.0);
BPoint c = a - b;
```

Point *c* is initialized to (55.0, –22.0).

–= (subtraction and assignment)

BPoint& **operator –=(**const BPoint&**)**

Modifies a BPoint object by subtracting another point from it. As in the case of the **–** (subtraction) operator, the members of the second point are subtracted from their counterparts in the first point. For example:

```
BPoint a(99.0, 66.0);
BPoint b(44.0, 88.0);
a -= b;
```

Point *a* is modified to (55.0, –22.0).

BPolygon

Derived from:	public BObject
Declared in:	<interface/Polygon.h>

Overview

A BPolygon object represents a *polygon*—a closed, many-sided figure that describes an area within a two-dimensional coordinate system. It differs from a BRect object in that it can have any number of sides and the sides don't have to be aligned with the coordinate axes.

A BPolygon is defined as a series of connected points. Each point is a potential vertex in the polygon. An outline of the polygon could be constructed by tracing a straight line from the first point to the second, from the second point to the third, and so on through the whole series, then by connecting the first and last points if they're not identical.

The BView functions that draw a polygon—**StrokePolygon()** and **FillPolygon()**—take BPolygon objects as arguments. **StrokePolygon()** offers the option of leaving the polygon open—of not stroking the line that connects the first and last points in the list. The polygon therefore won't look like a polygon, but like an chain of lines fastened at their endpoints.

Constructor and Destructor

BPolygon()

BPolygon(BPoint **pointList*, long *numPoints***)**
BPolygon(const BPolygon **polygon***)**
BPolygon(void**)**

Initializes the BPolygon by copying *numPoints* from *pointList*, or by copying the list of points from another *polygon*. If one polygon is constructed from another, the original and the copy won't share any data; independent memory is allocated for the copy to hold a duplicate list of points.

If a BPolygon is constructed without a point list, points must be set with the **AddPoints()** function.

See also: **AddPoints()**

~BPolygon()

virtual **~BPolygon(**void**)**

Frees all the memory allocated to hold the list of points.

Member Functions

AddPoints()

void **AddPoints(**const BPoint **pointList*, long *numPoints***)**

Appends *numPoints* from *pointList* to the list of points that already define the polygon.

See also: the BPolygon constructor

CountPoints()

inline long **CountPoints(**void**)** const

Returns the number of points that define the polygon.

Frame()

inline BRect **Frame(**void**)** const

Returns the polygon's frame rectangle—the smallest rectangle that encloses the entire polygon.

MapTo()

void **MapTo(**BRect *source*, BRect *destination***)**

Modifies the polygon so that it fits the *destination* rectangle exactly as it originally fit the *source* rectangle. Each vertex of the polygon is modified so that it has the same proportional position relative to the sides of the destination rectangle as it originally had to the sides of the source rectangle.

The polygon doesn't have to be contained in either rectangle. However, to modify a polygon so that it's exactly inscribed in the destination rectangle, you should pass its frame rectangle as the source:

```
BRect frame = myPolygon->Frame();
myPolygon->MapTo(frame, anotherRect);
```

PrintToStream()

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

Prints the BPolygon's point list to the standard output stream (**stdout**). The BPoint version of this function is called to report each point as a string in the form

```
"BPoint(x, y)"
```

where *x* and *y* stand for the coordinate values of the point in question.

See also: **BPoint::PrintToStream()**

Operators

= (assignment)

BPolygon& **operator =(**const BPolygon&**)**

Copies the point list of one BPolygon object and assigns it to another BPolygon. After the assignment, the two objects describe the same polygon, but are independent of each other. Destroying one of the objects won't affect the other.

BPopUpMenu

Derived from:	public BMenu
Declared in:	<interface/PopUpMenu.h>

Overview

A BPopUpMenu is a specialized menu that's typically used in isolation, rather than as part of an extensive menu hierarchy. By default, it operates in radio mode—the last item selected by the user, and only that item, is marked in the menu.

A menu of this kind can be used to choose one from among a limited set of mutually exclusive states—to pick a paper size or paragraph style, for example, or to select a category of information. It should not be used to group different kinds of choices (as other menus may), nor should it include items that initiate actions rather than set states, except in certain well-defined cases.

A pop-up menu can be used in any of four ways:

- It can be controlled by a BMenuBar object, often one that contains just a single item. The BMenuBar, in effect, functions as a button that pops up a list. The label of the marked item in the list can be displayed as the label of the controlling item in the BMenuBar. In this way, the BMenuBar is able to show the current state of the hidden menu. When this is the case, the menu pops up so its marked item is directly over the controlling item.

- A BPopUpMenu can also be controlled by a view other than a BMenuBar. It might be associated with a particular image the view displays, for example, and appear over the image when the user moves the cursor there and presses the mouse button. Or it might be associated with the view as a whole and come up under the cursor wherever the cursor happens to be. When the view is notified of a mouse-down event, it calls BPopUpMenu's **Go()** function to show the menu on-screen.

- The BPopUpMenu might also be controlled by a particular mouse button, typically the secondary mouse button. When the user presses the button, the menu appears at the location of the cursor. Instead of passing responsibility for the mouse-down event to a BView, the BWindow would intercept it and place the menu on-screen.

- Finally, the application's main menu must be a BPopUpMenu object. This menu should be set up to behave like an ordinary menu, even though it's not included in an ordinary menu hierarchy. (The main menu is the one that holds items with application-wide significance, like "About . . ." and "Quit". It's accessible when the application is the active application by pressing on the application icon in the left top corner of the screen. See **SetMainMenu()** in the BApplication class.)

Other than **Go()** (and the constructor), this class implements no functions that you'd ever need to call from application code. In all other respects, a BPopUpMenu can be treated like any other BMenu.

Constructor and Destructor

BPopUpMenu()

BPopUpMenu(const char **name*, bool *radioMode* = TRUE,
bool *labelFromMarked* = TRUE,
menu_layout *layout* = B_ITEMS_IN_COLUMN**)**

Initializes the BPopUpMenu object. If the object is added to a BMenuBar, its *name* also becomes the initial label of its controlling item (just as for other BMenus).

If the *labelFromMarked* flag is **TRUE** (as it is by default), the label of the controlling item will change to reflect the label of the item that the user last selected. In addition, the menu will operate in radio mode (regardless of the value passed as the *radioMode* flag). When the menu pops up, it will position itself so that the marked item appears directly over the controlling item in the BMenuBar.

If *labelFromMarked* is **FALSE**, the menu pops up < so that its first item is over the controlling item >.

If the *radioMode* flag is **TRUE** (as it is by default), the last item selected by the user will always be marked. In this mode, one and only one item within the menu can be marked at a time. If *radioMode* is **FALSE**, items aren't automatically marked or unmarked.

However, the *radioMode* flag has no effect unless the *labelFromMarked* flag is **FALSE**. As long as *labelFromMarked* is **TRUE**, radio mode will also be **TRUE**.

The BPopUpMenu that's used as the application's main menu should have both *labelFromMarked* and *radioMode* set to **FALSE**.

The *layout* of the items in a BPopUpMenu can be either **B_ITEMS_IN_ROW** or the default **B_ITEMS_IN_COLUMN**. It should never be **B_ITEMS_IN_MATRIX**. The menu is resized so that it exactly fits the items that are added to it.

The new BPopUpMenu is empty; you add items to it by calling BMenu's **AddItem()** function.

See also: **BMenu::SetRadioMode()**, **BMenu::SetLabelFromMarked()**

~BPopUpMenu()

virtual **~BPopUpMenu(**void**)**

Does nothing. The BMenu destructor is sufficient to clean up after a BPopUpMenu.

Member Functions

Go()

BMenuItem ***Go(**BPoint *screenPoint*,
bool *deliversMessage* = FALSE,
bool *openAnyway* = FALSE**)**

BMenuItem ***Go(**BPoint *screenPoint*,
bool *deliversMessage*,
bool *openAnyway*,
BRect *clickToOpenRect***)**

Places the pop-up menu on-screen so that its left top corner is located at *screenPoint* in the screen coordinate system. **Go()** doesn't return until the user dismisses the menu from the screen. If the user invoked an item in the menu, it returns a pointer to the item. If no item was invoked, it returns **NULL**.

Go() is typically called from within the **MouseDown()** function of a BView. For example:

```
void MyView::MouseDown(BPoint point)
{
    BMenuItem *selected;
    BMessage *copy;
    . . .
    ConvertToScreen(&point);
    selected = myPopUp->Go(point);
    . . .
    if ( selected ) {
        BLooper *looper;
        BHandler *target = selected->Target(&looper);
        if ( target == NULL )
            target = looper->PreferredHandler();
        copy = new BMessage(selected->Message());
        looper->PostMessage(copy, target);
    }
    . . .
}
```

Go() operates in two modes:

- If the *deliversMessage* flag is **TRUE**, the BPopUpMenu works just like a menu that's controlled by a BMenuBar. When the user invokes an item in the menu, the item posts a message to its target.

- If the *deliversMessage* flag is **FALSE**, a message is not posted. Invoking an item doesn't automatically accomplish anything. It's up to the application to look at the returned BMenuItem and decide what to do. It can mimic the behavior of other menus and post the message—as shown in the example above—or it can take some other course of action.

In the example, a copy of the BMessage returned by the item's **Message()** function was posted, not the returned message itself. Posting the returned message would turn it over to a message loop, which would eventually delete it. It would then be unavailable the next time the item was invoked.

Go() always puts the pop-up menu on-screen, but ordinarily keeps it there only as long as the user holds a mouse button down. When the user releases the button, the menu is hidden and **Go()** returns. However, the *openAnyway* flag and the *clickToOpenRect* arguments can alter this behavior so that the menu will stay open even when the user releases the mouse button (or even if a mouse button was never down). It will take another user action—such as invoking an item in the menu or clicking elsewhere—to dismiss the menu.

If the *openAnyway* flag is **TRUE**, **Go()** keeps the menu on-screen even if no mouse buttons are held down. This permits a user to open and operate a pop-up menu from the keyboard. If *openAnyway* is **FALSE**, mouse actions determine whether the menu stays on-screen.

If the user has the click-to-open menu preference turned on and releases the mouse button while the cursor lies inside the *clickToOpenRect* rectangle, **Go()** interprets the action as clicking to open the menu and keeps it on-screen. If the cursor is outside the rectangle when the mouse button goes up, the menu is removed from the screen and **Go()** returns. The rectangle should be stated in the screen coordinate system.

See also: **BMenuItem::SetMessage()**

ScreenLocation()

protected:
    virtual BPoint **ScreenLocation(**void**)**

Determines where the pop-up menu should appear on-screen (when it's being run automatically, not by **Go()**). As explained in the description of the class constructor, this largely depends on whether the label of the superitem changes to reflect the item that's currently marked in the menu. The point returned is stated in the screen coordinate system.

This function is called only for BPopUpMenus that have been added to a menu hierarchy (a BMenuBar). You should not call it to determine the point to pass to **Go()**. However, you can override it to change where a customized pop-up menu defined in a derived class appears on-screen when it's controlled by a BMenuBar.

See also: **BMenu::SetLabelFromMarked()**, **BMenu::ScreenLocation()**, the BPopUpMenu constructor

BPrintJob

Derived from:	public BObject
Declared in:	<interface/PrintJob.h>

Overview

A BPrintJob object runs a printing session. It negotiates everything after the user's initial request to print—from engaging the Print Server to formatting pages, calling upon BViews to draw, and spooling the results to the printer.

A print job begins when the user requests the application to print something. In response, the application should create a BPrintJob object, assign the job a name, and call **InitJob()** to initialize the printing environment. For example:

```
void MyDocumentManager::Print()
{
    BPrintJob *job = new BPrintJob("document");
    if ( job->InitJob() < B_NO_ERROR )
        goto end;
    else {
        . . .
    }
    . . .
end:
    delete job;
    return;
}
```

InitJob() has the Print Server interact with the user to set up the parameters for the job—the number of copies, the size of the paper, scaling, orientation on the page, and so on.

You may want to store the user's choices with the document so that they can be used to set the initial configuration for the job when the document is next printed. By calling

Config(), you can get the job configuration the user set up; **SetConfig()** initializes the configuration that's presented to the user. For example:

```
BMessage *configuration;
. . .
void MyDocumentManager::Print()
{
    BPrintJob *job = new BPrintJob("document");
    if ( configuration )
        job->SetConfig(configuration);
    if ( job->InitJob() < B_NO_ERROR )
        goto end;
    if ( job->CanContinue() ) {
        if ( configuration )
            delete configuration;
        configuration = job->Config();
    }
    else
        goto end;
    . . .
}
```

A number of things can happen to derail a print job after it has started—most significantly, the user can cancel it at any time. To be sure that the job hasn't been canceled or something else hasn't happened to defeat it, you can call **CanContinue()** at critical junctures in your code, as illustrated above. This function will tell you whether it's sensible to continue with the job.

The next step after initializing the job is to call **BeginPrinting()** to set up a spool file and begin the production of pages. After all the pages are produced, **Commit()** is called to commit them to the printer.

```
job->BeginPrinting();
/* draw pages here */
job->Commit();
```

BeginPrinting() and **Commit()** bracket all the drawing that's done during the job.

Each page is produced by asking one or more BViews to draw within the page's printable rectangle (the rectangle that excludes the unprinted margin around the edge of the paper). You can call **DrawView()** any number of times for a single page to ask any number of

BViews to contribute to the page. After all views have drawn, the page is spooled to the file that will eventually be committed to the printer. For example:

```
for ( . . . ) {
    if ( job->CanContinue() ) {
        job->DrawView(someView, viewRect, pointOnPage);
        job->DrawView(anotherView, anotherRect, differentPoint);
        . . .
        job->SpoolPage();
    }
    else
        goto end;
}
```

DrawView() calls the BView's **Draw()** function. That function can test whether it's drawing on the screen or on the printed page by calling the BView **IsPrinting()** function. **SpoolPage()** is called just once for each page.

< This is the first release of the printing API; it will be enhanced in future releases to provide greater control over printing parameters. >

See also: **BView::IsPrinting()**

Constructor and Destructor

BPrintJob()

BPrintJob(char **name***)**

Initializes the BPrintJob object and assigns the job a *name*. The Print Server isn't contacted until **InitJob()** is called. The spool file isn't created until **BeginPrinting()** starts the production of pages.

See also: **InitJob()**, **BeginPrinting()**

~BPrintJob()

virtual **~BPrintJob(**void**)**

Frees all memory allocated by the object.

Member Functions

BeginPrinting()

void **BeginPrinting(**void**)**

Opens a spool file for the job and prepares for the production of a series of pages. Call this function only once per printing session—just after initializing the job and just before drawing the first page.

See also: **Commit()**

CancelJob()

void **CancelJob(**void**)**

Cancels the print job programmatically and gets rid of the spool file. The job cannot be restarted; you must delete the BPrintJob object. Create a new object to renew printing.

CanContinue()

bool **CanContinue(**void**)**

Returns **TRUE** if there's no impediment to continuing with the print job, and **FALSE** if the user has canceled the job, the spool file has grown too big, or something else has happened to terminate printing. It's a good idea to liberally sprinkle **CanContinue()** queries throughout your printing code to make sure that the work you're about to do won't be wasted.

Commit()

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

Commits all spooled pages to the printer. This ends the print job; when **Commit()** returns, the BPrintJob object can be deleted. **Commit()** can be called only once per job.

See also: **BeginPrinting()**

Config() *see* SetConfig()

DrawView(), SpoolPage()

virtual void **DrawView(**BView **view*, BRect *rect*, BPoint *point***)**

void **SpoolPage(**void**)**

DrawView() calls upon a *view* to draw the *rect* portion of its display at *point* on the page. The *view*'s **Draw()** function will be called with *rect* passed as the update rectangle. The rectangle should be stated in the BView's coordinate system and it should be fashioned so that the view draws only in the page's printable rectangle. The *point* should be stated in a coordinate system that has the origin at the top left corner of the printable rectangle.

The *view* must be attached to a window; that is, it must be known to the Application Server. However, when printing, a BView can be asked to draw portions of its display that are not visible on-screen. Its drawing is not limited by the clipping region, its bounds rectangle, or the frame rectangles of ancestor views.

DrawView() doesn't look down the view hierarchy; it asks only the named *view* to draw, not any of its children. However, any number of BViews can draw on a page if they are subjects of separate **DrawView()** calls.

After all views have drawn and the page is complete, **SpoolPage()** adds it to the spool file. **SpoolPage()** must be called once to terminate each page.

See also: **PrintableRect()**, **BView::Draw()**

FirstPage(), LastPage()

long **FirstPage(**void**)**

long **LastPage(**void**)**

< These functions both currently return 0. >

InitJob()

long **InitJob(**void**)**

Engages the Print Server and initializes the job. If **SetConfig()** has been called to establish a recommended configuration for the job, this function will pass it to the Print Server so the Server can present it to the user. Otherwise, a default configuration will be used.

InitJob() returns **B_ERROR** if it has trouble communicating with the Server or if the job can't be established for any other reason. It returns **B_NO_ERROR** if all goes well.

See also: **SetConfig()**

LastPage() *see* FirstPage()

PaperRect(), PrintableRect()

BRect **PaperRect(**void**)**

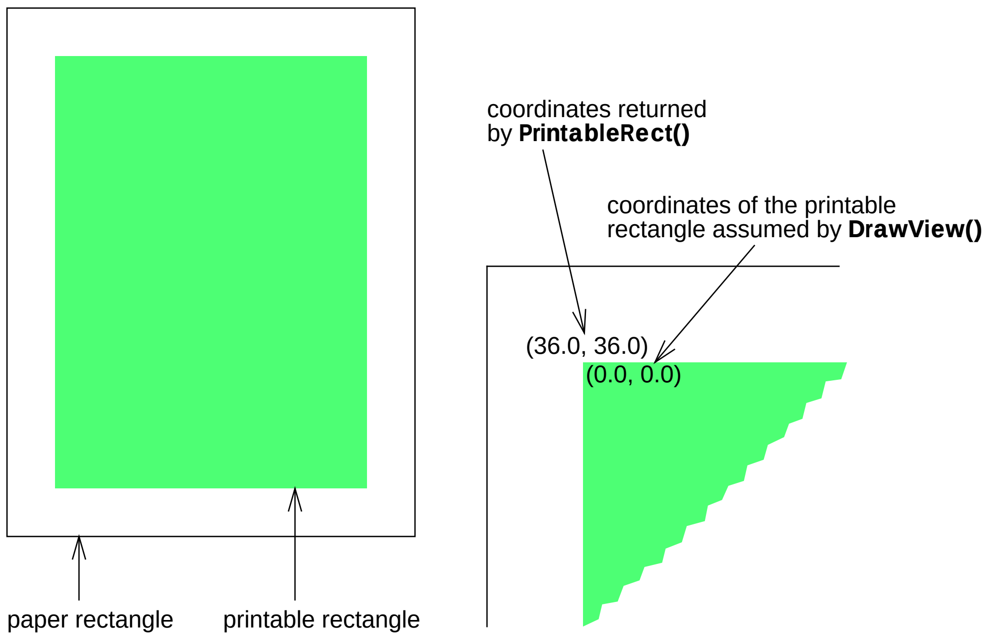
BRect **PrintableRect(**void**)**

These functions return rectangles that describe the size of a printed page

PaperRect() returns a rectangle that records the presumed size of the paper that the printer will use. It has 0.0 as its left and top coordinate values, and right and bottom coordinates that reflect the size of a sheet of paper. The size depends on choices made by the user when setting up the print job.

PrintableRect() returns a rectangle that encloses the portion of a page where printing can appear. It's stated in the same coordinate system as the rectangle returned by **PaperRect()**, but excludes the margins around the edge of the paper. When drawing on the printed page, the left top corner of this rectangle is taken to be the coordinate origin, (0.0, 0.0).

The diagram below illustrates the paper and printable rectangles, along with a closer view showing the coordinates of the left top corner of the printable rectangle as **PrintableRect()** would report them and as **DrawView()** would assume them, given a half-inch margin.


coordinates returned by **PrintableRect()**
coordinates of the printable rectangle assumed by **DrawView()**
(36.0, 36.0)
(0.0, 0.0)
paper rectangle
printable rectangle

See also: **DrawView()**

SetConfig(), Config()

void **SetConfig(**BMessage **configuration***)**

BMessage ***Config(**void**)**

These functions set and return the group of parameters that configure the Print Server for the current job. The parameters are recorded in a BMessage object that can be regarded as a black box; the entries in the message are interpreted by the Print Server and will be documented when the Server and the print driver API are documented.

Config() can be called to get the current configuration message, which can then be flattened and stored with the document. You can retrieve it later and pass it to **SetConfig()** to set initial configuration values the next time the document is printed, as illustrated in the "Overview".

See also: **InitJob()**

SpoolPage() *see* DrawView()

BRadioButton

Derived from: public BControl

Declared in: <interface/RadioButton.h>

Overview

A BRadioButton object draws a labeled, two-state button that's displayed in a group along with other similar buttons. The button itself is a round icon that has a filled center when the BRadioButton is turned on, and is empty when it's off. The label appears next to the icon.

Only one radio button in the group can be on at a time. When the user clicks a button to turn it on, the button that's currently on is turned off. The user can turn a button off only by turning another one on; one button in the group must be on at all times. The button that's on has a value of 1 (**B_CONTROL_ON**); the others have a value of 0 (**B_CONTROL_OFF**).

The BRadioButton class handles the interaction between radio buttons in the following way: A direct user action can only turn on a radio button, not turn it off. However, when the user turns a button on, the BRadioButton object turns off all sibling BRadioButtons—all BRadioButtons that have the same parent as the one that was turned on.

This means that a parent view should have no more than one group of radio buttons among its children. Each set of radio buttons should be assigned a separate parent—perhaps an empty BView that simply contains the radio buttons and does no drawing of its own.

Constructor

BRadioButton()

BRadioButton(BRect *frame*, const char **name*, const char **label*,
BMessage **message*,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = B_WILL_DRAW | B_NAVIGABLE**)**

Initializes the BRadioButton by passing all arguments to the BControl constructor without change. BControl initializes the radio button's *label* and assigns it a model *message* that identifies the action that should be taken when the radio button is turned on. When the user turns the button on, the BRadioButton posts a copy of the *message* so that it can be delivered to the target handler.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed without change from BControl to the BView constructor.

The BRadioButton draws at the bottom of its frame rectangle beginning at the left side. It ignores any extra space at the top or on the right. (However, the user can click anywhere within the *frame* rectangle to turn on the radio button). When the object is attached to a window, the height of the rectangle will be adjusted so that there is exactly the right amount of room to accommodate the label.

See also: the BControl and BView constructors, **AttachedToWindow()**

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of **AttachedToWindow()** to set the view and low colors of the BRadioButton to the match its parent's view color, and to resize the radio button vertically to fit the height of the label it displays. The height of the label depends on the BRadioButton's font (which the BControl class sets to Emily).

See also: **BControl::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the radio button—the circular icon—and its label. The center of the icon is filled when the BRadioButton's value is 1 (**B_CONTROL_ON**); it's left empty when the value is 0 (**B_CONTROL_OFF**).

See also: **BView::Draw()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Augments the inherited versions of **KeyDown()** to turn the radio button on and post a message to the target BHandler when *aChar* is **B_SPACE** or **B_ENTER**.

See also: **BView::KeyDown()**, **SetValue()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Responds to a mouse-down event in the radio button by tracking the cursor while the user holds the mouse button down. If the cursor is pointing to the radio button when the user releases the mouse button, this function turns the button on (and consequently turns all sibling BRadioButtons off), calls the BRadioButton's **Draw()** function, and posts a message that will be delivered to the target BHandler. Unlike a BCheckBox, a BRadioButton posts the message—it's "invoked"—only when it's turned on, not when it's turned off.

See also: **BControl::Invoke()**, **BControl::SetTarget()**, **SetValue()**

SetValue()

virtual void **SetValue(**long *value***)**

Augments the BControl version of **SetValue()** to turn all sibling BRadioButtons off (set their values to 0) when this BRadioButton is turned on (when the *value* passed is anything but 0).

See also: **BControl::SetValue()**

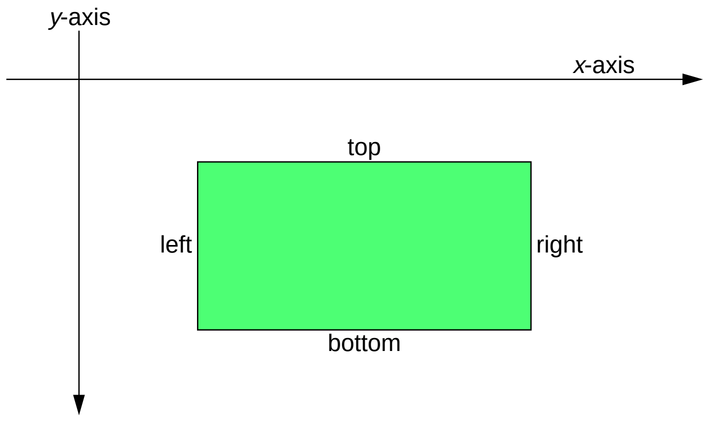
BRect

Derived from: none

Declared in: <interface/Rect.h>

Overview

A BRect object represents a *rectangle*, one with sides that parallel the *x* and *y* coordinate axes. The rectangle is defined by its left, top, right, and bottom coordinates, as illustrated below:



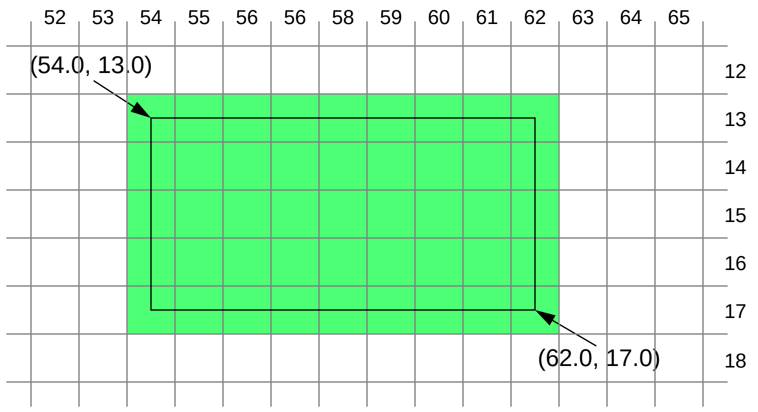
In a valid rectangle, the top *y* coordinate value is never greater than the bottom *y* coordinate, and the left *x* coordinate value is never greater than the right.

A BRect is the simplest, most basic way of specifying an area in a two-dimensional coordinate system. Windows, scroll bars, buttons, text fields, and the screen itself are all specified as rectangles. For more details on the definition of a rectangle, see "Coordinate Geometry" on page 16 in the chapter introduction.

When used to define the frame of a window or a view, or the bounds of a bitmap, the sides of the rectangle must line up on screen pixels. For this reason, the rectangle can't have any fractional coordinates. Coordinate units have a one-to-one correspondence with screen pixels.

Integral coordinates fall at the center of screen pixels, so frame rectangles cover a larger area than their coordinate values would indicate. Just as the number of elements in an array is one greater than the largest index, a frame rectangle covers one more column of pixels than its width and one more row than its height.

The figure below illustrates why this is the case. It shows a rectangle with a right side 8.0 units from its left (62.0–54.0) and a bottom 4.0 units below its top (17.0–13.0). Because the pixels that lie on all four sides of the rectangle are considered to be inside it, there's an extra pixel in each direction. When the rectangle is filled on-screen, it covers a 9-pixel-by-5-pixel area.



Because the BRect structure is a basic data type for graphic operations, it's constructed more simply than most other Interface Kit classes: All its data members are publicly accessible, it doesn't have virtual functions, it doesn't inherit from BObject or any other class, and it doesn't retain class information that it can reveal at run time. Within the Interface Kit, BRect objects are passed and returned by value.

Data Members

float **left**	The coordinate value of the rectangle's leftmost side (the smallest *x* coordinate in a valid rectangle).
float **top**	The coordinate value of the rectangle's top (the smallest *y* coordinate in a valid rectangle).
float **right**	The coordinate value of the rectangle's rightmost side (the largest *x* coordinate in a valid rectangle).
float **bottom**	The coordinate value of the rectangle's bottom (the largest *y* coordinate in a valid rectangle).

Constructor

BRect()

inline **BRect(**float *left*, float *top*, float *right*, float *bottom***)**
inline **BRect(**BPoint *leftTop*, BPoint *rightBottom***)**
inline **BRect(**const BRect& *rect***)**
inline **BRect(**void**)**

Initializes a BRect with its four coordinate values—*left*, *top*, *right*, and *bottom*. The four values can be directly stated,

```
BRect rect(11.0, 24.7, 301.5, 99.0);
```

or they can be taken from two points designating the rectangle's left top and right bottom corners,

```
BPoint leftTop(11.0, 24.7);
BPoint rightBottom(301.5, 99.0);
BRect rect(leftTop, rightBottom);
```

or they can be copied from another rectangle:

```
BRect anotherRect(11.0, 24.7, 301.5, 99.0);
BRect rect(anotherRect);
```

A rectangle that's not assigned any initial values,

```
BRect rect;
```

is constructed to be invalid (its top and left are greater than its right and bottom), until a specific assignment is made, typically with the **Set()** function:

```
rect.Set(77.0, 2.25, 510.8, 393.0);
```

See also: **Set()**

Member Functions

Contains()

bool **Contains(**BPoint *point***)** const
bool **Contains(**BRect *rect***)** const

Returns **TRUE** if *point*—or *rect*—lies inside the area the BRect defines, and **FALSE** if not. A rectangle contains a point even if the point coincides with one of the rectangle's corners or lies on one of its edges.

One rectangle contains another if their union is the same as the first rectangle and their intersection is the same as the second—that is, if the second rectangle lies entirely within the first. A rectangle is considered to be inside another rectangle even if they have one or more sides in common. Two identical rectangles contain each other.

See also: **Intersects()**, the **&** (intersection) and **|** (union) operators, **BPoint::ConstrainTo()**

Height() *see* Width()

InsetBy()

void **InsetBy(**float *horizontal*, float *vertical***)**
void **InsetBy(**BPoint *point***)**

Modifies the BRect by insetting its left and right sides by *horizontal* units and its top and bottom sides by *vertical* units. (If a *point* is passed, its *x* coordinate value substitutes for *horizontal* and its *y* coordinate value substitutes for *vertical*.)

For example, this code

```
BRect rect(10.0, 40.0, 100.0, 140.0);
rect.InsetBy(20.0, 30.0);
```

produces a rectangle identical to one that could be constructed as follows:

```
BRect rect(30.0, 70.0, 80.0, 110.0);
```

If *horizontal* or *vertical* is negative, the rectangle becomes larger in that dimension, rather than smaller.

See also: **OffsetBy()**

IntegerWidth(), IntegerHeight()

inline long **IntegerWidth(**void**)** const

inline long **IntegerHeight(**void**)** const

These functions return the width and height of the rectangle expressed as integers. Fractional widths and heights are rounded up to the next whole number.

See also: **Width()**

Intersects()

> bool **Intersects(**BRect *rect***)** const

Returns **TRUE** if the BRect has any area—even a corner or part of a side—in common with *rect*, and **FALSE** if it doesn't.

See also: the **&** (intersection) operator

IsValid()

> inline bool **IsValid(**void**)** const

Returns **TRUE** if the BRect's right side is greater than or equal to its left and its bottom is greater than or equal to its top, and **FALSE** otherwise. An invalid rectangle doesn't designate any area, not even a line or a point.

LeftBottom() *see* SetLeftBottom()

LeftTop() *see* SetLeftTop()

OffsetBy(), OffsetTo()

> void **OffsetBy(**float *horizontal*, float *vertical***)**
> void **OffsetBy(**BPoint *point***)**
>
> void **OffsetTo(**BPoint *point***)**
> void **OffsetTo(**float *x*, float *y***)**

These functions reposition the rectangle in its coordinate system, without altering its size or shape.

OffsetBy() adds *horizontal* to the left and right coordinate values of the rectangle and *vertical* to its top and bottom coordinates. (If a *point* is passed, *point.x* substitutes for *horizontal* and *point.y* for *vertical*.)

OffsetTo() moves the rectangle so that its left top corner is at *point*—or at (*x*, *y*). The coordinate values of all its sides are adjusted accordingly.

See also: **InsetBy()**

PrintToStream()

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

Prints the contents of the BRect object to the standard output stream (**stdout**) in the form:

```
"BRect(left, top, right, bottom)"
```

where *left*, *top*, *right*, and *bottom* stand for the current values of the BRect's data members.

RightBottom() *see* SetRightBottom()

RightTop() *see* SetRightTop()

Set()

inline void **Set(**float *left*, float *top*, float *right*, float *bottom***)**

Assigns the values *left*, *top*, *right*, and *bottom* to the BRect's corresponding data members. The following code

```
BRect rect;
rect.Set(0.0, 25.0, 50.0, 75.0);
```

is equivalent to:

```
BRect rect;
rect.left = 0.0;
rect.top = 25.0;
rect.right = 50.0;
rect.bottom = 75.0;
```

See also: the BRect constructor

SetLeftBottom(), LeftBottom()

void **SetLeftBottom(**const BPoint *point***)**

inline BPoint **LeftBottom(**void**)** const

These functions set and return the left bottom corner of the rectangle. **SetLeftBottom()** alters the BRect so that its left bottom corner is at *point*, and **LeftBottom()** returns its current left and bottom coordinates as a BPoint object.

See also: **SetLeftTop()**, **SetRightBottom()**, **SetRightTop()**

SetLeftTop(), LeftTop()

void **SetLeftTop(**const BPoint *point***)**

inline BPoint **LeftTop(**void**)** const

These functions set and return the left top corner of the rectangle. **SetLeftTop()** alters the BRect so that its left top corner is at *point*, and **LeftTop()** returns its current left and top coordinates as a BPoint object.

See also: **SetLeftBottom()**, **SetRightTop()**, **SetRightBottom()**

SetRightBottom(), RightBottom()

void **SetRightBottom(**const BPoint *point***)**

inline BPoint **RightBottom(**void**)** const

These functions set and return the right bottom corner of the rectangle. **SetRightBottom()** alters the BRect so that its right bottom corner is at *point*, and **RightBottom()** returns its current right and bottom coordinates as a BPoint object.

See also: **SetRightTop()**, **SetLeftBottom()**, **SetLeftTop()**

SetRightTop(), RightTop()

void **SetRightTop(**const BPoint *point***)**

inline BPoint **RightTop(**void**)** const

These functions set and return the right top corner of the rectangle. **SetRightTop()** alters the BRect so that its right top corner is at *point*, and **RightTop()** returns its current right and top coordinates as a BPoint object.

See also: **SetRightBottom()**, **SetLeftTop()**, **SetLeftBottom()**

Width(), Height()

inline float **Width(**void**)** const

inline float **Height(**void**)** const

These functions return the width of the rectangle (the difference between the coordinates of its left and right sides) and its height (the difference between its top and bottom coordinates). If either value is negative, the rectangle is invalid.

The width and height of a rectangle are not accurate guides to the number of pixels it covers on-screen. As illustrated in the “Overview” to this class, a rectangle without fractional coordinates covers an area that's one pixel broader than its coordinate width and one pixel taller than its coordinate height.

See also: **IntegerWidth()**

Operators

= (assignment)

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

Assigns the data members of one BRect object to another BRect:

```
BRect a(27.2, 36.8, 230.0, 359.1);
BRect b;
b = a;
```

Rectangle *b* is made identical to rectangle *a*.

== (equality)

bool **operator ==(**BRect**)** const

Compares the data members of two BRect objects and returns **TRUE** if each one exactly matches its counterpart in the other object, and **FALSE** if any of the members don't match. In the following example, the equality operator would return **FALSE**, since the two objects have different right boundaries:

```
BRect a(11.5, 22.5, 66.5, 88.5);
BRect b(11.5, 22.5, 46.5, 88.5);
if ( a == b )
     . . .
```

!= (inequality)

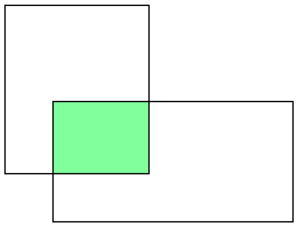
char **operator !=(**BRect**)** const

Compares two BRect objects and returns **TRUE** unless their data members match exactly (the two rectangles are identical), in which case it returns **FALSE**. This operator is the inverse of the **==** (equality) operator.

& (intersection)

BRect **operator &(**BRect**)** const

Returns the intersection of two rectangles—a rectangle enclosing the area they have in common. The shaded area below shows where the two outlined rectangles intersect.



The intersection is computed by taking the greatest left and top coordinate values of the two rectangles, and the smallest right and bottom values. In the following example,

```
BRect a(10.0, 40.0, 80.0, 100.0);
BRect b(35.0, 15.0, 95.0, 65.0);
BRect c = a & b;
```

rectangle *c* will be identical to one constructed as follows:

```
BRect c(35.0, 40.0, 80.0, 65.0);
```

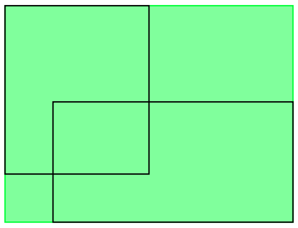
If the two rectangles don't actually intersect, the result will be invalid. You can test for this by calling the **Intersects()** function on the original rectangles, or by calling **IsValid()** on the result.

See also: **Intersects()**, **IsValid()**, the **|** (union) operator

| (union)

BRect **operator |(**BRect**)** const

Returns the union of two rectangles—the smallest rectangle that encloses them both. The shaded area below illustrates the union of the two outlined rectangles. Note that it includes areas not in either of them.



The union is computed by selecting the smallest left and top coordinate values from the two rectangles, and the greatest right and bottom coordinate values. In the following example,

```
BRect a(10.0, 40.0, 80.0, 100.0);
BRect b(35.0, 15.0, 95.0, 65.0);
BRect c = a | b;
```

rectangle *c* will be identical to one constructed as follows:

```
BRect c(10.0, 15.0, 95.0, 100.0);
```

Note that two rectangles will have a valid union even if they don't intersect.

See also: the **&** (intersection) operator

BRegion

Derived from:	public BObject
Declared in:	<interface/Region.h>

Overview

A BRegion object describes an arbitrary area within a two-dimensional coordinate system. The area can have irregular boundaries, contain holes, or be discontinuous. It’s convenient to think of a region as a set of locations or points, rather than as a closed shape like a rectangle or a polygon.

The points that a region includes can be described by a set of rectangles. Any point that lies within at least one of the rectangles belongs to the region. You can define a region incrementally by passing rectangles to functions like **Set()**, **Include()**, and **Exclude()**.

BView's **GetClippingRegion()** function modifies a BRegion object so that it represents the current clipping region of the view. A BView can pass **GetClippingRegion()** a pointer to an empty BRegion,

```
BRegion temp;
GetClippingRegion(&temp);
```

then call BRegion’s **Intersects()** and **Contains()** functions to test whether the potential drawing it might do falls within the region:

```
if ( temp.Intersects(someRect) )
    . . .
```

Constructor and Destructor

BRegion()

BRegion(const BRegion& *region***)**
BRegion(void**)**

Initializes the BRegion object to have the same area as another *region*—or, if no other region is specified, to an empty region.

The original BRegion object and the newly constructed one each have their own copies of the data describing the region. Altering or freeing one of the objects will not affect the other.

BRegion objects can be allocated on the stack and assigned to other objects:

```
BRegion regionOne(anotherRegion);
BRegion regionTwo = regionOne;
```

However, due to their size, it's more efficient to pass them by pointer rather than by value.

~BRegion

virtual **~BRegion(**void**)**

Frees any memory that was allocated to hold data describing the region.

Member Functions

Contains()

bool **Contains(**BPoint *point***)** const

Returns **TRUE** if *point* lies within the region, and **FALSE** if not.

Exclude()

void **Exclude(**BRect *rect***)**
void **Exclude(**const BRegion **region***)**

Modifies the region so that it excludes all points contained within *rect* or *region* that it might have included before.

See also: **Include()**, **IntersectWith()**

Frame()

BRect **Frame(**void**)** const

Returns the frame rectangle of the BRegion—the smallest rectangle that encloses all the points within the region.

If the region is empty, the rectangle returned won't be valid.

See also: **BRect::IsValid()**

Include()

void **Include(**BRect *rect***)**
void **Include(**const BRegion **region***)**

Modifies the region so that it includes all points contained within the *rect* or *region* passed as an argument.

See also: **Exclude()**

IntersectWith()

void **IntersectWith(**const BRegion **region***)**

Modifies the region so that it includes only those points that it has in common with another *region*.

See also: **Include()**

Intersects()

bool **Intersects(**BRect *rect***)** const

Returns **TRUE** if the BRegion has any area in common with *rect*, and **FALSE** if not.

MakeEmpty()

void **MakeEmpty(**void**)**

Empties the BRegion of all its points. It will no longer designate any area and its frame rectangle won't be valid.

See also: the BRegion constructor

OffsetBy()

void **OffsetBy(**long *horizontal*, long *vertical***)**

Offsets all points contained within the region by adding *horizontal* to each *x* coordinate value and *vertical* to each *y* coordinate value.

PrintToStream()

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

Prints the contents of the BRegion to the standard output stream (**stdout**) as an array of strings. Each string describes a rectangle in the form:

```
"BRect(left, top, right, bottom)"
```

where *left*, *top*, *right*, and *bottom* are the coordinate values that define the rectangle.

The first string in the array describes the BRegion's frame rectangle. Each subsequent string describes one portion of the area included in the BRegion.

See also: **BRect::PrintToStream()**, **Frame()**

Set()

void **Set(**BRect *rect***)**

Modifies the BRegion so that it describes an area identical to *rect*. A subsequent call to **Frame()** should return the same rectangle (unless some other change was made to the region in the interim).

See also: **Include()**, **Exclude()**

Operators

= (assignment)

BRegion& **operator =(**const BRegion&**)**

Assigns the region described by one BRegion object to another BRegion:

```
BRegion region = anotherRegion;
```

After the assignment, the two regions will be identical, but independent, copies of one another. Each object allocates its own memory to store the description of the region.

BScrollBar

Derived from:	public BView
Declared in:	<interface/ScrollBar.h>

Overview

A BScrollBar object displays a scroll bar that users can operate to scroll the contents of another view, a *target view.* Scroll bars usually come in pairs, one horizontal and one vertical, and are often grouped as siblings of the target view under a common parent. That way, when the parent is resized, the target and scroll bars can be automatically resized to match. (A companion class, BScrollView, defines just such a container view; a BScrollView object sets up the scroll bars for a target view and makes itself the parent of the target and the scroll bars.)

The Update Mechanism

BScrollBars are different from other views in one important respect: All their drawing and event handling is carried out within the Application Server, not in the application. A BScrollBar object doesn't receive **Draw()** or **MouseDown()** notifications; the Server intercepts updates and interface messages that would otherwise be reported to the BScrollBar and handles them itself. As the user moves the knob on a scroll bar or presses a scroll arrow, the Application Server continuously refreshes the scroll bar's image on-screen and informs the application with a steady stream of messages reporting value-changed events.

The window dispatches these messages by calling the BScrollBar's **ValueChanged()** function. Each function call notifies the BScrollBar of a change in its value and, consequently, of a need to scroll the target view.

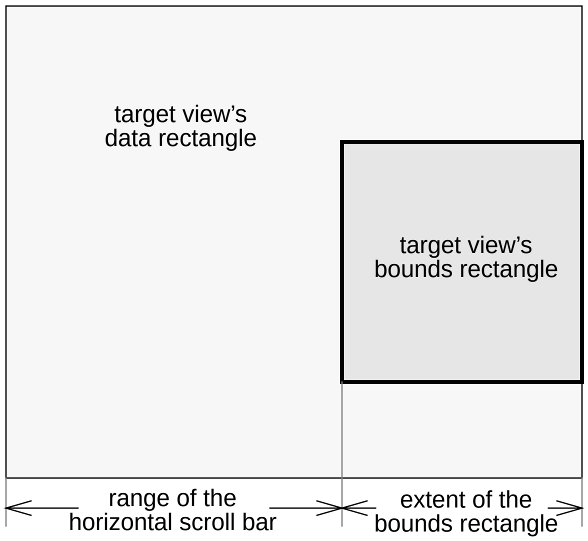
Confining the update mechanism for scroll bars to the Application Server limits the volume of communication between the application and Server and enhances the efficiency of scrolling. The application's messages to the Server can concentrate on updating the target view as its contents are being scrolled, rather than on updating the scroll bars themselves.

Value and Range

A scroll bar's value determines what the target view displays. The default assumption is that the left coordinate value of the target view's bounds rectangle should match the value of the horizontal scroll bar, and the top of the target view's bounds rectangle should match the value of the vertical scroll bar. When a BScrollBar is notified of a change of value (through its **ValueChanged()** function), it scrolls the target view to put the new value at the left or top of the bounds rectangle.

The value reported in a **ValueChanged()** notification depends on where the user moves the scroll bar's knob and on the range of values the scroll bar represents. The range is first set in the BScrollBar constructor and can be modified by the **SetRange()** function.

The range must be large enough to bring all the coordinate values where the target view can draw into its bounds rectangle. If everything the target view can draw is conceived as being enclosed in a "data rectangle," the range of a horizontal scroll bar must extend from a minimum that makes the left side of the target's bounds rectangle coincide with the left side of its data rectangle, to a maximum that puts the right side of the bounds rectangle at the right side of the data rectangle. This is illustrated in part below:



As this illustration helps demonstrate, the maximum value of a horizontal scroll bar can be no less than the right coordinate value of the data rectangle minus the width of the bounds rectangle. Similarly, for a vertical scroll bar, the maximum value can be no less than the bottom coordinate of the data rectangle minus the height of the bounds rectangle. The range of a scroll bar subtracts the dimensions of the target's bounds rectangle from its data rectangle. (The minimum values of horizontal and vertical scroll bars can be no greater than the left and top sides of the data rectangle.)

What the target view can draw may change from time to time as the user adds or deletes data. As this happens, the range of the scroll bar should be updated with the **SetRange()** function. The range may also need to be recalculated when the target view is resized.

Scroll Bar Options

Users have control over some aspects of how scroll bars look and behave. With the ScrollBar preferences application, they can choose:

- Whether the knob should be a fixed size, or whether it should grow and shrink to proportionally represent how much of a document (how much of the data rectangle) is visible within the target view. A proportional knob is the default.
- Whether double, bidirectional scroll arrows should appear on each end of the scroll bar, or whether each end should have only a single, unidirectional arrow. Double arrows are the default.
- Which of three patterns should appear on the knob.
- What the size of the knob should be—the minimum length of a proportional knob or the fixed length of a knob that's not proportional. The default length is 15 pixels.

When this class constructs a new BScrollBar, it conforms the object to the choices the user has made.

See also: **set_scroll_bar_info()**, **BView::ScrollBar()**, the BScrollView class

Hook Functions

ValueChanged()	Scrolls the target view when the BScrollBar is informed that its value has changed; can be implemented to alter the default interpretation of the scroll bar's value.

Constructor and Destructor

BScrollBar()

BScrollBar(BRect *frame*, const char **name*, BView **target*,
long *min*, long *max*, orientation *posture***)**

Initializes the BScrollBar and connects it to the *target* view that it will scroll. It will be a horizontal scroll bar if *posture* is **B_HORIZONTAL** and a vertical scroll bar if *posture* is **B_VERTICAL**.

The range of values that the scroll bar can represent at the outset is set by *min* and *max*. These values should be calculated from the boundaries of a rectangle that encloses the entire contents of the target view—everything that it can draw. If *min* and *max* are both 0, the scroll bar is disabled and the knob is not drawn.

The object's initial value is 0 < even if that falls outside the range set for the scroll bar >.

The other arguments, *frame* and *name*, are the same as for other BViews:

- The *frame* rectangle locates the scroll bar within its parent view. For consistency in the user interface, a horizontal scroll bar should be **B_H_SCROLL_BAR_HEIGHT** coordinate units high, and a vertical scroll bar should be **B_V_SCROLL_BAR_WIDTH** units wide.
- The BScrollBar's *name* identifies it and permits it to be located by the **FindView()** function. It can be **NULL**.

Unlike other BViews, the BScrollBar constructor doesn't set an automatic resizing mode. By default, scroll bars have the resizing behavior that befits their posture—horizontal scroll bars resize themselves horizontally (as if they had a resizing mode that combined **B_FOLLOW_LEFT_RIGHT** with **B_FOLLOW_BOTTOM**) and vertical scroll bars resize themselves vertically (as if their resizing mode combined **B_FOLLOW_TOP_BOTTOM** with **B_FOLLOW_RIGHT**).

~BScrollBar()

virtual **~BScrollBar(**void**)**

Disconnects the scroll bar from its target.

Member Functions

GetRange() *see* SetRange()

GetSteps() *see* SetSteps()

Orientation()

inline orientation **Orientation(**void**)** const

Returns **HORIZONTAL** if the object represents a horizontal scroll bar and **VERTICAL** if it represents a vertical scroll bar.

See also: the BScrollBar constructor

SetProportion(), Proportion()

void **SetProportion(**float *ratio***)**

float **Proportion(**void**)** const

These functions set and return a value between 0.0 and 1.0 that represents the proportion of the entire document that can be displayed within the target view—the ratio of the width (or height) of the target's bounds rectangle to the width (or height) of its data rectangle. This ratio determines the size of a proportional scroll knob relative to the whole scroll bar. It's not adjusted to take into account the minimum size of the knob.

The proportion should be reset as the size of the data rectangle changes (as data is entered and removed from the document) and when the target view is resized.

SetRange(), GetRange()

void **SetRange(**long *min*, long *max***)**

void **GetRange(**long **min*, long **max***)** const

These functions modify and return the range of the scroll bar. **SetRange()** sets the minimum and maximum values of the scroll bar to *min* and *max*. **GetRange()** places the current minimum and maximum in the variables that *min* and *max* refer to.

If the scroll bar's current value falls outside the new range, it will be reset to the closest value—either *min* or *max*—within range. **ValueChanged()** is called to inform the BScrollBar of the change whether or not it's attached to a window.

If the BScrollBar is attached to a window, any change in its range will be immediately reflected on-screen. The knob will move to the appropriate position to reflect the current value.

Setting both the minimum and maximum to 0 disables the scroll bar. It will be drawn without a knob.

See also: the BScrollBar constructor

SetSteps(), GetSteps()

void **SetSteps(**long *smallStep*, long *bigStep***)**

void **GetSteps(**long **smallStep*, long **bigStep***)** const

SetSteps() sets how much a single user action should change the value of the scroll bar—and therefore how far the target view should scroll. **GetSteps()** provides the current settings.

When the user presses one of the scroll arrows at either end of the scroll bar, its value changes by a *smallStep*. When the user clicks in the bar itself (other than on the knob), it changes by a *bigStep*. For an application that displays text, the small step of a vertical scroll bar should be large enough to bring another line of text into view.

The default small step is 1, which should be too small for most purposes; the default large step is 10, which is also probably too small.

< Currently, a BScrollBar's steps can be successfully set only after it's attached to a window. >

See also: **ValueChanged()**

SetTarget(), Target()

void **SetTarget(**BView **view***)**
void **SetTarget(**const char **name***)**

inline BView ***Target(**void**)** const

These functions set and return the target of the BScrollBar, the view that the scroll bar scrolls. **SetTarget()** sets the target to *view*, or to the BView identified by *name*. **Target()** returns the current target view. The target can also be set when the BScrollBar is constructed.

SetTarget() can be called either before or after the BScrollBar is attached to a window. If the target is set by *name*, the named view must eventually be found within the same window as the scroll bar. Typically, the target and its scroll bars are children of a container view that serves to bind them together as a unit.

See also: the BScrollBar constructor, **ValueChanged()**, **BView::ScrollBar()**

SetValue(), Value()

void **SetValue(**long *value***)**

long **Value(**void**)** const

These functions modify and return the value of the scroll bar. The value is usually set as the result of user actions; **SetValue()** provides a way to do it programmatically. **Value()** returns the current value, whether set by **SetValue()** or by the user.

SetValue() assigns a new *value* to the scroll bar and calls the **ValueChanged()** hook function, whether or not the new value is really a change from the old. If the *value* passed lies outside the range of the scroll bar, the BScrollBar is reset to the closest value within range—that is, to either the minimum or the maximum value previously specified.

If the scroll bar is attached to a window, changing its value updates its on-screen display. The call to **ValueChanged()** enables the object to scroll the target view so that it too is updated to conform to the new value.

The initial value of a scroll bar is 0.

See also: **ValueChanged()**, **SetRange()**

Target() *see* **SetTarget()**

Value() *see* **SetValue()**

ValueChanged()

virtual void **ValueChanged(**long *newValue***)**

Responds to a notification that the value of the scroll bar has changed to *newValue*. For a horizontal scroll bar, this function interprets *newValue* as the coordinate value that should be at the left side of the target view's bounds rectangle. For a vertical scroll bar, it interprets *newValue* as the coordinate value that should be at the top of the rectangle. It calls **ScrollTo()** to scroll the target view's contents accordingly.

ValueChanged() does nothing if a target BView hasn't been set—or if the target has been set by name, but the name doesn't correspond to an actual BView within the scroll bar's window.

Derived classes can override this function to interpret *newValue* differently, or to do something in addition to scrolling the target view.

ValueChanged() is called as the result both of value-changed messages received from the Application Server and of **SetValue()** and **SetRange()** function calls within the application.

See also: **SetTarget()**

BScrollView

Derived from:	public BView
Declared in:	<interface/ScrollView.h>

Overview

A BScrollView object is a container for another view, a *target view*, typically a view that can be scrolled. The BScrollView creates and positions the scroll bars the target view needs and makes itself the parent of the scroll bars and the target view. It's a convenient way to set up scroll bars for another view.

If requested, the BScrollView draws a one-pixel wide black border around its children. Otherwise, it does no drawing and simply contains the family of views it set up.

The **ScrollBar()** function provides access to the scroll bars the BScrollView creates, so you can set their ranges and values as needed.

Constructor and Destructor

BScrollView()

BScrollView(const char **name*, BView **target*,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = 0,
bool *horizontal* = FALSE,
bool *vertical* = FALSE,
bool *bordered* = TRUE**)**

Initializes the BScrollView. It will have a frame rectangle large enough to contain the *target* view and any scroll bars that are requested. If *horizontal* is **TRUE**, there will be a horizontal scroll bar. If *vertical* is **TRUE**, there will be a vertical scroll bar. Scroll bars are not provided unless you ask for them.

If *bordered* is **TRUE**, as it is by default, the frame rectangle will also be large enough to draw a narrow black border around the target view and scroll bars. A BScrollView can be used without scroll bars to simply contain and border the target view.

The BScrollView adapts its frame rectangle from the frame rectangle of the target view. It positions itself so that its left and top sides are exactly where the left and top sides of the target view originally were. It then adds the target view as its child along with any requested scroll bars. In the process, it modifies the target view's frame rectangle (but not its bounds rectangle) so that it will fit within its new parent.

If the resize mode of the target view is **B_FOLLOW_ALL_SIDES**, it and the scroll bars will be automatically resized to fill the container view whenever the container view is resized.

The scroll bars created by the BScrollView have an initial range extending from a minimum of 0 to a maximum of 1000. You'll generally need to ask for the scroll bars (using the **ScrollBar()** function) and set their ranges to more appropriate values.

The *name*, *resizeMode*, and *flags* arguments are identical to those declared in the BView class and are passed unchanged to the BView constructor.

See also: the BView constructor

~BScrollView()

virtual **~BScrollView(**void**)**

Does nothing.

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Resizes scroll bars belonging to BScrollViews that occupy the right bottom corner of a document window (**B_DOCUMENT_WINDOW**) so that room is left for the resize knob. This function assumes that vertical scroll bars are **B_V_SCOLL_BAR_WIDTH** units wide and horizontal scroll bars are **B_H_SCROLL_BAR_HEIGHT** units high. It doesn't check to make sure the window is actually resizable.

See also: **BView::AttachedToWindow()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws a one-pixel wide black border around the target view and scroll views, provided the *bordered* flag wasn't set to **FALSE** in the BScrollView constructor.

See also: the BScrollView constructor, **BView::Draw()**

IsBordered() ***see*** **SetBordered()**

ScrollBar()

BScrollBar ***ScrollBar(**orientation *posture***)** const

Returns the horizontal scroll bar if *posture* is **B_HORIZONTAL** and the vertical scroll bar if *posture* is **B_VERTICAL**. If the BScrollView doesn't contain a scroll bar with the requested orientation, this function returns **NULL**.

See also: the BScrollBar class

SetBordered(), IsBordered()

virtual void **SetBordered(**bool *bordered***)**

inline bool **IsBordered(**void**)** const

SetBordered() determines whether a narrow black border will be drawn around the edge of the view. Calling this function is equivalent to passing a *bordered* flag to the BScrollView constructor. **Bordered()** returns the current flag.

See also: the BScrollView constructor

BSeparatorItem

Derived from: public BMenuItem

Declared in: <interface/MenuItem.h>

Overview

A BSeparatorItem is a menu item that serves only to separate the items that precede it in the menu list from the items that follow it. It's drawn as a horizontal line across the menu from the left border to the right. Although it has an indexed position in the menu list just like other items, it doesn't have a label, can't be selected, posts no messages, and is permanently disabled.

Since the separator is drawn horizontally, it's assumed that items in the menu are arranged in a column, as they are by default. It's inappropriate to use a separator in a menu bar or another menu where the items are arranged in a row.

A separator can be added to a BMenu by constructing an object of this class and calling BMenu's **AddItem()** function. As a shorthand, you can simply call BMenu's **AddSeparatorItem()** function, which constructs the object for you and adds it to the list.

A BSeparatorItem that's returned to you (by BMenu's **ItemAt()** function, for example) will always respond **NULL** to **Message()**, **Command()**, and **Submenu()** queries and **FALSE** to **IsEnabled()**.

See also: **BMenu::AddSeparatorItem()**

Constructor and Destructor

BSeparatorItem()

BSeparatorItem(void**)**

Initializes the BSeparatorItem and disables it.

~BSeparatorItem()

virtual **~BSeparatorItem(**void**)**

Does nothing.

Member Functions

Draw()

protected:
    virtual void **Draw(**void**)**

Draws the item as a horizontal line across the width of the menu.

GetContentSize()

protected:
    virtual void **GetContentSize(**float **width*, float **height***)**

Provides a minimal size for the item so that it won't constrain the size of the menu.

SetEnabled()

    virtual void **SetEnabled(**bool *flag***)**

Does nothing. A BSeparatorItem is disabled when it's constructed and must stay that way.

BStringView

Derived from:	public BView
Declared in:	<interface/StringView.h>

Overview

A BStringView object draws a static character string. The user can't select the string or edit it; a BStringView doesn't respond to user actions. An instance of this class can be used to draw a label or other text that simply delivers a message of some kind to the user. Use a BTextView object for selectable and editable text.

You can also draw strings by calling BView's **DrawString()** function. However, assigning a string to a BStringView object locates it in the view hierarchy. The string will be updated automatically, just like other views. And, by setting the resizing mode of the object, you can make sure that it will be positioned properly when the window or the view it's in (the parent of the BStringView) is resized.

Constructor and Destructor

BStringView()

BStringView(BRect *frame*, const char **name*, const char **text*,
ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
ulong *flags* = B_WILL_DRAW**)**

Initializes the BStringView by assigning it a *text* string, the **B_OP_OVER** drawing mode, and the Erich bitmap font. These last two values are cached and communicated to the Application Server when the BStringView is attached to a window.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class. They're passed unchanged to the BView constructor.

The *frame* rectangle needs to be large enough to display the entire string in the current font. The string is drawn at the bottom of the frame rectangle and, by default, is aligned to the left side. A different horizontal alignment can be set by calling **SetAlignment()**.

See also: **SetAlignment()**

~BStringView()

virtual **~BStringView(**void**)**

Frees the text string.

Member Functions

Alignment() *see* SetAlignment()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the string along the bottom of the BStringView's frame rectangle in the current high color.

See also: **BView::Draw()**

SetAlignment(), Alignment()

void **SetAlignment(**alignment *flag***)**

inline alignment **Alignment(**void**)** const

These functions align the string within the BStringView's frame rectangle and return the current alignment. The alignment *flag* can be:

B_ALIGN_LEFT	The string is aligned at the left side of the frame rectangle.
B_ALIGN_RIGHT	The string is aligned at the right side of the frame rectangle.
B_ALIGN_CENTER	The string is aligned so that the center of the string falls midway between the left and right sides of the frame rectangle.

The default is **B_ALIGN_LEFT**.

SetText(), Text()

void **SetText(**const char **string***)**

inline const char ***Text(**void**)** const

These functions set and return the text string that the BStringView draws. **SetText()** frees the previous string and copies *string* to replace it. **Text()** returns the null-terminated string.

BTextControl

Derived from:	public BControl
Declared in:	<interface/TextControl.h>

Overview

A BTextControl object displays a labeled text field that behaves like other control devices. When the user takes certain key actions after modifying the text in the field, it posts a message to a designated target.

There are two parts to the view: A static label on the left, which the user cannot modify, and an editable field on the right, which behaves just like a one-line BTextView. In fact, the BTextControl installs a BTextView object as its child to handle editing chores within this part of the view. It's this child view that responds to events for the BTextControl rather than the control object itself.

The child BTextView must become the focus view for the window before the user can enter or edit text in the field. If the user modifies the contents of the field and then causes the child to cease being the focus view, the BTextControl posts a copy of its model message to its target, just like any other BControl object when it's invoked. The message notifies the target that the user has finished making changes to the text. (It doesn't matter what causes the change in focus—a click in another text field, for example, or a **B_TAB** character that navigates to another view.)

The message is also posted when the user types a **B_ENTER** character, though this doesn't change the focus view. It selects all the text in the field.

You can also arrange for another message—a "modification message"—to be posted when the user makes the first change to the text after the child BTextView has become the focus view (or after **B_ENTER** caused all the text to be selected).

Because the label is drawn by the BTextControl itself and the editable text is drawn by its child BTextView, you can assign different properties (color or font, for example) to each string. The BTextControl has only one child, so **ChildAt()** returns it when passed an index of 0.

Constructor and Destructor

BTextControl()

> **BTextControl(**BRect *frame*, const char **name*,
> const char **label*, const char **text*,
> BMessage **message*,
> ulong *resizingMode* = B_FOLLOW_LEFT | B_FOLLOW_TOP,
> ulong *flags* = B_WILL_DRAW | B_NAVIGABLE**)**

Initializes the BTextControl by assigning it a *label* and some *text*, both of which can be **NULL**. If the *label* is **NULL**, the text can fill the bounds rectangle. Otherwise, half the view is assigned to the label and half to the text, though the exact proportion can be changed by the **SetDivider()** function. The label always is on the left and the text always on the right. By default, both label and text are aligned at the left margins of their respective sections; call **SetAlignment()** to alter the alignment.

The *message* parameter is the same as the one declared for the BControl constructor. It establishes a model for the messages the BTextControl will send when it's invoked. It can be **NULL**. See **SetMessage()**, **SetTarget()**, and **Invoke()** in the BControl class for more information.

The *frame*, *name*, *resizingMode*, and *flags* arguments are the same as those declared for the BView class and are passed up the inheritance hierarchy to the BView constructor without change.

See also: **SetDivider()**, **SetAlignment()**, **BControl::SetMessage()**, **BControl::SetTarget()**, **BControl::Invoke()**

~BTextControl()

> virtual **~BTextControl(**void**)**

Frees memory allocated by the BTextControl and its BTextView child.

Member Functions

AttachedToWindow()

> virtual void **AttachedToWindow(**void**)**

Augments the BControl version of **AttachedToWindow()** to make the background color of the BTextControl the same as the background color of its parent and to set up its child BTextView.

See also: **BView::AttachedToWindow()**, **BControl::AttachedToWindow()**

Divider() ***see*** **SetDivider()**

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the label. (The BTextControl defers to its child BTextView to draw the editable text string.)

See also: **BView::Draw()**

GetAlignment() ***see*** **SetAlignment()**

Label() ***see*** **SetLabel()**

MakeFocus()

virtual void **MakeFocus(**bool *flag* = TRUE**)**

Passes the **MakeFocus()** instruction on to the child BTextView. If the *flag* is **TRUE**, this function selects all the text in the child BTextView, which becomes the new focus view for the window. If the *flag* is **FALSE**, the child will no longer be the focus view. If the text has changed when the child ceases to be the focus view, the BTextControl is considered to have been invoked; a copy of its model message is posted so that it will be delivered to the target handler.

Note that the BTextControl itself never becomes the focus view, so will return **FALSE** to all **IsFocus()** queries.

See also: **BView::MakeFocus()**

ModificationMessage() ***see*** **SetModificationMessage()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Does nothing. The child BTextView handles the job of responding to the user.

See also: **BTextView::MouseDown()**

SetAlignment(), GetAlignment()

virtual void **SetAlignment(**alignment *forLabel*, alignment *forText***)**

void **GetAlignment(**alignment **forLabel*, alignment **forText***)** const

These functions set and report the alignment of the label and the text within their respective portions of the view. Three settings are possible:

B_ALIGN_LEFT	The label or text is aligned at the left boundary of its part of the view rectangle.
B_ALIGN_RIGHT	The label or text is aligned at the right boundary of its part of the view rectangle.
B_ALIGN_CENTER	The label or text is centered within its part of the view rectangle.

The default alignment is **B_ALIGN_LEFT** for both label and text.

See also: **SetDivider()**

SetDivider(), Divider()

virtual void **SetDivider(**float *xCoordinate***)**

float **Divider(**void**)** const

These functions set and return the *x* coordinate value that marks the division between the label portion of the view rectangle on the left and the text portion on the right. It's stated in the coordinate system of the BTextControl.

See also: the BTextControl constructor

SetEnabled()

virtual void **SetEnabled(**bool *enabled***)**

Disables the BTextControl if the *enabled* flag is **FALSE**, and reenables it if *enabled* is **TRUE**. BTextControls are enabled by default.

This function augments the BControl version of **SetEnabled()**. When the control is disabled, it makes the text unselectable (and therefore uneditable) and draws it in a way that displays its disabled state. When the control is re-enabled, it makes the text editable (and therefore selectable) and draws it as normal text.

See also: **BControl::SetEnabled()**

SetLabel(), Label()

virtual void **SetLabel(**const char **text***)**

const char ***Label(**void**)** const

These functions set and return the label displayed by the BTextControl. The label is first set by the constructor.

SetModificationMessage(), ModificationMessage()

virtual void **SetModificationMessage(**BMessage **message***)**

BMessage ***ModificationMessage(**void**)** const

These functions set and return the message that the BTextControl posts when the user begins to enter or edit text.

SetModificationMessage() assigns *message* to the BTextControl, freeing the message previously assigned, if any. The message becomes the responsibility of the BTextControl object and will be freed only when it's replaced by another message or the BTextControl is freed; you shouldn't free it yourself. Passing a **NULL** pointer to this function deletes the current modification message without replacing it.

The assigned BMessage becomes the model for the message that the BTextControl posts when the user first modifies the text after the child BTextView has become the focus view (or after the user pressed the Enter key). The message is sent only for the first character the user types, pastes, or deletes. Subsequent changes don't invoke the message, until after the user presses the Enter key to select all the text or after the child BTextView loses focus view status and regains it again.

Before posting the message, the BTextControl adds two data entries to the copy:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the user modified the text, as measured in microseconds since the machines was last booted.
"source"	**B_OBJECT_TYPE**	A pointer to the BTextControl object.

These names should not be used for any data that you place in the model *message*.

ModificationMessage() returns the model message.

SetText(), Text()

> virtual void **SetText(**const char **text***)**
>
> const char ***Text(**void**)** const

These functions set and return the text displayed by the BTextControl—or rather by its child BTextView. The text is first set by the constructor.

BTextView

Derived from:	public BView
Declared in:	<interface/TextView.h>

Overview

The BTextView class defines a view that displays text on-screen and supports a standard user interface for entering, selecting, and editing text from the keyboard and mouse. It also supports the principal editing commands—Cut, Copy, Paste, Delete, and Select All.

BTextView objects are suitable for displaying small amounts of text in the user interface and for creating textual data in ASCII format. Full-scale text editors and word processors will need to define their own objects to handle richer data formats.

A BTextView displays all its text in a single font, the font that it inherits as a BView graphics parameter. Multiple fonts are not supported. Paragraph properties—such as alignment and tab widths—are similarly uniform for all text displayed within the view.

Resizing

A BTextView can be made to resize itself to exactly fit the text that the user enters. This is sometimes appropriate for small one-line text fields. See the **MakeResizable()** function.

Shortcuts and Menu Items

When a BTextView is the focus view for its window, it responds to these standard keyboard shortcuts for cutting, copying, and pasting text:

- Command-*x* to cut text and copy it to the clipboard,
- Command-*c* to copy text without cutting it, and
- Command-*v* to paste text taken from the clipboard.

These shortcuts work even in the absence of Cut, Copy, and Paste menu items; they're implemented by the BWindow for any view that might be the focus view. All the focus view has to do is cooperate, as a BTextView does, by handling the messages the shortcuts generate.

The only trick is to set up menu items that are compatible with the shortcuts. Follow these guidelines if you put a menu with editing commands in a window that has a BTextView:

- Create Cut, Copy, and Paste menu items and assign them the Command-*x*, Command-*c*, and Command-*v* shortcuts.
- Assign the items model **B_CUT**, **B_COPY** and **B_PASTE** messages. These messages don't need to contain any information (other than a **what** data member initialized to the proper constant).
- Target the messages to the BWindow's focus view (or directly to the BTextView). No changes to the BTextView are necessary. When it gets these messages, the BTextView calls its **Cut()**, **Copy()**, and **Paste()** functions.

You can also set up menu items that trigger calls to other BTextView editing and layout functions. Simply create menu items like Select All or Align at Left that are targeted to the focus view of the window where the BTextView is located, or to the BTextView itself. The model messages assigned to these items can be structured with whatever command constants and data entries you wish; the BTextView class imposes no constraints.

Then, in a class derived from BTextView, implement a **MessageReceived()** function that responds to messages posted from the menu items by calling BTextView functions like **SelectAll()** and **SetAlignment()**. For example:

```
void myText::MessageReceived(BMessage *message)
{
    switch ( message->what ) {
    case SELECT_ALL:
        SelectAll();
        break;
    case ALIGN_AT_LEFT:
        SetAlignment(B_ALIGN_LEFT);
        break;
    case ALIGN_AT_RIGHT:
        SetAlignment(B_ALIGN_RIGHT);
        break;
    . . .
    default:
        BTextView::MessageReceived(message);
        break;
    }
}
```

The **MessageReceived()** function you implement should be sure to call BTextView's version of the function, which already handles **B_CUT**, **B_COPY**, and **B_PASTE** messages.

Newlines and Carriage Returns

A BTextView object treats newline characters ('\n', 0x0a) and carriage return characters ('\r', 0x0d) alike. It converts received return characters into newlines and stores them only as newlines. By default, none of keys on the BeBox is mapped to a carriage return. The **B_ENTER** character is a newline.

Hook Functions

AcceptsChar()	Can be implemented to preview the characters the user types and either accept or reject them before they're added to the display.
BreaksAtChar()	Breaks word selection on spaces, tabs, and other invisible characters, permitting all adjacent visible characters to be selected when the user double-clicks a word. This function can be augmented to break word selection on other characters in addition to the invisible ones.

Constructor and Destructor

BTextView()

BTextView(BRect *frame*, const char **name*, BRect *textRect*,
ulong *resizingMode*, ulong *flags***)**

Initializes the BTextView to the *frame* rectangle, stated in its eventual parent's coordinate system, assigns it an identifying *name*, sets its resizing behavior to *resizingMode* and its drawing behavior with *flags*. These four arguments—*frame*, *name*, *resizingMode*, and *flags*—are identical to those declared for the BView class and are passed unchanged to the BView constructor.

The text rectangle, *textRect*, is stated in the BTextView's coordinate system. It determines where text in placed within the view's bounds rectangle:

- The first line of text is placed at the top of the text rectangle. As additional lines of text are entered into the view, the text grows downward and may actually extend beyond the bottom of the rectangle.
- The left and right sides of the text rectangle determine where lines of text are placed within the view. Lines can be aligned to either side of the rectangle, or they can be centered between the two sides. See the **SetAlignment()** function.
- When lines wrap on word boundaries, the width of the text rectangle determines the maximum length of a line; each line of text can be as long as the rectangle is wide.

When word wrapping isn't turned on, lines can extend beyond the boundaries of the text rectangle. See the **SetWordWrap()** function.

The bottom of the text rectangle is ignored; it doesn't limit the amount of text the view can contain. The text can be limited by the number of characters, but not by the number of lines.

The constructor establishes the following default properties for a new BTextView:

- The text is left-aligned.
- The tab width is 44.0 coordinate units.
- Automatic indenting and word wrapping are turned off.
- The text is selectable and editable.
- All characters the user may type are acceptable.

A BTextView isn't fully initialized until it's assigned to a window and it receives an **AttachedToWindow()** notification.

See also: **AttachedToWindow()**, the BView constructor

~BTextView()

virtual **~BTextView(**void**)**

Frees the memory the BTextView allocated to hold the text and to store information about it.

Member Functions

AcceptsChar()

virtual bool **AcceptsChar(**ulong *aChar***)** const

Implemented by derived classes to return **TRUE** if *aChar* designates a character that the BTextView can add to its text, and **FALSE** if not. By returning **FALSE**, this function prevents the character from being displayed or retained by the object.

AcceptsChar() is called for every character the user types (including those, like **B_BACKSPACE** and **B_RIGHT_ARROW**, that are used for editing the text). The default version of this function always returns **TRUE**, but it can be overridden in a derived class to restrict the text the user can enter. For example, a BTextView might reject uppercase letters, or permit only numbers, or allow only those characters that are valid in a pathname.

Sometimes, a character will be meaningful and trigger a response of some kind, even though it can't be displayed. For example, a **B_TAB** (0x09) might be rejected as a character to display, and instead shift the selection to another text field. Similarly, a BTextView that has room to display only a single line of text might return **FALSE** for the newline character (**B_ENTER**, 0x0a), yet take the occasion to simulate a click on a button.

When rejecting a character outright (not using it to take some other action), an application has an obligation to explain to the user why the character is unacceptable, perhaps by displaying an alert panel or dialog box.

As an alternative to implementing an **AcceptsChar()** function, you can simply inform the BTextView at the outset that certain characters should not be allowed. Call **DisallowChar()** when setting up the BTextView to tell it which characters won't be acceptable.

See also: **KeyDown()**, **DisallowChar()**

Alignment() *see* SetAlignment()

AllowChar() *see* DisallowChar()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Completes the initialization of the BTextView object after it becomes attached to a window. This function sets up the object so that it can correctly format text and display it. It makes sure that all properties that were previously set—for example, word wrapping, tab width, and alignment—are correctly reflected in the display on-screen. In addition, it calls **SetFontName()** and **SetFontSize()** to set the font to the 9.0-point Erich bitmap font (no rotation, 90° shear).

Because the BTextView uses pulses to animate (or "blink") the caret, the vertical line that marks the current insertion point, it enables pulsing in the window and fixes the pulse rate at 2 per second (once every 500,000 microseconds).

This function is called for you when the BTextView becomes part a window's view hierarchy; you shouldn't call it yourself, though you can override it to set a different default font and do other graphics initialization. For more information on when it's called, see the BView class.

An **AttachedToWindow()** function that's implemented by a derived class should begin by incorporating the BTextView version:

```
void MyText::AttachedToWindow()
{
    BTextView::AttachedToWindow()
    . . .
}
```

If it doesn't, the BTextView won't be able to properly display the text.

See also: **BView::AttachedToWindow()**, **SetFontName()**

BreaksAtChar()

virtual bool **BreaksAtChar(**ulong *aChar***)** const

Implemented by derived classes to return **TRUE** if the *aChar* character can break word selection, and **FALSE** if it cannot. The BTextView class calls this function when the user selects a word by double-clicking it. A return of **TRUE** means that the character breaks the selection—it cannot be selected as part of the word. A return of **FALSE** means that the character will be included in the selected word.

By default, **BreaksAtChar()** returns **TRUE** if the character is a **B_SPACE** (0x20), a **B_TAB** (0x09), a newline (**B_ENTER**, 0x0a), or some other character with an ASCII value less than that of a space, and **FALSE** otherwise.

It can be reimplemented to add hyphens to the list of characters that break word selection, as follows:

```
bool MyTextView::BreaksAtChar(ulong someChar)
{
    if ( someChar == '-' )
        return TRUE;
    return BTextView::BreaksAtChar(someChar);
}
```

See also: **Text()**

CharAt() *see* Text()

Copy()

virtual void **Copy(**BClipboard **clipboard***)**

Copies the current selection to the clipboard. The *clipboard* argument is identical to the global **be_clipboard** object.

See also: **Paste()**, **Cut()**

CountLines() *see* GoToLine()

CurrentLine() *see* GoToLine()

Cut()

virtual void **Cut(**BClipboard **clipboard***)**

Copies the current selection to the clipboard, deletes it from the BTextView's text, and removes it from the display. The *clipboard* argument is identical to the global **be_clipboard** object.

See also: **Paste()**, **Copy()**

Delete()

void **Delete(**void**)**

Deletes the current selection from the BTextView's text and removes it from the display, without copying it to the clipboard.

See also: **Cut()**

DisallowChar(), AllowChar()

void **DisallowChar(**ulong *aChar***)**

void **AllowChar(**ulong *aChar***)**

These functions inform the BTextView whether the user should be allowed to enter *aChar* into the text. By default, all characters are allowed. Call **DisallowChar()** for each character you want to prevent the BTextView from accepting, preferably when first setting up the object.

AllowChar() reverses the effect of **DisallowChar()**.

Alternatively, and for more control over the context in which characters are accepted or rejected, you can implement an **AcceptsChar()** function for the BTextView. **AcceptsChar()** is called for each key-down event that's reported to the object.

See also: **AcceptsChar()**

DoesAutoindent() *see* SetAutoindent()

DoesWordWrap() *see* SetWordWrap()

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the text on-screen. The Interface Kit calls this function for you whenever the text display needs to be updated—for example, whenever the user edits the text, enters new characters, or scrolls the contents of the BTextView.

See also: **BView::Draw()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Overrides the BView version of this function to reset the ranges of the BTextView's scroll bars and to update the sizes of their proportional knobs whenever the size of the BTextView changes.

See also: **BView::FrameResized()**

GetSelection()

void **GetSelection(**long **start*, long **finish***)**

Provides the current selection by writing the offset before the first selected character into the variable referred to by *start* and the offset after the last selected character into the variable referred to by *finish*. If no characters are selected, both offsets will record the position of the current insertion point.

The offsets designate positions between characters. The position at the beginning of the text is offset 0, the position between the first and second characters is offset 1, and so on. If the 175th through the 202nd characters were selected, the *start* offset would be 174 and the *finish* offset would be 202.

If the text isn't selectable, both offsets will be 0.

See also: **Select()**

GetText() *see* Text()

GoToLine(), CountLines(), CurrentLine()

void **GoToLine(**long *index***)**

long **CurrentLine(**void**)** const

inline long **CountLines(**void**)** const

GoToLine() moves the insertion point to the beginning of the line at *index*. The first line has an index of 0, the second line an index of 1, and so on. If the *index* is out-of-range, the insertion point is moved to the beginning of the line with the nearest in-range index—that is, to either the first or the last line.

CurrentLine() returns the index of the line where the first character of the selection—or the character following the insertion point—is currently located.

CountLines() returns how many lines of text the BTextView currently contains.

Like other functions that change the selection, **GoToLine()** doesn't automatically scroll the display to make the new selection visible. Call **ScrollToSelection()** to be sure that the user can see the start of the selection.

See also: **ScrollToSelection()**

Highlight()

void **Highlight(**long *start*, long *finish***)**

Highlights the characters from *start* through *finish*, where *start* and *finish* are the same sort of offsets into the text array as are passed to **Select()**.

Highlight() is the function that the BTextView calls to highlight the current selection. You don't need to call it yourself for this purpose. It's in the public API just in case you may need to highlight a range of text in some other circumstance.

See also: **Select()**

IndexAtPoint()

long **IndexAtPoint(**BPoint *point***)** const
long **IndexAtPoint(**float *x*, float *y***)** const

Returns the index of the character displayed closest to *point*—or (*x*, *y*)—in the BTextView's coordinate system. The first character in the text array is at index 0.

If the point falls after the last line of text, the return value is the index of the last character in the last line. If the point falls before the first line of text, or if the BTextView doesn't contain any text, the return value is 0.

See also: **Text()**

Insert()

void **Insert(**const char **text*, long *length***)**
void **Insert(**const char **text***)**

Inserts *length* characters of *text*—or if a *length* isn't specified, all the characters of the *text* string up to the null character that terminates it—at the beginning of the current selection. The current selection is not deleted and the insertion is not selected.

See also: **SetText()**

IsEditable() *see* MakeEditable()

IsSelectable() *see* MakeSelectable()

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Enters text at the current selection in response to the user's typing. This function is called from the window's message loop for every report of a key-down event—once for every character the user types. However, it does nothing unless the BTextView is the focus view and the text it contains is editable.

If *aChar* is one of the arrow keys (**B_UP_ARROW**, **B_LEFT_ARROW**, **B_DOWN_ARROW**, or **B_RIGHT_ARROW**), **KeyDown()** moves the insertion point in the appropriate direction. If *aChar* is the **B_BACKSPACE** character, it deletes the current selection (or one character at the current insertion point). Otherwise, it checks whether the character was registered as unacceptable (by **DisallowChar()**) and it calls the **AcceptsChar()** hook function to give the application a chance to reject the character or handle it in some other way. If the character isn't disallowed and **AcceptsChar()** returns **TRUE**, it's entered into the text and displayed.

See also: **BView::KeyDown()**, **AcceptsChar()**, **DisallowChar()**

LineHeight()

inline float **LineHeight(**void**)** const

Returns the height of a single line of text, as measured from the baseline of one line of single-spaced text to the baseline of the line above or below it.

The height is stated in coordinate units and depends on the current font. It's the sum of how far characters can ascend above and descend below the baseline, plus the amount of leading that separates lines.

See also: **BView::GetFontInfo()**

LineWidth()

float **LineWidth(**long *index* = 0**)** const

Returns the width of the line at *index*—or, if no index is given, the width of the first line. The value returned is the sum of the widths (in coordinate units) of all the characters in the line, from the first through the last, including tabs and spaces.

Line indices begin at 0.

If the *index* passed is out-of-range, it's reinterpreted to be the nearest in-range index—that is, as the index to the first or the last line.

MakeEditable(), IsEditable()

void **MakeEditable(**bool *flag* = TRUE**)**

bool **IsEditable(**void**)** const

The first of these functions sets whether the user can edit the text displayed by the BTextView; the second returns whether or not the text is currently editable. Text is editable by default.

To edit text, the user must be able to select it. Therefore, when **MakeEditable()** is called with an argument of **TRUE** (or with no argument), it makes the text both editable and selectable. Similarly, when **IsEditable()** returns **TRUE**, the text is selectable as well as editable; **IsSelectable()** will also return **TRUE**.

A value of **FALSE** means that the text can't be edited, but implies nothing about whether or not it can be selected.

See also: **MakeSelectable()**

MakeFocus()

virtual void **MakeFocus(**bool *flag* = TRUE**)**

Overrides the BView version of **MakeFocus()** to highlight the current selection when the BTextView becomes the focus view (when *flag* is **TRUE**) and to unhighlight it when the BTextView no longer is the focus view (when *flag* is **FALSE**). However, the current selection is highlighted only if the BTextView's window is the current active window.

This function is called for you whenever the user's actions make the BTextView become the focus view, or force it to give up that status.

See also: **BView::MakeFocus()**, **MouseDown()**

MakeResizable()

void **MakeResizable(**BView **containerView***)**

Makes the BTextView's frame rectangle and text rectangle automatically grow and shrink to exactly enclose all the characters entered by the user. The *containerView* is a view that should be resized with the BTextView; typically it's a view that draws a border around the text (like a BScrollView object) and is the parent of the BTextView. This function won't work without a container view.

MakeResizable() is an alternative to the automatic resizing behavior provided in the BView class. It triggers resizing on the user's entry of text, not on a change in the parent view's size. The two schemes are incompatible; the BTextView and the container view should not automatically resize themselves when their parents are resized.

< This function currently requires the text to be either left aligned or center aligned; it doesn't work for text that's right aligned. >

See also: **SetAlignment()**

MakeSelectable(), IsSelectable()

void **MakeSelectable(**bool *flag* = TRUE**)**

bool **IsSelectable(**void**)** const

The first of these functions sets whether it's possible for the user to select text displayed by the BTextView; the second returns whether or not the text is currently selectable. Text is selectable by default.

When text is selectable but not editable, the user can select one or more characters to copy to the clipboard, but can't position the insertion point (an empty selection), enter characters from the keyboard, or paste new text into the view.

Since the user must be able to select text to edit it, calling **MakeSelectable()** with an argument of **FALSE** causes the text to become uneditable as well as unselectable. Similarly, if **IsSelectable()** returns **FALSE**, the user can neither select nor edit the text; **IsEditable()** will also return **FALSE**.

A value of **TRUE** means that the text is selectable, but says nothing about whether or not it's also editable.

See also: **MakeEditable()**

MessageReceived()

virtual void **MessageReceived(**BMessage **message***)**

Overrides the BHandler version of **MessageReceived()** to handle four messages.

If this function gets a **B_SIMPLE_DATA** message, it looks for a data named “text” registered as **B_ASCII_TYPE**. Failing that, it looks for a single character named “char” registered as **B_LONG_TYPE**. If successful, it assumes that the message was dragged and dropped on the view. It changes the current selection to the point of drop and inserts the text or character at that point.

This function handles **B_CUT**, **B_COPY**, and **B_PASTE** messages by calling the **Cut()**, **Copy()**, and **Paste()** virtual functions. For the BTextView to get these messages, Cut, Copy, and Paste menu items should be:

- Assigned model messages with **B_CUT**, **B_COPY**, and **B_PASTE** as their **what** data members, and
- Targeted to the BTextView, or to the current focus view in the window that displays the BTextView.

The BTextView, through this function, takes care of the rest.

To inherit this functionality, **MessageReceived()** functions implemented by derived classes should be sure to call the BTextView version.

See also: **BMenuItem::SetMessage()**, **BMenuItem::SetTarget()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Selects text and positions the insertion point in response to the user’s mouse actions. If the BTextView isn’t already the focus view for its window, this function calls **MakeFocus()** to make it the focus view.

MouseDown() is called for each mouse-down event that occurs inside the BTextView’s frame rectangle.

See also: **BView::MouseDown()**, **BView::MakeFocus()**

MouseMoved()

virtual void **MouseMoved(**BPoint *point*, ulong *transit*, BMessage **message***)**

Responds to messages reporting mouse-moved events by changing the cursor to the standard I-beam image for editing text whenever the cursor enters the view and by resetting it to the standard hand image when the cursor exits the view.

The cursor is changed to an I-beam only for text that is selectable, and only if the BTextView is the current focus view in the active window.

See also: **BView::MouseMoved()**

Paste()

virtual void **Paste(**BClipboard **clipboard***)**

Takes textual data from the clipboard and pastes it into the text. The new text replaces the current selection, or is placed at the site of the current insertion point.

The *clipboard* argument is identical to the global **be_clipboard** object.

See also: **Cut()**, **Copy()**

Pulse()

virtual void **Pulse(**void**)**

Turns the caret marking the current insertion point on and off when the BTextView is the focus view in the active window. **Pulse()** is called by the system at regular intervals.

This function is first declared in the BView class.

See also: **BView::Pulse()**

ScrollToSelection()

void **ScrollToSelection(**void**)**

Scrolls the text so that the beginning of the current selection is within the visible region of the view, provided that the BTextView is equipped with a scroll bar that permits scrolling in the required direction (horizontal or vertical).

See also: **BView::ScrollBy()**

Select()

void **Select(**long *start*, long *finish***)**

Selects the characters from *start* up to *finish*, where *start* and *finish* are offsets into the BTextView's text. The offsets designate positions between characters. For example,

```
Select(0, 2);
```

selects the first two characters of text,

```
Select(17, 18);
```

selects the eighteenth character, and

```
Select(0, TextLength());
```

selects the entire text just as the **SelectAll()** function does. If *start* and *finish* are the same, the selection will be empty (an insertion point).

Normally, the selection is changed by the user. This function provides a way to change it programmatically.

If the BTextView is the current focus view in the active window, **Select()** highlights the new selection (or displays a blinking caret at the insertion point). However, it doesn't automatically scroll the contents of the BTextView to make the new selection visible. Call **ScrollToSelection()** to be sure that the user can see the start of the selection.

See also: **Text()**, **GetSelection()**, **ScrollToSelection()**, **GoToLine()**, **MouseDown()**

SelectAll()

void **SelectAll(**void**)**

Selects the entire text of the BTextView, and highlights it if the BTextView is the current focus view in the active window.

See also: **Select()**

SetAlignment(), Alignment()

void **SetAlignment(**alignment *where***)**

alignment **Alignment(**void**)** const

These functions set the way text is aligned within the text rectangle and return the current alignment. Three settings are possible:

B_ALIGN_LEFT	Each line is aligned at the left boundary of the text rectangle.
B_ALIGN_RIGHT	Each line is aligned at the right boundary of the text rectangle.
B_ALIGN_CENTER	Each line is centered between the left and right boundaries of the text rectangle.

The default is **B_ALIGN_LEFT**.

SetAutoindent(), DoesAutoindent()

void **SetAutoindent(**bool *flag***)**

bool **DoesAutoindent(**void**)** const

These functions set and return whether a new line of text is automatically indented the same as the preceding line. When set to **TRUE** and the user types Return at the end of a line that begins with tabs or spaces, the new line will automatically indent past those tabs and spaces to the position of the first visible character.

The default value is **FALSE**.

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

These functions override their BView counterparts to recalculate the layout of the text when the font changes, and to prevent the text displayed by a BTextView object from being rotated.

Font rotation is disabled; the BTextView version of **SetFontRotation()** does nothing. The other three functions invoke their BView counterparts to change the font, then make sure the entire text is recalculated and rewrapped for the new font. However, the text display is not updated.

SetFontName() and **SetFontSize()** are called by **AttachedToWindow()** to set the BTextView's default font to 9.0-point Erich.

See also: **BView::SetFontName()**

SetMaxChars()

void **SetMaxChars(**long *max***)**

Sets the maximum number of characters that the BTextView can accept. The default is the maximum number of characters that can be designated by a **long** integer, a number sufficiently large to accommodate all uses of a BTextView. Use this function only if you need to restrict the number of characters that the user can enter in a text field.

SetSymbolSet()

virtual void **SetSymbolSet(**const char **name***)**

Overrides its BView counterpart to recalculate the text layout when the symbol set changes.

See also: **BView::SetSymbolSet()**

SetTabWidth(), TabWidth()

void **SetTabWidth(**float *width***)**

float **TabWidth(**void**)** const

These functions set the distance between tab stops to *width* coordinate units and return the current tab width. Tabs cannot be removed nor can they be individually set; all tabs have a uniform width. The default tab width is 44.0 coordinate units.

SetText()

void **SetText(**const char **text*, long *length***)**
void **SetText(**const char **text***)**

Removes any text currently in the BTextView and copies *length* characters of *text* to replace it—or all the characters in the *text* string, up to the null character, if a *length* isn't specified. If *text* is **NULL** or *length* is 0, this function empties the BTextView. Otherwise, it copies the required number of *text* characters passed to it.

This function is typically used to set the text initially displayed in the view. If the BTextView is attached to a window, it's updated to show its new contents.

See also: **Text()**, **TextLength()**

SetTextRect(), TextRect()

void **SetTextRect(**BRect *rect***)**

inline BRect **TextRect(**void**)** const

SetTextRect() makes *rect* the BTextView's text rectangle—the rectangle that locates where text is placed within the view. This replaces the text rectangle originally set in the BTextView constructor. The layout of the text is recalculated to fit the new rectangle, and the text is redisplayed.

TextRect() returns the current text rectangle.

See also: the BTextView constructor

SetWordWrap(), DoesWordWrap()

void **SetWordWrap(**bool *flag***)**

bool **DoesWordWrap(**void**)** const

These functions set and return whether the BTextView wraps lines on word boundaries, dropping entire words that don't fit at the end of a line to the next line. Words break on tabs, spaces, and other invisible characters; all adjacent visible characters wrap together.

By default, word wrapping is turned off (**DoesWordWrap()** returns **FALSE**). Lines break only on a newline character (where the user types return).

See also: **SetTextRect()**

TabWidth() *see* SetTabWidth()

Text(), GetText(), CharAt()

const char ***Text(**void**)**

const char ***GetText(**char **buffer*, long *index*, long *length***)** const

char **CharAt(**long *index***)** const

These functions reveal the text contained in the BTextView.

Text() returns a pointer to the text, which may be a pointer to an empty string if the BTextView is empty. The returned pointer can be used to read the text, but not to alter it (use **SetText()**, **Insert()**, **Delete()**, and other BTextView functions to do that).

GetText() copies up to *length* characters of the text into *buffer*, beginning with the character at *index*, and adds a null terminator ('\0'). The first character in the BTextView is at index 0, the second at index 1, and so on. Fewer than *length* characters are copied if there aren't that many between *index* and the end of the text. The results won't be reliable if the *index* is out-of-range.

CharAt() returns the specific character located at *index*.

The pointer that **Text()** returns is to the BTextView's internal representation of the text. When it returns, the text string is guaranteed to be null-terminated and without gaps. However, the BTextView may have had to manipulate the text to get it in that condition. Therefore, there may be a performance price to pay if **Text()** is called frequently. If you're going to copy the text, it's more efficient to have **GetText()** do it for you. If you're going to index into the text, it may be more efficient to call **CharAt()**.

The pointer that **Text()** returns may no longer be valid after the user or the program next changes the text. Even if valid, the string may no longer be null-terminated and gaps may appear.

See also: **TextLength()**

TextLength()

long **TextLength(**void**)** const

Returns the number of characters the BTextView currently contains—the number of characters that **Text()** returns (not counting the null terminator).

See also: **Text()**, **SetMaxChars()**

TextRect() *see* SetTextRect()

WindowActivated()

virtual void **WindowActivated(**bool *flag***)**

Highlights the current selection when the BTextView's window becomes the active window (when *flag* is **TRUE**)—provided that the BTextView is the current focus view—and removes the highlighting when the window ceases to be the active window (when *flag* is **FALSE**).

If the current selection is empty (if it's an insertion point), it's highlighted by turning the caret on and off (blinking it).

The Interface Kit calls this function for you whenever the BTextView's window becomes the active window or it loses that status.

See also: **BView::WindowActivated()**, **MakeFocus()**

BView

Derived from:	public BHandler
Declared in:	<interface/View.h>

Overview

BView objects are the agents for drawing and message handling within windows. Each object sets up and takes responsibility for a particular *view*, a rectangular area that's associated with at most one window at a time. The object draws within the view rectangle and responds to reports of events elicited by the images drawn.

Classes derived from BView implement the actual functions that draw and handle messages; BView merely provides the framework. For example, a BTextView object draws and edits text in response to the user's activity on the keyboard and mouse. A BButton draws the image of a button on-screen and responds when the button is clicked. BTextView and BButton inherit from the BView class—as do most classes in the Interface Kit.

The following Kit classes derive, directly or indirectly, from BView:

BControl	BButton	BMenu
BScrollBar	BPictureButton	BMenuBar
BScrollView	BRadioButton	BMenuField
BBox	BCheckBox	BPopUpMenu
BStringView	BColorControl	BListView
BTextView	BTextControl	

Serious applications will need to define their own classes derived from BView.

Views and Windows

For a BView to do its work, you must attach it to a window. The views in a window are arranged in a hierarchy—there can be views within views—with those that are most directly responsible for drawing and message handling located at the terminal branches of the hierarchy and those that contain and organize other views situated closer to its trunk and root. A BView begins life unattached. You can add it to a hierarchy by calling the **AddChild()** function of the BWindow, or of another BView.

Within the hierarchy, a BView object plays two roles:

- It's a BHandler for messages delivered to the window thread. BViews implement the functions that respond to the most common system messages—including those that report keyboard and mouse events. They can also be targeted to handle application-defined messages that affect what they view displays.

- It's an agent for drawing. Adding a BView to a window gives it an independent graphics environment. A BView draws on the initiative of the BWindow and the Application Server, whenever they determine that the appearance of any part of the view rectangle needs to be "updated." It also draws on its own initiative in response to events.

The relationship of BViews to BWindows and the framework for drawing and responding to the user were discussed in the introduction to this chapter. The concepts and terminology presented there are assumed in this class description. See especially "BView Objects" on page 11, "The View Hierarchy" on page 13, "Drawing" beginning on page 18, and "Responding to the User" beginning on page 41.

BViews can also be called upon to create bitmap images. See the BBitmap class for details.

User Interface

Since they provide the content that's displayed within windows, BViews carry most of the burden of implementing an application's user interface. Often this is simply a matter of how a BView implements a hook function—how **Draw()** presents the view or how **MouseDown()** handles a double-click. User-interface guidelines should be followed, but the BView is essentially on its own. However, in some cases the Interface Kit provides a mechanism that derived classes can participate in, if they coordinate with Kit-defined code. Two such mechanisms are described below—keyboard navigation and the drag-and-drop delivery of messages.

Keyboard Navigation

Keyboard navigation is a mechanism for allowing users to manipulate views—especially buttons, check boxes, and other control devices—from the keyboard. It gives users the ability to:

- Move the focus of keyboard actions from view to view within a window by pressing the Tab key, and

- Operate the view that's currently in focus by pressing the space bar and Enter key (to invoke it) or the arrow keys (to move around inside it).

The first ability—navigation between views—is implemented by the Interface Kit. The second—navigation within a view—is up to individual applications, as are most view-specific aspects of the user interface. The only trick, and it's not a difficult one, is to make the two kinds of navigation work together.

To have the BView class you implement participate in the navigation mechanism, you need to coordinate four pieces of code:

- Include **B_NAVIGABLE** in the BView's flag mask whenever it's possible for the user to navigate to it (when it can become the focus view). This flag should be removed from the mask when the view is disabled, and included again when it's re-enabled. The mask is first set on construction and can be altered with the **SetFlags()** function.
- Make sure the BView's **Draw()** function provides some sort of visual indication of whether the view is the current focus for keyboard actions. Guidelines are forthcoming on what the indication should be. Currently, Be-defined views underline text (for example, a button label) when the view is in focus, and avoid drawing the underline when it's not. **Draw()** can call **IsFocus()** to test the BView's current status.
- Override the **MakeFocus()** hook function to have it change the way the view is displayed when it becomes the focus view and when it loses that status. It's perhaps simplest just to have **MakeFocus()** call **Draw()**.
- Override **KeyDown()** to handle the keystrokes that are used to operate the view (for view-internal navigation). Always incorporate the inherited version so that it can take care of navigation between views.

Several Kit classes that derive from BView implement these functions. For example, BControl has a simple **KeyDown()** function and a **MakeFocus()** function that calls **Draw()**. If you base your class on BControl, you won't have to implement **MakeFocus()** and may find that its **KeyDown()** is adequate for your needs.

Drag and Drop

The BView class supports a drag-and-drop user interface. The user can transfer a parcel of information from one place to another by dragging an image from a source view and dropping it on a destination view—perhaps a view in a different window or even a different application.

A source BView initiates dragging by calling **DragMessage()** from within its **MouseDown()** function. The BView bundles all information relevant to the dragging session into a BMessage object and passes it to **DragMessage()**. It also passes an image to represent the data package on-screen.

The Application Server then takes charge of the BMessage object and animates the image as the user drags it on-screen. As the image moves across the screen, the views it passes over are informed with **MouseMoved()** function calls. These notifications give views a chance to show the user whether or not they're willing to accept the message being dragged. When the user releases the mouse button, dropping the dragged message, the message is delivered to the BWindow and targeted to the destination BView.

Aside from creating a BMessage object and passing it to **DragMessage()**, or implementing **MouseMoved()** and **MessageReceived()** functions to handle any messages that come its way, there's nothing an application needs to do to support a drag-and-drop user interface. The bulk of the work is done by the Application Server and Interface Kit.

Locking the Window

If a BView is attached to a window, any operation that affects the view might also affect the window and the BView's shadow counterpart in the Application Server. For this reason, any code that calls a BView function should first lock the window—so that one thread can't modify essential data structures while another thread is using them. A window can be locked by only one thread at a time.

By default, before they do anything else, almost all BView functions check to be sure the caller has the window locked. If the window isn't properly locked, they print warning messages and fail.

This check should help you develop an application that correctly regulates access to windows and views. However, it adds a certain amount of time to each function call. Once your application has been debugged and is ready to ship, you can turn the check off by calling BWindow's **SetDiscipline()** function and passing it an argument of **FALSE**. The discipline flag is separately set for each window.

BView functions can require the window to be locked only if the view has a window to lock; the requirement can't be enforced if the BView isn't attached to a window. However, as discussed under "Views and the Server" on page 31 of the introduction to this chapter, many BView functions, including all those that depend on graphics parameters, don't work at all unless the view is attached—in which case the window must be locked.

Whenever the system calls a BView function to notify it of something—whenever it calls **WindowActivated()**, **Draw()**, **MessageReceived()** or another hook function—it first locks the window thread. The application doesn't have to explicitly lock the window when responding to an update, an interface message, or some other notification. The window is already locked.

Derived Classes

When it comes time for a BView to draw, its **Draw()** virtual function is called automatically. When it needs to respond to an event, a virtual function named after the kind of event is called—**MouseMoved()**, **KeyDown()**, and so on. Classes derived from BView implement these hook functions to do the particular kind of drawing and message handling characteristic of the derived class.

- Some classes derived from BView implement control devices—buttons, dials, selection lists, check boxes, and so on—that translate user actions on the keyboard and mouse into more explicit instructions for the application. In the Interface Kit, BMenu, BListView, BButton, BCheckBox, and BRadioButton are examples of control devices.

- Other BViews visually organize the display—for example, a view that draws a border around and arranges other views, or one that splits a window into two or more resizable panels. The BBox, BScrollBar, and BScrollView classes fall into this category.
- Some BViews implement highly organized displays the user can manipulate, such as a game board or a scientific simulation.
- Perhaps the most important BViews are those that permit the user to create, organize, and edit data. These views display the current selection and are the focus of most user actions. They carry out the main work of an application. BTextView is the only Interface Kit example of such a view.

Almost all the BView classes defined in the Interface Kit fall into the first two of these groups. Control devices and organizational views can serve a variety of different kinds of applications, and therefore can be implemented in a kit that's common to all applications

However, the BViews that will be central to most applications fall into the last two groups. Of particular importance are the BViews that manage editable data. Unfortunately, these are not views that can be easily implemented in a common kit. Just as most applications devise their own data formats, most applications will need to define their own data-handling views.

Nevertheless, the BView class structures and simplifies the task of developing application-specific objects that draw in windows and interact with the user. It takes care of the lower-level details and manages the view's relationship to the window and other views in the hierarchy. You should make yourself familiar with this class before implementing you own application-specific BViews.

Hook Functions

AllAttached()	Can be implemented to finish initializing the BView after it's attached to a window, where the initialization depends on a descendent view's **AttachedToWindow()** function having been called.
AllDetached()	Can be implemented to prepare the BView for being detached from a window, where the preparations depend on a descendent view's **DetachedFromWindow()** function having been called.
AttachedToWindow()	Can be implemented to finish initializing the BView after it becomes part of a window's view hierarchy.

DetachedFromWindow()	Can be implemented to prepare the BView for its impending removal from a window's view hierarchy.
Draw()	Can be implemented to draw the view.
FrameMoved()	Can be implemented to respond to a message notifying the BView that it has moved in its parent's coordinate system.
FrameResized()	Can be implemented to respond to a message informing the BView that its frame rectangle has been resized.
KeyDown()	Can be implemented to respond to a message reporting a key-down event.
MakeFocus()	Makes the BView the focus view, or causes it to give up being the focus view; can be augmented to take any action the change in status may require.
MouseDown()	Can be implemented to respond to a message reporting a mouse-down event.
MouseMoved()	Can be implemented to respond to a notification that the cursor has entered the view's visible region, moved within the visible region, or exited from the view.
Pulse()	Can be implemented to do something at regular intervals. This function is called repeatedly when no other messages are pending.
WindowActivated()	Can be implemented to respond to a notification that the BView's window has become the active window, or has lost that status.

Constructor and Destructor

BView()

BView(BRect *frame*, const char **name*, ulong *resizingMode*, ulong *flags***)**

Sets up a view with the *frame* rectangle, which is specified in the coordinate system of its eventual parent, and assigns the BView an identifying *name*, which can be **NULL**.

When it's created, a BView doesn't belong to a window and has no parent. It's assigned a parent by having another BView adopt it with the **AddChild()** function. If the other view is in a window, the BView becomes part of that window's view hierarchy. A BView can be made a child of the window's top view by calling BWindow's version of the **AddChild()** function.

When the BView gains a parent, the values in *frame* are interpreted in the parent's coordinate system. The sides of the view must be aligned on screen pixels. Therefore, the *frame* rectangle should not contain coordinates with fractional values. Fractional coordinates will be rounded to the nearest whole number.

The *resizingMode* mask determines the behavior of the view when its parent is resized. It should combine one constant for horizontal resizing,

B_FOLLOW_LEFT
B_FOLLOW_RIGHT
B_FOLLOW_LEFT_RIGHT
B_FOLLOW_H_CENTER

with one for vertical resizing:

B_FOLLOW_TOP
B_FOLLOW_BOTTOM
B_FOLLOW_TOP_BOTTOM
B_FOLLOW_V_CENTER

For example, if **B_FOLLOW_LEFT** is chosen, the margin between the left side of the view and left side of its parent will remain constant—the view's left side will "follow" the parent's left side. Similarly, if **B_FOLLOW_RIGHT** is chosen, the view's right side will follow the parent's right side. If **B_FOLLOW_H_CENTER** is chosen, the horizontal center of the view will maintain a constant distance from the horizontal center of the parent.

If the constants name opposite sides of the view rectangle—left and right, or top and bottom—the view will necessarily be resized in that dimension when the parent is.

If a side is not mentioned, the distance between that side of the view and the corresponding side of the parent is free to fluctuate. This may mean that the view will move within its parent's coordinate system when the parent is resized. **B_FOLLOW_RIGHT** plus **B_FOLLOW_BOTTOM**, for example, would keep a view from being resized, but the view will move to follow the right bottom corner of its parent whenever the parent is resized. **B_FOLLOW_LEFT** plus **B_FOLLOW_TOP** prevents a view from being resized *and* from being moved.

In addition to the constants listed above, there are two other possibilities:

B_FOLLOW_ALL_SIDES
B_FOLLOW_NONE

B_FOLLOW_ALL_SIDES is a shorthand for **B_FOLLOW_LEFT_RIGHT** and **B_FOLLOW_TOP_BOTTOM**. It means that the view will be resized in tandem with its parent, both horizontally and vertically.

B_FOLLOW_NONE keeps the view at its absolute position on-screen; the parent view is resized around it. (Nevertheless, because the parent is resized, the view may wind up being moved in its parent's coordinate system.)

Typically, a parent view is resized because the user resizes the window it's in. When the window is resized, the top view is too. Depending on how the *resizingMode* flag is set for the top view's children and for the descendants of its children, automatic resizing can cascade down the view hierarchy. A view can also be resized programmatically by the **ResizeTo()** and **ResizeBy()** functions.

The resizing mode can be changed after construction with the **SetResizingMode()** function.

The *flags* mask determines what kinds of notifications the BView will receive. It can be any combination of these four constants:

B_WILL_DRAW	Indicates that the BView does some drawing of its own and therefore can't be ignored when the window is updated. If this flag isn't set, the BView won't receive update notifications—it won't be erased to its background color and its **Draw()** function won't be called.
B_PULSE_NEEDED	Indicates that the BView should receive **Pulse()** notifications.
B_FRAME_EVENTS	Indicates that the BView should receive **FrameResized()** and **FrameMoved()** notifications when its frame rectangle changes—typically as a result of the automatic resizing behavior described above. **FrameResized()** is called when the dimensions of the view change; **FrameMoved()** is called when the position of its left top corner in its parent's coordinate system changes.
B_FULL_UPDATE_ON_RESIZE	Indicates that the entire view should be updated when it's resized. If this flag isn't set, only the portions that resizing adds to the view will be included in the clipping region.
B_NAVIGABLE	Indicates that the BView can become the focus view for keyboard actions. This flag makes it possible for the user to navigate to the view and put it in focus by pressing the Tab key. See "Keyboard Navigation" above.

If none of these constants apply, *flags* can be **NULL**. The flags can be reset after construction with the **SetFlags()** function.

See also: **SetResizingMode()**, **SetFlags()**, **BHandler::SetName()**

~BView()

virtual **~BView(**void**)**

Removes the BView from the view hierarchy and ensures that each of its descendants is also removed and destroyed.

Member Functions

AddChild()

virtual void **AddChild(**BView **aView***)**

Makes *aView* a child of the BView, provided that *aView* doesn't already have a parent. If the BView is attached to a window, *aView* and all its descendants become attached to the same window. Each of them is notified of this change through **AttachedToWindow()** and **AllAttached()** function calls.

AddChild() fails if *aView* already belongs to a view hierarchy. A view can live with only one parent at a time.

When a BView object becomes attached to a BWindow, two other connections are automatically established for it:

- The view is added to the BWindow's flat list of BHandler objects, making it an eligible target for messages received by the BWindow.
- The BView's parent view becomes its next handler. Messages that the BView doesn't recognize will be passed to its parent.

See also: **BWindow::AddChild()**, **AttachedToWindow()**, **BLooper::AddHandler()**, **BHandler::SetNextHandler()**, **RemoveChild()**

AddLine() *see* BeginLineArray()

AllAttached() *see* AttachedToWindow()

AllDetached() *see* DetachedFromWindow()

AttachedToWindow(), AllAttached()

virtual void **AttachedToWindow(**void**)**

virtual void **AllAttached(**void**)**

Implemented by derived classes to complete the initialization of the BView when it's assigned to a window. A BView is assigned to a window when it, or one of its ancestors in the view hierarchy, becomes a child of a view already attached to a window.

AttachedToWindow() is called immediately after the BView is formally made a part of the window's view hierarchy and after it has become known to the Application Server and its graphics parameters are set. The **Window()** function can identify which BWindow the BView belongs to.

All of the BView's children, if it has any, also become attached to the window and receive their own **AttachedToWindow()** notifications. Parents receive the notification before their children, but only after all views have become attached to the window and recognized as part of the window's view hierarchy. This function can therefore depend on all ancestor and descendent views being in place.

For example, **AttachedToWindow()** can be implemented to set a view's background color to the same color as its parent, something that can't be done before the view belongs to a window and knows who its parent is.

```
void MyView::AttachedToWindow()
{
    if ( Parent() )
        SetViewColor(Parent()->ViewColor());
    inherited::AttachedToWindow();
}
```

The **AllAttached()** notification follows on the heels of **AttachedToWindow()**, but works its way up the view hierarchy rather than down. When **AllAttached()** is called for a BView, all its descendants have received both **AttachedToWindow()** and **AllAttached()** notifications. Therefore, parent views can depend on any calculations that their children make in either function. For example, a parent can resize itself to fit the size of its children, where their sizes depend on calculations done in **AttachedToWindow()**.

The default (BView) version of both these functions are empty.

See also: **AddChild()**, **Window()**

BeginLineArray(), AddLine(), EndLineArray()

void **BeginLineArray(**long *count***)**

void **AddLine(**BPoint *start*, BPoint *end*, rgb_color *color***)**

void **EndLineArray(**void**)**

These functions provide a more efficient way of drawing a large number of lines than repeated calls to **StrokeLine()**. **BeginLineArray()** signals the beginning of a series of up to *count* **AddLine()** calls; **EndLineArray()** signals the end of the series. Each **AddLine()** call defines a line from the *start* point to the *end* point, associates it with a particular *color*, and adds it to the array. The lines can each be a different color; they don't have to be contiguous. When **EndLineArray()** is called, all the lines are drawn—using the then current pen size—in the order that they were added to the array.

These functions don't change any graphics parameters. For example, they don't move the pen or change the current high and low colors. Parameter values that are in effect when **EndLineArray()** is called are the ones used to draw the lines. The high and low colors are ignored in favor of the *color* specified for each line.

The *count* passed to **BeginLineArray()** is an upper limit on the number of lines that can be drawn. Keeping the count close to accurate and within reasonable bounds helps the efficiency of the line-array mechanism. It's a good idea to keep it less than 256; above that number, memory requirements begin to impinge on performance.

See also: **StrokeLine()**

BeginPicture(), EndPicture()

void **BeginPicture(**BPicture **picture***)**

BPicture ***EndPicture(**void**)**

BeginPicture() instructs the Application Server to begin recording a set of drawing instructions for a *picture*; **EndPicture()** instructs the Server to end the recording session. It returns the same object that was passed to **BeginPicture()**.

The BPicture records exactly what the BView draws—and only what the BView draws—between the **BeginPicture()** and **EndPicture()** calls. The drawing of other views is ignored, as are function calls that don't draw or affect graphics parameters. The picture captures only primitive graphics operations—that is, functions defined in this class, such as **DrawString()**, **FillArc()**, and **SetFont()**. If a complex drawing function (such as **Draw()**) is called, only the primitive operations that it contains are recorded.

A BPicture can be recorded only if the BView is attached to a window. The window it's in can be off-screen and the view itself can be hidden or reside outside the current clipping region. However, if the window is on-screen and the view is visible, the drawing that the BView does will both be captured in the *picture* and rendered in the window.

See also: the BPicture class, **DrawPicture()**

BeginRectTracking(), EndRectTracking()

void **BeginRectTracking(**BRect *rect*, ulong *how* = B_TRACK_WHOLE_RECT**)**

void **EndRectTracking(**void**)**

These functions instruct the Application Server to display a rectangular outline that will track the movement of the cursor. **BeginRectTracking()** puts the rectangle on-screen and initiates tracking; **EndRectTracking()** terminates tracking and removes the rectangle. The initial rectangle, *rect*, is specified in the BView's coordinate system.

This function supports two kinds of tracking, depending on the constant passed as the *how* argument:

B_TRACK_WHOLE_RECT	The whole rectangle moves with the cursor. Its position changes, but its size remains fixed.
B_TRACK_RECT_CORNER	The left top corner of the rectangle remains fixed within the view while its right and bottom edges move with the cursor.

Tracking is typically initiated from within a BView's **MouseDown()** function and is allowed to continue as long as a mouse button is held down. For example:

```
void MyView::MouseDown(BPoint point)
{
    ulong buttons;

    BRect rect(point, point);
    BeginRectTracking(rect, B_TRACK_RECT_CORNER);
    do {
        snooze(30.0 * 1000.0);
        GetMouse(&point, &buttons);
    } while ( buttons );
    EndRectTracking();

    rect.SetRightBottom(point);
    . . .
}
```

This example uses **BeginRectTracking()** to drag out a rectangle from the point recorded for a mouse-down event. It sets up a modal loop to periodically check on the state of the mouse buttons. Tracking ends when the user releases all buttons. The right and bottom sides of the rectangle are then updated from the cursor location last reported by the **GetMouse()** function.

See also: **ConvertToScreen()**, **GetMouse()**

Bounds()

BRect **Bounds(**void**)** const

Returns the BView's bounds rectangle. If the BView is attached to a window, this function gets the current bounds rectangle from the Application Server. If not, it returns a rectangle the same size as the BView's frame rectangle, but with the left and top sides at 0.0.

See also: **Frame()**

ChildAt() *see* Parent()

ConstrainClippingRegion()

virtual void **ConstrainClippingRegion(**BRegion **region***)**

Restricts the drawing that the BView can do to *region*.

The Application Server keeps track of a clipping region for each BView that's attached to a window. It clips all drawing the BView does to that region; the BView can't draw outside of it.

By default, the clipping region contains only the visible area of the view and, during an update, only the area that actually needs to be drawn. By passing a *region* to this function, an application can further restrict the clipping region. When calculating the clipping region, the Server intersects it with the *region* provided. The BView can draw only in areas common to the *region* passed and the clipping region as the Server would otherwise calculate it. The region passed can't expand the clipping region beyond what it otherwise would be.

If called during an update, **ConstrainClippingRegion()** restricts the clipping region only for the duration of the update.

Calls to **ConstrainClippingRegion()** are not additive; each *region* that's passed replaces the one that was passed in the previous call. Passing a **NULL** pointer removes the previous region without replacing it. The function works only for BViews that are attached to a window.

See also: **GetClippingRegion()**, **Draw()**

ConvertToParent(), ConvertFromParent()

BPoint **ConvertToParent(**BPoint *localPoint***)** const
void **ConvertToParent(**BPoint **localPoint***)** const

BRect **ConvertToParent(**BRect *localRect***)** const
void **ConvertToParent(**BRect **localRect***)** const

BPoint **ConvertFromParent(**BPoint *parentPoint***)** const
void **ConvertFromParent(**BPoint **parentPoint***)** const

BRect **ConvertFromParent(**BRect *parentRect***)** const
void **ConvertFromParent(**BRect **parentRect***)** const

These functions convert points and rectangles to and from the coordinate system of the BView's parent. **ConvertToParent()** converts *localPoint* or *localRect* from the BView's coordinate system to the coordinate system of its parent BView. **ConvertFromParent()** does the opposite; it converts *parentPoint* or *parentRect* from the coordinate system of the BView's parent to the BView's own coordinate system.

If the point or rectangle is passed by value, the function returns the converted value. If a pointer is passed, the conversion is done in place.

Both functions fail if the BView isn't attached to a window.

See also: **ConvertToScreen()**

ConvertToScreen(), ConvertFromScreen()

BPoint **ConvertToScreen(**BPoint *localPoint***)** const
void **ConvertToScreen(**BPoint **localPoint***)** const

BRect **ConvertToScreen(**BRect *localRect***)** const
void **ConvertToScreen(**BRect **localRect***)** const

BPoint **ConvertFromScreen(**BPoint *screenPoint***)** const
void **ConvertFromScreen(**BPoint **screenPoint***)** const

BRect **ConvertFromScreen(**BRect *screenRect***)** const
void **ConvertFromScreen(**BRect **screenRect***)** const

ConvertToScreen() converts *localPoint* or *localRect* from the BView's coordinate system to the global screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts *screenPoint* or *screenRect* from the screen coordinate system to the BView's local coordinate system.

If the point or rectangle is passed by value, the function returns the converted value. If a pointer is passed, the conversion is done in place.

The screen coordinate system has its origin, (0.0, 0.0), at the left top corner of the main screen.

Neither function will work if the BView isn't attached to a window.

See also: **BWindow::ConvertToScreen()**, **ConvertToParent()**

CopyBits()

void **CopyBits(**BRect *source*, BRect *destination***)**

Copies the image displayed in the *source* rectangle to the *destination* rectangle, where both rectangles lie within the view and are stated in the BView's coordinate system.

If the two rectangles aren't the same size, the source image is scaled to fit. If not all of the destination rectangle lies within the BView's visible region, the source image is clipped rather than scaled.

If not all of the source rectangle lies within the BView's visible region, only the visible portion is copied. It's mapped to the corresponding portion of the destination rectangle.

The BView is then invalidated so its **Draw()** function will be called to update the part of the destination rectangle that can't be filled with the source image.

The BView must be attached to a window.

CountChildren() *see* Parent()

DetachedFromWindow, AllDetached()

virtual void **DetachedFromWindow(**void**)**

virtual void **AllDetached(**void**)**

Implemented by derived classes to make any adjustments necessary when the BView is about to be removed from a window's view hierarchy. These two functions parallel the more commonly implemented **AttachedToWindow()** and **AllAttached()** functions.

DetachedFromWindow() notifications work their way down the hierarchy of views being detached, followed by **AllDetached()** notifications, which work their way up the hierarchy. The second function call permits an ancestor view to take actions that depend on calculations a descendant might have to make when it's first notified of being detached.

The BView is still attached to the window when both functions are called.

See also: **AttachedToWindow()**

DragMessage()

void **DragMessage(**BMessage **message*, BBitmap **image*, BPoint *point*,
BHandler **replyTarget* = NULL**)**
void **DragMessage(**BMessage **message*, BRect *rect*,
BHandler **replyTarget* = NULL**)**

Initiates a drag-and-drop session. The first argument, *message,* is a BMessage object that bundles the information that will be dragged and dropped on the destination view. Once passed to **DragMessage()**, this object becomes the responsibility of—and will eventually be freed by—the system. You shouldn't free it yourself, try to access it later, or pass it to another function. (Since data is copied when it's added to a BMessage, only the copies are automatically freed, not the originals.)

The second argument, *image*, represents the message on-screen; it's the visible image that the user drags. Like the BMessage, this BBitmap object becomes the responsibility of the system; it will be freed when the message is dropped. If you want to keep the image yourself, make a copy to pass to **DragMessage()**. The image isn't dropped on the destination BView; if you want the destination to have the image, you must add it to the *message* as well as pass it as the *image* argument.

The third argument, *point*, locates the point within the image that's aligned with the hot spot of the cursor—that is, the point that's aligned with the location passed to **MouseDown()** or returned by **GetMouse()**. It's stated within the coordinate system of the source image and should lie somewhere within its bounds rectangle. The bounds rectangle and coordinate system of a BBitmap are set when the object is constructed.

Alternatively, you can specify that an outline of a rectangle, *rect*, should be dragged instead of an image. The rectangle is stated in the BView's coordinate system. (Therefore, a *point* argument isn't needed to align it with the cursor.)

The final argument, *replyTarget*, names the object that you want to handle any message that might be sent in reply to the dragged message. If *replyTarget* is **NULL**, as it is by default, any reply that's received will be directed to the BView object that initiated the drag-and-drop session.

This function works only for BViews that are attached to a window.

See also: **BMessage::WasDropped()**, the BBitmap class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Implemented by derived classes to draw the *updateRect* portion of the view. The update rectangle is stated in the BView's coordinate system. It's the smallest rectangle that encloses the current clipping region for the view.

Since the Application Server won't render anything a BView draws outside its clipping region, applications will be more efficient if they avoid sending drawing instructions to the Server for images that don't intersect with *updateRect*. For more efficiency and precision, you can ask for the clipping region itself (by calling **GetClippingRegion()**) and confine drawing to images that intersect with it.

A BView's **Draw()** function is called (as the result of an update message) whenever the view needs to present itself on-screen. This may happen when:

- The window the view is in is first shown on-screen, or shown after being hidden (see the BWindow version of the **Hide()** function).
- The view is made visible after being hidden (see BView's **Hide()** function).
- Obscured parts of the view are revealed, as when a window is moved from in high of the view or an image is dragged across the view.
- The view is resized.
- The contents of the view are scrolled (see **ScrollBy()**).
- A child view is added, removed, or resized.

- A rectangle has been invalidated that includes at least some of the view (see **Invalidate()**).
- **CopyBits()** can't completely fill a destination rectangle within the view.

See also: **BWindow::UpdateIfNeeded()**, **Invalidate()**, **GetClippingRegion()**

DrawBitmap(), DrawBitmapAsync()

void **DrawBitmap(**const BBitmap **image***)**
void **DrawBitmap(**const BBitmap **image*, BPoint *point***)**
void **DrawBitmap(**const BBitmap **image*, BRect *destination***)**
void **DrawBitmap(**const BBitmap **image*, BRect *source*, BRect *destination***)**

void **DrawBitmapAsync(**const BBitmap **image***)**
void **DrawBitmapAsync(**const BBitmap **image*, BPoint *point***)**
void **DrawBitmapAsync(**const BBitmap **image*, BRect *destination***)**
void **DrawBitmapAsync(**const BBitmap **image*, BRect *source*,
BRect *destination***)**

These functions place a bitmap *image* in the view at the current pen position, at the *point* specified, or within the designated *destination* rectangle. The *point* and the *destination* rectangle are stated in the BView's coordinate system.

If a *source* rectangle is given, only that part of the bitmap image is drawn. Otherwise, the entire bitmap is placed in the view. The *source* rectangle is stated in the internal coordinates of the BBitmap object.

If the source image is bigger than the destination rectangle, it's scaled to fit.

The two functions differ in only one respect: **DrawBitmap()** waits for the Application Server to finish rendering the image before it returns. **DrawBitmapAsync()** doesn't wait; it passes the image to the Server and returns immediately.

See also: "Drawing Modes" on page 27 in the chapter introduction, the BBitmap class

DrawChar()

void **DrawChar(**char *c***)**
void **DrawChar(**char *c*, BPoint *point***)**

Draws the character *c* at the current pen position—or at the *point* specified—and moves the pen to a position immediately to the right of the character. This function is equivalent to passing a string of one character to **DrawString()**. The *point* is specified in the BView's coordinate system.

See also: **DrawString()**

DrawingMode() *see* SetDrawingMode()

DrawPicture()

void **DrawPicture(**const BPicture **picture***)**
void **DrawPicture(**const BPicture **picture*, BPoint *point***)**

Draws the previously recorded *picture* at the current pen position—or at the specified *point* in the BView's coordinate system. The point or pen position is taken as the coordinate origin for all the drawing instructions recorded in the BPicture.

Nothing that's done in the BPicture can affect anything in the BView's graphics state—for example, the BPicture can't reset the current high color or the pen position. Conversely, nothing in the BView's current graphics state affects the drawing instructions captured in the picture. The graphics parameters that were in effect when the picture was recorded determine what the picture looks like.

See also: **BeginPicture()**, the BPicture class

DrawString()

void **DrawString(**const char **string***)**
void **DrawString(**const char **string*, long *length***)**
void **DrawString(**const char **string*, BPoint *point***)**
void **DrawString(**const char **string*, long *length*, BPoint *point***)**

Draws *length* characters of *string*—or, if the number of characters isn't specified, all the characters in the string, up to the null terminator ('\0').

This function places the first character on a baseline that begins at the current pen position—or at the specified *point* in the BView's coordinate system. It moves the pen to the baseline immediately to the right of the last character drawn. A series of simple **DrawString()** calls (with no *point* specified) will produce a continuous string. For example, these two lines of code,

```
DrawString("tog");
DrawString("ether");
```

will produce the same result as this one:

```
DrawString("together");
```

This is a graphical drawing function, so all the characters to be drawn should have visible representations (including whitespace). Control characters (those with values less than **B_SPACE**, 0x20) will be rejected (skipped over) but at a substantial price in performance.

See also: **MovePenBy()**, **SetFontName()**

EndLineArray() *see* BeginLineArray()

EndPicture() *see* BeginPicture()

EndRectTracking() *see* BeginRectTracking()

FillArc() *see* StrokeArc()

FillEllipse() *see* StrokeEllipse()

FillPolygon() *see* StrokePolygon()

FillRect() *see* StrokeRect()

FillRoundRect() *see* StrokeRoundRect()

FillTriangle() *see* StrokeTriangle()

FindView()

BView ***FindView(**const char **name***)** const

Returns the BView identified by *name*, or **NULL** if the view can't be found. Names are assigned by the BView constructor and can be modified by the **SetName()** function inherited from BHandler.

FindView() begins the search by checking whether the BView's name matches *name*. If not, it continues to search down the view hierarchy, among the BView's children and more distant descendants. To search the entire view hierarchy, use the BWindow version of this function.

See also: **BWindow::FindView()**, **BHandler::SetName()**

Flags() *see* SetFlags()

Flush(), Sync()

void **Flush(**void**)** const

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

These functions flush the window's connection to the Application Server. If the BView isn't attached to a window, neither function has an effect.

For reasons of efficiency, the window's connection to the Application Server is buffered. Drawing instructions destined for the Server are placed in the buffer and dispatched as a group when the buffer becomes full. Flushing empties the buffer, sending whatever it contains to the Server, even if it's not yet full.

The buffer is automatically flushed on every update. However, if you do any drawing outside the update mechanism—in response to interface messages, for example—you need to explicitly flush the connection so that drawing instructions won't languish in the buffer while waiting for it to fill up or for the next update. You should also flush it if you call any drawing functions from outside the window's thread.

Flush() simply flushes the buffer and returns. It does the same work as BWindow's function of the same name.

Sync() flushes the connection, then waits until the Server has executed the last instruction that was in the buffer before returning. This alternative to **Flush()** prevents the application from getting ahead of the Server (ahead of what the user sees on-screen) and keeps both processes synchronized.

It's a good idea, for example, to call **Sync()**, rather than **Flush()**, after employing BViews to produce a bitmap image (a BBitmap object). **Sync()** is the only way you can be sure the image has been completely rendered before you attempt to draw with it.

(Note that all BViews attached to a window share the same connection to the Application Server. Calling **Flush()** or **Sync()** for any one of them flushes the buffer for all of them.)

See also: **BWindow::Flush()**, the BBitmap class

Frame()

BRect **Frame(**void**)** const

Returns the BView's frame rectangle. The frame rectangle is first set by the BView constructor and is altered only when the view is moved or resized. It's stated in the coordinate system of the BView's parent.

If the BView is not attached to a window, **Frame()** reports the object's own cached conception of its frame rectangle. If it is attached, **Frame()** reports the Application Server's conception of the rectangle. When a BView is added to a window, its cached rectangle is communicated to the Server. While it remains attached, the functions that move and resize the frame rectangle affect the Server's conception of the view, but don't alter the rectangle kept by the object. Therefore, if the BView is removed from the window, **Frame()** will again report the frame rectangle that it had before it was attached, no matter how much it was moved and resized while it belonged to the window.

See also: **MoveBy()**, **ResizeBy()**, the BView constructor

FrameMoved()

virtual void **FrameMoved(**BPoint *parentPoint***)**

Implemented by derived classes to respond to a notification that the view has moved within its parent's coordinate system. *parentPoint* gives the new location of the left top corner of the BView's frame rectangle.

FrameMoved() is called only if the **B_FRAME_EVENTS** flag is set and the BView is attached to a window.

If the view is both moved and resized, **FrameMoved()** is called before **FrameResized()**. This might happen, for example, if the BView's automatic resizing mode is a combination of **B_FOLLOW_TOP_BOTTOM** and **B_FOLLOW_RIGHT** and its parent is resized both horizontally and vertically.

The default (BView) version of this function is empty.

< Currently, **FrameMoved()** is also called when a hidden window is shown on-screen. >

See also: **MoveBy()**, **BWindow::FrameMoved()**, **SetFlags()**

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Implemented by derived classes to respond to a notification that the view has been resized. The arguments state the new *width* and *height* of the view. The resizing could have been the result of a user action (resizing the window) or of a programmatic one (calling **ResizeTo()** or **ResizeBy()**).

FrameResized() is called only if the **B_FRAME_EVENTS** flag is set and the BView is attached to a window.

BView's version of this function is empty.

See also: **ResizeBy()**, **BWindow::FrameResized()**, **SetFlags()**

GetCharEscapements(), GetCharEdges()

void **GetCharEscapements(**char *charArray*[], long *numChars*,
float *escapementArray*[], float **factor***)** const

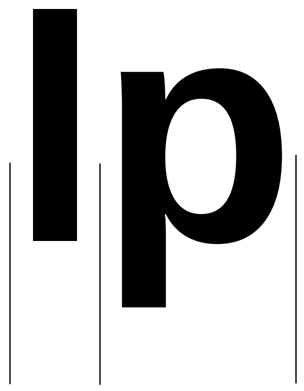
void **GetCharEdges(**char *charArray*[], long *numChars*,
edge_info *edgeArray*[]**)** const

These two functions are designed for programmers who want to precisely position characters on the screen or printed page. For each character passed in the *charArray*, they write information about the horizontal dimension of the character into the *escapementArray* or the *edgeArray.* Both functions assume the BView's current font. Therefore, neither has any effect unless the BView is attached to a window.

< These functions provide inaccurate results for bitmap fonts. >

Escapement. An "escapement" is simply the width of a character recorded in very small units. The units are sufficiently tiny to permit detailed information to be kept in integer form for every character in the font—although declared as **float**s, none of the values in the *escapementArray* have fractional parts. Because the units are small, escapement values are quite large. (The term "escapement" has its historical roots in the fact that the carriage of a typewriter had to move or "escape" a certain distance after each character was typed to make room for the next character.)

The escapement of a character measures the amount of horizontal room it requires when positioned between other characters in a line of text. It includes a measurement of the space required to display the character itself, plus some extra room on the left and right edges to separate the character from its neighbors. In a proportionally spaced font, each character has a distinctive escapement. The illustration below shows the approximate escapements for the letters 'l' and 'p' as they might appear together in a word like "help" or "ballpark." The escapement for each character is the distance between the vertical lines:



GetCharEscapements() measures the same space that functions such as BView's **StringWidth()** and BTextView's **LineWidth()** do, though it measures each character individually and records the result in arbitrary (rather than coordinate) units.

The escapement of a character in a particular font is a constant no matter what the font size. To convert an escapement value to coordinate units, you must multiply it by three values:

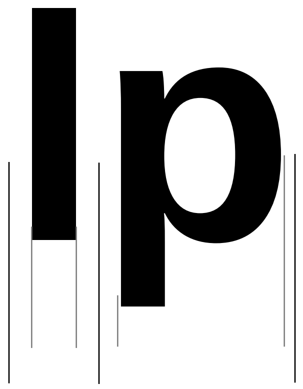
- A floating-point conversion factor,
- The font size (in points), and
- The resolution of the output device.

GetCharEscapements() writes the conversion factor into the variable referred to by *factor.* **GetFontInfo()** can provide the current font size. When the output device is a printer, the resolution should be the actual resolution (the dpi or "dots per inch") at which it prints. When the output device is the screen, the resolution should be 72.0. (This reflects the fact that screen pixels are taken to equal coordinate units—and one coordinate unit is 1/72 of an inch, or roughly equivalent to one typographical point.)

Edges. Edge values measure how far a character outline is inset from its left and right escapement boundaries. **GetCharEdges()** provides edge values in standard coordinate units, not escapement units, that take the size of the current font into account. It places the edge values into an array of **edge_info** structures. Each structure has a **left** and a **right** data member, as follows:

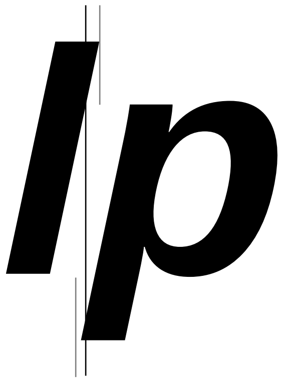
```
typedef struct {
    float left;
    float right;
} edge_info;
```

The illustration below shows typical character edges. As in the illustration above, the solid vertical lines mark escapement boundaries. The dotted lines mark off the part of each escapement that's an edge, the distance between the character outline and the escapement boundary:



This is the normal case. The left edge is a positive value measured rightward from the left escapement boundary. The right edge is a negative value measured leftward from the right escapement boundary.

However, if the characters of a font overlap, the left edge can be a negative value and the right edge can be positive. This is illustrated below:



Note that the italic ‘*l*’ extends beyond its escapement to the right, and that the ‘*p*’ begins before its escapement to the left. In this case, instead of separating the adjacent characters, the edges determine how much they overlap.

Edge values are specific to each character and depend on nothing but the character (and the font). They don’t take into account any contextual information; for example, the right edge for italic ‘*l*’ would be the same no matter what letter followed. Edge values therefore aren’t sufficient to decide how character pairs can be kerned. Kerning is contextually dependent on the combination of two particular characters.

See also: **GetFontInfo()**

GetClippingRegion()

void **GetClippingRegion(**BRegion **region***)** const

Modifies the BRegion object passed as an argument so that it describes the current clipping region of the BView, the region where the BView is allowed to draw. It’s most efficient to allocate temporary BRegions on the stack:

```
BRegion clipper;
GetClippingRegion(&clipper);
. . .
```

Ordinarily, the clipping region is the same as the visible region of the view, the part of the view currently visible on-screen. The visible region is equal to the view’s bounds rectangle minus:

- The frame rectangles of its children,
- Any areas that are clipped because the view doesn’t lie wholly within the frame rectangles of all its ancestors in the view hierarchy, and
- Any areas that are obscured by other windows or that lie in a part of the window that’s off-screen.

The clipping region can be smaller than the visible region if the program restricted it by calling **ConstrainClippingRegion()**. It will exclude any area that doesn't intersect with the region passed to **ConstrainClippingRegion()**.

While the BView is being updated, the clipping region contains just those parts of the view that need to be redrawn. This may be smaller than the visible region, or the region restricted by **ConstrainClippingRegion()**, if:

- The update occurs during scrolling. The clipping region will exclude any of the view's visible contents that the Application Server is able to shift to their new location and redraw automatically.
- The view rectangle has grown (because, for example, the user resized the window larger) and the update is needed only to draw the new parts of the view.
- The update was caused by **Invalidate()** and the rectangle passed to **Invalidate()** didn't cover all of the visible region.
- The update was necessary because **CopyBits()** couldn't fill all of a destination rectangle.

This function works only if the BView is attached to a window. Unattached BViews can't draw and therefore have no clipping region.

See also: **ConstrainClippingRegion()**, **Draw()**, **Invalidate()**

GetFontInfo()

void **GetFontInfo(**font_info **fontInfo***)** const

Writes information about the BView's current font into the structure referred to by *fontInfo*. The **font_info** structure contains the following fields:

font_name **name**	The name of the font, which can be as long as 64 characters, plus a null terminator. The name can be set by BView's **SetFontName()** function.
float **size**	The size of the font in points. It can be set by **SetFontSize()**.
float **shear**	The shear angle, which is 90.0° by default and can vary between 45.0° and 135.0°. It can be set by **SetFontShear()**.
float **rotation**	The angle of rotation, which is 0.0° by default. It's set by **SetFontRotation()**.
float **ascent**	How far characters ascend above the baseline.

float **descent**	How far characters descend below the baseline.
float **leading**	The amount of space separating lines (between the descent of the line above and the ascent of the line below).

The ascent, descent, and leading are measured in coordinate units.

This function works only if the BView is attached to a window.

See also: **SetFontName()**

GetKeys()

long **GetKeys(**key_info **keyInfo*, bool *checkQueue***)**

Writes information about the state of the keyboard into the **key_info** structure referred to by *keyInfo*. This structure contains fields that match the BMessage entries that record information about a key-down event. They are:

ulong **char_code**	An ASCII character value, such as 'a' or **B_BACKSPACE**.
ulong **key_code**	A code identifying the key that produced the character.
ulong **modifiers**	A mask indicating which modifier keys are down and which keyboard locks are on.
uchar **key_states**[16]	A bit array that records the state of all the keys on the keyboard, and all the keyboard locks. This array works identically to the "states" array passed in a key-down message. See "Key States" on page 56 for information on how to read information from the array.

If the *checkQueue* flag is **FALSE**, **GetKeys()** provides information about the current state of the keyboard. When this is the case, the **modifiers** field contains the same information that the **modifiers()** function returns.

However, if the *checkQueue* flag is **TRUE**, **GetKeys()** first checks the message queue to see whether it contains any messages reporting keyboard (key-down or key-up) events. If there are keyboard messages waiting in the queue, it takes the information from the oldest message, places it in the *keyInfo* structure, and removes the message from the queue. Each time **GetKeys()** is called, it gets another keyboard message from the queue. If the queue doesn't contain any keyboard messages, it reports the current state of the keyboard, just as if *checkQueue* were **FALSE**.

When called repeatedly in a loop, **GetKeys()** will empty the queue of keyboard messages and then reflect the current state of the keyboard. In this way, you can be sure that your application has not jumped ahead of the user and overlooked any reports of the user's keyboard actions.

This function never looks at the current message, even if it happens to report a keyboard event and *checkQueue* is **TRUE**. The current message isn't in the queue; to get information about it, you must call BLooper's **CurrentMessage()** function:

```
BMessage *current == myView->Window()->CurrentMessge();
```

If **GetKeys()** takes a keyboard message from the queue, all the **key_info** fields are filled in from the message. However, if it captures the current state of the keyboard, the **char_code** and **key_code** fields are set to 0; these fields are appropriate only for reporting particular events.

GetKeys() returns **B_NO_ERROR** if it was able to get the requested information, and **B_ERROR** if the return results are unreliable.

See also: **KeyDown()**, "Keyboard Information" on page 47 of the chapter introduction, **modifiers()**

GetMouse()

void **GetMouse(**BPoint **cursor*, ulong **buttons*, bool *checkQueue* = TRUE**)** const

Provides the location of the cursor and the state of the mouse buttons. The position of the cursor is recorded in the variable referred to by *cursor*; it's provided in the BView's own coordinates. A bit is set in the variable referred to by *buttons* for each mouse button that's down. This mask may be 0 (if no buttons are down) or it may contain one or more of the following constants:

B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

The cursor doesn't have to be located within the view for this function to work; it can be anywhere on-screen. However, the BView must be attached to a window.

If the *checkQueue* flag is set to **FALSE**, **GetMouse()** provides information about the current state of the mouse buttons and the current location of the cursor.

If *checkQueue* is **TRUE**, as it is by default, this function first looks in the message queue for any pending reports of mouse-moved or mouse-up events. If it finds any, it takes the one that has been in the queue the longest (the oldest message), removes it from the queue, and reports the *cursor* location and *button* states that were recorded in the message. Each **GetMouse()** call removes another message from the queue. If the queue doesn't hold any **B_MOUSE_MOVED** or **B_MOUSE_UP** messages, **GetMouse()** reports the current state of the mouse and cursor, just as if *checkQueue* were **FALSE**.

This function is typically called from within a **MouseDown()** function to track the location of the cursor and wait for the mouse button to go up. By having it check the message queue, you can be sure that you haven't overlooked any of the cursor's movement or missed a mouse-up event (quickly followed by another mouse-down) that might have occurred before the first **GetMouse()** call.

See also: **modifiers()**

HandlersRequested()

virtual void **HandlersRequested(**BMessage **message***)**

Responds to the **B_HANDLERS_REQUESTED** *message* passed as an argument by sending a **B_HANDLERS_INFO** message in reply. The reply message contains BMessenger objects for the BView's children in an entry labeled "handlers".

If the received *message* contains an entry named "index", the BView provides a BMessenger for the child at that index. Otherwise, if the *message* contains an entry labeled "name", the BView provides a BMessenger for the child with that name. If the *message* contains neither an index nor a name, the BView places BMessengers for all its children in the "handlers" array of the reply.

However, if the "index" or "name" doesn't successfully designate a child of the BView, or if the BView doesn't have any children, this function doesn't put any BMessengers in the reply message. Instead, it places an appropriate error code—**B_BAD_INDEX**, **B_NAME_NOT_FOUND**, or **B_ERROR**—in the message under the name "error".

You can override this function to use different protocols for specifying child views, or to prevent the BView from revealing any information about its children.

See also: **BHandler::HandlersRequested()**

Hide(), Show()

virtual void **Hide(**void**)**

virtual void **Show(**void**)**

These functions hide a view and show it again.

Hide() makes the view invisible without removing it from the view hierarchy. The visible region of the view will be empty and the BView won't receive update messages. If the BView has children, they also are hidden.

Show() unhides a view that had been hidden. This function doesn't guarantee that the view will be visible to the user; it merely undoes the effects of **Hide()**. If the view didn't have any visible area before being hidden, it won't have any after being shown again (given the same conditions).

Calls to **Hide()** and **Show()** can be nested. For a hidden view to become visible again, the number of **Hide()** calls must be matched by an equal number of **Show()** calls.

However, **Show()** can only undo a previous **Hide()** call on the same view. If the view became hidden when **Hide()** was called to hide the window it's in or to hide one of its ancestors in the view hierarchy, calling **Show()** on the view will have no effect. For a view to come out of hiding, its window and all its ancestor views must be unhidden.

Hide() and **Show()** can affect a view before it's attached to a window. The view will reflect its proper state (hidden or not) when it becomes attached. Views are created in an unhidden state.

See also: **BWindow::Hide()**, **IsHidden()**

HighColor() *see* SetHighColor()

Invalidate()

void **Invalidate(**BRect *rect***)**
void **Invalidate(**void**)**

Invalidates the *rect* portion of the view, causing update messages—and consequently **Draw()** notifications—to be generated for the BView and all descendants that lie wholly or partially within the rectangle. The rectangle is stated in the BView's coordinate system.

If no rectangle is specified, the BView's entire bounds rectangle is invalidated.

Since only BViews that are attached to a window can draw, only attached BViews can be invalidated.

See also: **Draw()**, **GetClippingRegion()**, **BWindow::UpdateIfNeeded()**

InvertRect()

void **InvertRect(**BRect *rect***)**

Inverts all the colors displayed within the *rect* rectangle. A subsequent **InvertRect()** call on the same rectangle restores the original colors.

The rectangle is stated in the BView's coordinate system.

See also: **system_colors()** global function

IsFocus()

bool **IsFocus(**void**)** const

Returns **TRUE** if the BView is the current focus view for its window, and **FALSE** if it's not. The focus view changes as the user chooses one view to work in and then another—for example, as the user moves from one text field to another when filling out an on-screen form. The change is made programmatically through the **MakeFocus()** function.

See also: **BWindow::CurrentFocus()**, **MakeFocus()**

IsHidden()

bool **IsHidden(**void**)** const

Returns **TRUE** if the view has been hidden by the **Hide()** function, and **FALSE** otherwise.

This function returns **TRUE** whether **Hide()** was called to hide the BView itself, to hide an ancestor view, or to hide the BView's window. When a window is hidden, all its views are hidden with it. When a BView is hidden, all its descendants are hidden with it.

If the view has no visible region—perhaps because it lies outside its parent's frame rectangle or is obscured by a window in front—this function may nevertheless return **FALSE**. It reports only whether the **Hide()** function has been called to hide the view, hide one of the view's ancestors in the view hierarchy, or hide the window where the view is located.

If the BView isn't attached to a window, **IsHidden()** returns the state that it will assume when it becomes attached. By default, views are not hidden.

See also: **Hide()**

IsPrinting()

bool **IsPrinting(**void**)** const

Returns **TRUE** if the BView is being asked to draw for the printer, and **FALSE** if the drawing it produces will be rendered on-screen (or if the BView isn't being asked to draw at all).

This function is typically called from within **Draw()** to determine whether the drawing it does is destined for the printer or the screen. When drawing to the printer, the BView may choose different parameters—such as fonts, bitmap images, or colors—than when drawing to the screen.

See also: the BPrintJob class, **Draw()**

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Implemented by derived classes to respond to a message reporting a key-down event. Whenever a BView is the focus view of the active window, it receives a **KeyDown()** notification for each character the user types, except for those that:

- Are produced while a Command key is held down. Command key events are interpreted as keyboard shortcuts.
- Are produced by the Tab key when an Option key is held down. Option-Tab events are interpreted as instructions to change the focus view (for keyboard navigation).
- Can operate the default button in a window. The BButton object's **KeyDown()** function is called, rather than the focus view's.

The argument, *aChar*, names the character reported in the message. It's an ASCII value that takes into account the affect of any modifier keys that were held down or keyboard locks that were in effect at the time of the keystroke. For example, Shift-*i* is reported as uppercase 'I' (0x49) and Control-*i* is reported as a **B_TAB** (0x09).

The character can be tested against ASCII codes and these constants:

B_BACKSPACE	**B_LEFT_ARROW**	**B_INSERT**
B_ENTER	**B_RIGHT_ARROW**	**B_DELETE**
B_RETURN	**B_UP_ARROW**	**B_HOME**
B_SPACE	**B_DOWN_ARROW**	**B_END**
B_TAB		**B_PAGE_UP**
B_ESCAPE	**B_FUNCTION_KEY**	**B_PAGE_DOWN**

B_ENTER and **B_RETURN** are the same character, a newline ('\n').

Only keys that generate characters produce key-down events; the modifier keys on their own do not.

You can determine which modifier keys were being held down at the time of the event by calling BLooper's **CurrentMessage()** function and looking up the "modifiers" entry in the BMessage it returns. If *aChar* is **B_FUNCTION_KEY** and you want to know which key produced the character, you can look up the "key" entry in the BMessage and test it against these constants:

B_F1_KEY	**B_F6_KEY**	**B_F11_KEY**
B_F2_KEY	**B_F7_KEY**	**B_F12_KEY**
B_F3_KEY	**B_F8_KEY**	**B_PRINT_KEY** (Print Screen)
B_F4_KEY	**B_F9_KEY**	**B_SCROLL_KEY** (Scroll Lock)
B_F5_KEY	**B_F10_KEY**	**B_PAUSE_KEY**

For example:

```
if ( aChar == B_FUNCTION_KEY ) {
    BMessage *msg = Window()->CurrentMessage();
    long key = msg->FindLong("key");
    if ( msg->Error == B_NO_ERROR ) {
        switch ( key ) {
        case B_F1_KEY:
            . . .
            break;
        case B_F2_KEY:
            . . .
            break;
          . . .
        }
    }
}
```

The BView version of **KeyDown()** handles keyboard navigation from view to view through **B_TAB** characters. If the view you define is navigable, its **KeyDown()** function should permit **B_SPACE** characters to operate the object and perhaps allow the arrow keys to navigate inside the view. It should also call the inherited version of **KeyDown()** to enable between-view navigation. For example:

```
void MyView::KeyDown(ulong aChar)
{
    switch ( aChar ) {
    case B_SPACE:
        /* mimic a click in the view */
        break;
    case B_RIGHT_ARROW:
        /* move one position to the right in the view */
        break;
    case B_LEFT_ARROW:
        /* move one position to the left in the view */
        break;
    default:
        inherited::KeyDown(aChar);
        break;
    }
}
```

If your BView is navigable but needs to respond to **B_TAB** characters—for example, if it permits users to insert tabs in a text string—its **KeyDown()** function should simply grab the characters and not pass them to the inherited function. Users will have to rely on the Option-Tab combination to navigate from your view.

See also: "Keyboard Information" on page 47 in the chapter introduction, "**B_KEY_DOWN**" on page 7 in the *Message Protocols* appendix, **BWindow::SetDefaultButton()**, **modifiers()**

LeftTop()

BPoint **LeftTop(**void**)** const

Returns the coordinates of the left top corner of the view—the smallest *x* and *y* coordinate values within the bounds rectangle.

See also: **BRect::LeftTop()**, **Bounds()**

LowColor() *see* SetHighColor()

MakeFocus()

virtual void **MakeFocus(**bool *focused* = TRUE**)**

Makes the BView the current focus view for its window (if the *focused* flag is **TRUE**), or causes it to give up that status (if *focused* is **FALSE**). The focus view is the view that displays the current selection and is expected to handle reports of key-down events when the window is the active window. There can be no more than one focus view per window at a time.

When called to make a BView the focus view, this function invokes **MakeFocus()** for the previous focus view, passing it an argument of **FALSE**. It's thus called twice—once for the new and once for the old focus view.

Calling **MakeFocus()** is the only way to make a view the focus view; the focus doesn't automatically change on mouse-down events. BViews that can display the current selection (including an insertion point) or that can accept pasted data should call **MakeFocus()** in their **MouseDown()** functions.

A derived class can override **MakeFocus()** to add code that takes note of the change in status. For example, a BView that displays selectable data may want to highlight the current selection when it becomes the focus view, and remove the highlighting when it's no longer the focus view. A BView that participates in the keyboard navigation system should visually indicate that it can be operated from the keyboard when it becomes the focus view, and remove that indication when the user navigates to another view and it's notified that it's no longer the focus view.

If the BView isn't attached to a window, this function has no effect.

See also: **BWindow::CurrentFocus()**, **IsFocus()**

MouseDown()

virtual void **MouseDown(**BPoint *point***)**

Implemented by derived classes to respond to a message reporting a mouse-down event within the view. The location of the cursor at the time of the event is given by *point* in the BView's coordinates.

MouseDown() functions are often implemented to track the cursor while the user holds the mouse button down and then respond when the button goes up. You can call the **GetMouse()** function to learn the current location of the cursor and the state of the mouse buttons. For example:

```
void MyView::MouseDown(BPoint point)
{
    ulong buttons;
    . . .
    buttons = Window()->CurrentMessage()->FindLong("buttons");
    while ( buttons ) {
        . . .
        snooze(20.0 * 1000.0);
        GetMouse(&point, &buttons, TRUE);
    }
    . . .
}
```

It's important to snooze between **GetMouse()** calls so that the loop doesn't monopolize system resources; 20,000.0 microseconds is a minimum time to wait.

To get complete information about the mouse-down event, look inside the BMessage object returned by BLooper's **CurrentMessage()** function. The "clicks" entry in the message can tell you if this mouse-down is a solitary event or the latest in a series constituting a multiple click.

The BView version of **MouseDown()** is empty.

See also: "**B_MOUSE_DOWN**" on page 9 in the *Message Protocols* appendix, **GetMouse()**

MouseMoved()

virtual void **MouseMoved(**BPoint *point*, ulong *transit*, BMessage **message***)**

Implemented by derived classes to respond to reports of mouse-moved events associated with the view. As the user moves the cursor over a window, the Application Server generates a continuous stream of messages reporting where the cursor is located.

The first argument, *point*, gives the cursor's new location in the BView's coordinate system. The second argument, *transit*, is one of three constants,

B_ENTERED_VIEW,
B_INSIDE_VIEW, or
B_EXITED_VIEW

which explains whether the cursor has just entered the visible region of the view, is now inside the visible region having previously entered, or has just exited from the view. When the cursor crosses a boundary separating the visible regions of two views (perhaps moving from a parent to a child view, or from a child to a parent), **MouseMoved()** is called for each of the BViews, once with a *transit* code of **B_EXITED_VIEW** and once with a code of **B_ENTERED_VIEW**.

If the user is dragging a bundle of information from one location to another, the final argument, *message*, is a pointer to the BMessage object that holds the information. If a message isn't being dragged, *message* is **NULL**.

A **MouseMoved()** function might be implemented to ignore the **B_INSIDE_VIEW** case and respond only when the cursor enters or exits the view. For example, a BView might alter its display to indicate whether or not it can accept a message that has been dragged to it. Or it might be implemented to change the cursor image when it's over the view.

MouseMoved() notifications should not be used to track the cursor inside a view. Use the **GetMouse()** function instead. **GetMouse()** provides the current cursor location plus information on whether any of the mouse buttons are being held down.

The default version of **MouseMoved()** is empty.

See also: "**B_MOUSE_MOVED**" on page 10 in the *Message Protocols* appendix, **DragMessage()**

MoveBy(), MoveTo()

void **MoveBy(**float *horizontal*, float *vertical***)**

void **MoveTo(**BPoint *point***)**
void **MoveTo(**float *x*, float *y***)**

These functions move the view in its parent's coordinate system without altering its size.

MoveBy() adds *horizontal* coordinate units to the left and right components of the frame rectangle and *vertical* units to the top and bottom components. If *horizontal* and *vertical* are positive, the view moves downward and to the right. If they're negative, it moves upward and to the left.

MoveTo() moves the upper left corner of the view to *point*—or to (*x*, *y*)—in the parent view's coordinate system and adjusts all coordinates in the frame rectangle accordingly.

Neither function alters the BView's bounds rectangle or coordinate system.

None of the values passed to these functions should specify fractional coordinates; the sides of a view must line up on screen pixels. Fractional values will be rounded to the closest whole number.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new location. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame rectangle.

See also: **FrameMoved()**, **ResizeBy()**, **Frame()**

MovePenBy(), MovePenTo(), PenLocation()

void **MovePenBy(**float *horizontal*, float *vertical***)**

void **MovePenTo(**BPoint *point***)**
void **MovePenTo(**float *x*, float *y***)**

BPoint **PenLocation(**void**)** const

These functions move the pen (without drawing a line) and report the current pen location.

MovePenBy() moves the pen *horizontal* coordinate units to the right and *vertical* units downward. If *horizontal* or *vertical* are negative, the pen moves in the opposite direction. **MovePenTo()** moves the pen to *point*—or to (*x*, *y*)—in the BView's coordinate system.

PenLocation() returns the point where the pen is currently positioned in the BView's coordinate system. The default pen position is at (0.0, 0.0).

Some drawing functions also move the pen—to the end of whatever they draw. In particular, this is true of **StrokeLine()**, **DrawString()**, and **DrawChar()**. Functions that stroke a closed shape (such as **StrokeEllipse()**) don't move the pen.

The pen location is a parameter of the BView's graphics environment, which the Application Server maintains. If the BView doesn't belong to a window, **MovePenTo()** and **MovePenBy()** cache the location, so that later, when the BView is attached to a window, it can be handed to the Server to become the initial pen location for the BView. If the BView belongs to a window, these functions alter the Server parameter, but don't change any value that may have previously been cached. **PenLocation()** returns the current pen position if the BView is attached, and the cached value if not.

See also: **SetPenSize()**

MoveTo() *see* MoveBy()

NextSibling() *see* Parent()

Parent(), NextSibling(), PreviousSibling(), ChildAt(), CountChildren()

BView ***Parent(**void**)** const

BView ***NextSibling(**void**)** const

BView ***PreviousSibling(**void**)** const

BView ***ChildAt(**long *index***)** const

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

These functions provide various ways of navigating the view hierarchy. **Parent()** returns the BView's parent view, unless the parent is the top view of the window, in which case it returns **NULL**. It also returns **NULL** if the BView doesn't belong to a view hierarchy and has no parent.

All the children of the same parent are arranged in a linked list. **NextSibling()** returns the next sibling of the BView in the list, or **NULL** if the BView is the last child of its parent. **PreviousSibling()** returns the previous sibling of the BView, or **NULL** if the BView is the first child of its parent.

ChildAt() returns the view at *index* in the list of the BView's children, or **NULL** if the BView has no such child. Indices begin at 0 and there are no gaps in the list. **CountChildren()** returns the number of children the BView has. If the BView has no children, **CountChildren()** returns **NULL**, as will **ChildAt()** for all indices, including 0.

To scan the list of a BView's children, you can increment the index passed to **ChildAt()** until it returns **NULL**. However, it's more efficient to ask for the first child and then use **NextSibling()** to walk down the sibling list. For example:

```
BView *child;
if ( child = myView->ChildAt(0) ) {
    while ( child ) {
        . . .
        child = child->NextSibling();
    }
}
```

See also: **AddChild()**

PenLocation() *see* MovePenBy()

PenSize() *see* SetPenSize()

PreviousSibling() *see* Parent()

Pulse()

virtual void **Pulse(**void**)**

Implemented by derived classes to do something at regular intervals. Pulses are regularly timed events, like the tick of a clock or the beat of a steady pulse. A BView receives **Pulse()** notifications when no other messages are pending, but only if it asks for them with the **B_PULSE_NEEDED** flag.

The interval between **Pulse()** calls can be set with BWindow's **SetPulseRate()** function. The default interval is around 500 milliseconds. The pulse rate is the same for all views within a window, but can vary between windows.

Derived classes can implement a **Pulse()** function to do something that must be repeated continuously. However, for time-critical actions, you should implement your own timing mechanism.

The BView version of this function is empty.

See also: **SetFlags()**, the BView constructor, **BWindow::SetPulseRate()**

RemoveChild()

virtual bool **RemoveChild(**BView **childView***)**

Severs the link between the BView and *childView*, so that *childView* is no longer a child of the BView. The *childView* retains all its own children and descendants, but they become an isolated fragment of a view hierarchy, unattached to a window.

If it succeeds in removing *childView*, this function returns **TRUE**. If it fails, it returns **FALSE**. It will fail if *childView* is not, in fact, a child of the BView.

Removing a BView from a window's view hierarchy also removes it from the BWindow's flat list of BHandler objects; the BView will no longer be eligible to handle messages dispatched by the BWindow.

See also: **AddChild()**, **RemoveSelf()**, **DetachedFromWindow()**

RemoveSelf()

bool **RemoveSelf(**void**)**

Removes the BView from its parent and returns **TRUE**, or returns **FALSE** if the BView doesn't have a parent or for some reason can't be removed from the view hierarchy.

This function acts just like **RemoveChild()**, except that it removes the BView itself rather than one of its children.

See also: **AddChild()**, **RemoveChild()**

ResizeBy(), ResizeTo()

void **ResizeBy(**float *horizontal*, float *vertical***)**

void **ResizeTo(**float *width*, float *height***)**

These functions resize the view, without moving its left and top sides. **ResizeBy()** adds *horizontal* coordinate units to the width of the view and *vertical* units to the height. **ResizeTo()** makes the view *width* units wide and *height* units high. Both functions adjust the right and bottom components of the frame rectangle accordingly.

Since a BView's frame rectangle must be aligned on screen pixels, only integral values should be passed to these functions. Values with fractional components will be rounded to the nearest whole integer.

If the BView is attached to a window, these functions cause its parent view to be updated, so the BView is immediately displayed in its new size. If it doesn't have a parent or isn't attached to a window, these functions merely alter its frame and bounds rectangles.

See also: **FrameResized()**, **MoveBy()**, **BRect::Width()**, **Frame()**

ResizingMode() *see* SetResizingMode()

ScrollBar()

BScrollBar ***ScrollBar(**orientation *posture***)** const

Returns a BScrollBar object that scrolls the BView (that has the BView as its target). The requested scroll bar has the *posture* orientation—**B_VERTICAL** or **B_HORIZONTAL**. If the BView isn't the target of a scroll bar with the specified orientation, this function returns **NULL**.

See also: **ScrollBar::SetTarget()**

ScrollBy(), ScrollTo()

void **ScrollBy(**float *horizontal*, float *vertical***)**

void **ScrollTo(**BPoint *point***)**
void **ScrollTo(**float *x*, float *y***)**

These functions scroll the contents of the view.

ScrollBy() adds *horizontal* to the left and right components of the BView's bounds rectangle, and *vertical* to the top and bottom components. This serves to shift the display *horizontal* coordinate units to the left and *vertical* units upward. If *horizontal* and *vertical* are negative, the display shifts in the opposite direction.

ScrollTo() shifts the contents of the view as much as necessary to put *point*—or (*x*, *y*)—at the upper left corner of its bounds rectangle. The point is specified in the BView's coordinate system.

Anything in the view that was visible before scrolling and also visible afterwards is automatically redisplayed at its new location. The remainder of the view is invalidated, so the BView's **Draw()** function will be called to fill in those parts of the display that were previously invisible. The update rectangle passed to **Draw()** will be the smallest rectangle that encloses just these new areas. If the view is scrolled in only one direction, the update rectangle will be exactly the area that needs to be drawn.

These function don't work on BViews that aren't attached to a window.

See also: **GetClippingRegion()**

SetDrawingMode(), DrawingMode()

virtual void **SetDrawingMode(**drawing_mode *mode***)**

drawing_mode **DrawingMode(**void**)** const

These functions set and return the BView's drawing mode, which can be any of the following nine constants:

B_OP_COPY	**B_OP_MIN**	**B_OP_ADD**
B_OP_OVER	**B_OP_MAX**	**B_OP_SUBTRACT**
B_OP_ERASE	**B_OP_INVERT**	**B_OP_BLEND**

The drawing mode is one element of the BView's graphics environment, which the Application Server maintains. If the BView isn't attached to a window, **SetDrawingMode()** caches the *mode*. When the BView is placed in a window and becomes known to the Server, the cached value is automatically set as the current mode. If the BView belongs to a window, **SetDrawingMode()** changes the current drawing mode, but doesn't alter any value that may have been previously cached. **DrawingMode()** returns the current mode if the view is in a window, and the cached value if not.

The default drawing mode is **B_OP_COPY**. It and the other modes are explained under "Drawing Modes" on page 27 of the introduction to this chapter.

See also: "Drawing Modes" in the chapter introduction

SetFlags(), Flags()

virtual void **SetFlags(**ulong *mask***)**

inline ulong **Flags(**void**)** const

These functions set and return the flags that inform the Application Server about the kinds of notifications the BView should receive. The *mask* set by **SetFlags()** and the return value of **Flags()** is formed from combinations of the following constants:

B_WILL_DRAW,
B_FULL_UPDATE_ON_RESIZE,
B_FRAME_EVENTS, and
B_PULSE_NEEDED

The flags are first set when the BView is constructed; they're explained in the description of the BView constructor.

To set just one of the flags, combine it with the current setting:

```
myView->SetFlags(Flags() | B_FRAME_EVENTS);
```

The *mask* passed to **SetFlags()** and the value returned by **Flags()** can be 0.

See also: the BView constructor, **SetResizingMode()**

SetFontName(), SetFontSize(), SetFontRotation(), SetFontShear()

virtual void **SetFontName(**const char **name***)**

virtual void **SetFontSize(**float *points***)**

virtual void **SetFontRotation(**float *degrees***)**

virtual void **SetFontShear(**float *angle***)**

These functions set characteristics of the font in which the BView draws text. The font is part of the BView's graphics state. It's used by **DrawString()** and **DrawChar()** and assumed by **StringWidth()**, **GetFontInfo()**, and **GetCharEdges()**.

SetFontName() sets the precise name of the font, including the designation of whether it's bold, italic, oblique, black, narrow, or some other style. The name passed to this function must be the same as the name assigned to the font by the vendor. For example, this code

```
SetFontName("Futura II Italic ATT");
```

sets the BView's font to the TrueType™ italic Futura II font.

For **SetFontName()** to be successful, the name it's passed must select a font that's installed on the user's machine. The global **get_font_name()** function can provide the names of all fonts that are currently installed. (Users can see the names listed in the Keyboard application's "Font" menu.)

A handful of fonts are provided with the release, including Arial MT, Baskerville MT, Courier New, Times New Roman, and their stylistic variations. < Additional fonts can be installed by placing them in the proper subdirectory of **/system/fonts** and rebooting the machine. > The names of the bitmap fonts that come with the system are:

Emily
Erich
Kate

At present, they're available in only one size each—12.0 points for Emily and 9.0 points for Erich and Kate. Kate is the default font; it's built into the system. If you ask for a font that isn't available, you'll get Kate instead.

< Currently, you must specifically ask for a bitmap font. In the future, bitmap equivalents to the outline fonts will be automatically provided for on-screen display. >

SetFontSize() sets the size of the font. Valid sizes range from 4 points through 999 points. < Currently, fractional font sizes are not supported. >

SetFontRotation() sets the rotation of the baseline. The baseline rotates counterclockwise from an axis on the left side of the character. The default (horizontal) baseline is at 0°. For example, this code

```
SetFontRotation(45.0);
DrawString("to the northeast");
```

would draw a string that extended upwards and to the right. < Currently, fractional angles of rotation are not supported. >

SetFontShear() sets the angle at which characters are drawn relative to the baseline. The default (perpendicular) shear for all font styles, including oblique and italic ones, is 90.0°. The shear is measured counterclockwise and can be adjusted within the range 45.0° (slanted to the right) through 135.0° (slanted to the left). < Currently, fractional shear angles are not supported. >

The font name, size, rotation, and shear are all elements of the BView's graphics environment, which the Application Server maintains. If the BView isn't attached to a window, these functions cache the values they're passed so that later, when the BView is placed in a window and becomes known to the Server, the cached values can automatically be established as the current font parameters for the BView. If the BView belongs to a window, these functions alter the current parameters, but don't change any values that may have been previously cached.

< The **SetFontSize()**, **SetFontRotation()**, and **SetFontShear()** functions don't work for bitmap fonts. >

Derived classes can override these functions to take any collateral measures required by the font change. For example, BTextView and BListView override them to redisplay the text in the new font.

See also: **GetFontInfo()**, **AttachedToWindow()**, **get_font_name()**

SetHighColor(), HighColor(), SetLowColor(), LowColor()

virtual void **SetHighColor(**rgb_color *color***)**
void **SetHighColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

rgb_color **HighColor(**void**)** const

virtual void **SetLowColor(**rgb_color *color***)**
void **SetLowColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

rgb_color **LowColor(**void**)** const

These functions set and return the current high and low colors of the BView. These colors combine to form a pattern that's passed as an argument to the **Stroke**...**()** and **Fill**...**()** drawing functions. The **B_SOLID_HIGH** pattern is the high color alone, and **B_SOLID_LOW** is the low color alone.

The default high color is black—*red*, *green*, and *blue* values all equal to 0. The default low color is white—*red*, *green*, and *blue* values all equal to 255. < The *alpha* component of the color is currently ignored. >

The versions of **SetHighColor()** and **SetLowColor()** that take separate arguments for the *red*, *blue*, and *green* color components work by creating an **rgb_color** data structure and passing it to the corresponding function that's declared **virtual**. Therefore, if you want to override either of these functions, you should override the virtual version. (However, due to the peculiarities of C++, overriding any version of an overloaded function hides all versions of the function. For continued access to the nonvirtual version without explicitly specifying the "BView::" prefix, you'll need to reimplement it also.)

The high and low colors are parameters of the BView's graphics environment, which is kept in the BView's shadow counterpart in the Application Server. If the BView isn't attached to a window, **SetHighColor()** and **SetLowColor()** cache the *color* value so that later, when the BView is placed in a window and becomes known to the Server, the cached value can automatically be established as the current high or low color for the BView. If the BView belongs to a window, they alter the current parameters, but don't change any values that may have previously been cached. **HighColor()** and **LowColor()** return the current parameters if the BView is in a window, and the cached values if not.

See also: "Patterns" on page 26 of the chapter introduction, **SetViewColor()**

SetPenSize(), PenSize()

virtual void **SetPenSize(**float *size***)**

float **PenSize(**void**)** const

SetPenSize() sets the size of the BView's pen—the graphics parameter that determines the thickness of stroked lines—and **PenSize()** returns the current pen size. The pen size is stated in coordinate units, but is translated to a device-specific number of pixels for each output device.

The pen tip can be thought of as a brush that's centered on the line path and held perpendicular to it. If the brush is broader than one pixel, it paints roughly the same number of pixels on both sides of the path.

The default pen size is 1.0 coordinate unit. It can be set to any non-negative value, including 0.0. If set to 0.0, the size is translated to 1 pixel for all devices. This guarantees that it will always draw the thinnest possible line no matter what the resolution of the device.

Thus, lines drawn with pen sizes of 1.0 and 0.0 will look alike on the screen (one pixel thick), but the line drawn with a pen size of 1.0 will be 1/72 of an inch thick when printed, however many printer pixels that takes, while the line drawn with a 0.0 pen size will be just one pixel thick.

The pen size is a parameter of the BView's graphics environment maintained by the Application Server. If the BView isn't attached to a window, **SetPenSize()** caches the *size* so that later, when the BView is added to a window and becomes known to the Server, the cached value can automatically be established as the initial pen size for the BView. If the BView belongs to a window, this function changes the current pen size, but doesn't alter any value that may have previously been cached. **PenSize()** returns the current pen size if the BView is in a window, and the cached value if not.

See also: "The Pen" on page 24 and "Picking Pixels to Stroke and Fill" on page 34 of the chapter introduction, **StrokeArc()** and the other **Stroke**...**()** functions, **MovePenBy()**

SetResizingMode(), ResizingMode()

virtual void **SetResizingMode(**ulong *mode***)**

inline ulong **ResizingMode(**void**)** const

These functions set and return the BView's automatic resizing mode. The resizing mode is first set when the BView is constructed. The various possible modes are explained where the constructor is described.

See also: the BView constructor, **SetFlags()**

SetSymbolSet()

virtual void **SetSymbolSet(**const char **name***)**

Determines the set of characters that the BView can display. A symbol set maps graphic symbols (glyphs) to character values (ASCII codes). Sets differ mainly in which symbols they associate with character values beyond the traditional ASCII range (above 0x7f), though they sometimes also differ within the traditional range as well.

The default symbol set is “Macintosh”. However, there are many other possibilities to choose from, including:

> “ISO 8859/9 Latin 5”,
> “Legal”,
> “PC-850 Multilingual”, and
> “Windows 3.1 Latin 2”.

The **get_symbol_set_name()** global function can provide a list of all currently available symbol sets.

Except for the bitmap fonts, every font implements every symbol set. However, some fonts may not provide all the characters in every set.

Derived classes can override this function to take any collateral measures required by the change in symbol set. For example, BTextView and BListView override it to recalculate how displayed text is laid out.

The symbol set is part of the BView’s graphics environment, which is to say that the Application Server maintains it. If the BView isn’t attached to a window, **SetSymbolSet()** copies and caches the *name* so that later, when the BView is added to a window and becomes known to the Server, it can automatically be established as the BView’s current symbol set. If the BView belongs to a window, this function changes the current symbol set, but doesn’t alter any string that may have previously been cached.

See also: **SetFontName()**, **get_symbol_set_name()**

SetViewColor(), ViewColor()

> virtual void **SetViewColor(**rgb_color *color***)**
> void **SetViewColor(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**
>
> rgb_color **ViewColor(**void**)** const

These functions set and return the background color that’s shown in all areas of the view rectangle that the BView doesn’t cover with its own drawing. When the clipping region is erased prior to an update, it’s erased to the view color. When a view is resized to expose new areas, the new areas are first displayed in the view color. The default view color is white, which matches the background color of the window’s content area.

If you know that a BView will cover every pixel in the clipping region when it draws, you may want to avoid having the region erased to a color that will immediately be obliterated. If you set the view color to **TRANSPARENT_32_BIT**, the Application Server will not draw its background color before updates nor fill new areas with the background color. (Note that, despite the name, this doesn’t make the view transparent—you can’t see through it to what the view behind it would draw in that region.)

If the view color is anything but white, the **B_WILL_DRAW** flag needs to be set, even if the BView does no other drawing except provide a background color. The flag informs the

Application Server that there are specific drawing operations (in this case, a specific background color) associated with the view.

The version of **SetViewColor()** that takes separate arguments for the *red*, *blue*, and *green* color components works by creating an **rgb_color** data structure and passing it to the corresponding function that's declared **virtual**. Therefore, you need override only the **rgb_color** version to augment both functions. (However, due to the peculiarities of C++, overriding any version of an overloaded function hides all versions of the function. For continued access to the nonvirtual version without explicitly specifying the "BView::" prefix, you'll need to reimplement it also.)

< The *alpha* color component is currently ignored. >

It's best to set the view color before the window is shown on-screen.

The view color is a parameter of the BView's graphics environment, which the Application Server maintains. If the BView doesn't belong to a window, **SetViewColor()** caches the *color* it's passed so that later, when the BView is attached to a window, it can automatically be handed to the Server. If the BView belongs to a window, **SetViewColor()** alters the Server parameter, but doesn't change any value that may have previously been cached. **ViewColor()** returns the current parameter if the BView is attached, and the cached value if not.

See also: "The View Color" on page 22 of the introduction to the chapter, **SetHighColor()**

Show() *see* Hide()

StringWidth()

float **StringWidth(**const char **string***)** const
float **StringWidth(**const char **string*, long *length***)** const

Returns how much room is required to draw *length* characters of *string* in the BView's current font. If no length is specified, the entire string is measured, up to the null character, '\0', which terminates it. The return value totals the width of all the characters. It measures, in coordinate units, the length of the baseline required to draw the string.

This function works only for BViews that are attached to a window (since only attached views have a current font).

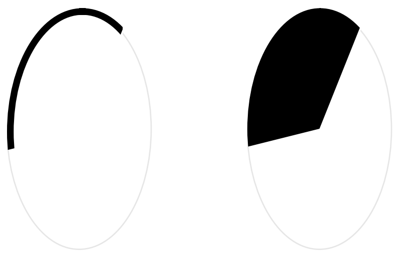
See also: **GetFontInfo()**, **GetCharEscapements()**

StrokeArc(), FillArc()

void **StrokeArc(**BRect *rect*, float *angle*, float *span*,
pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokeArc(**BPoint *center*, float *xRadius*, float *yRadius*, float *angle*, float *span*,
pattern *aPattern* = B_SOLID_HIGH**)**

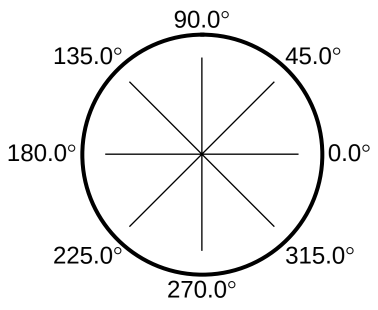
void **FillArc(**BRect *rect*, float *angle*, float *span*,
pattern *aPattern* = B_SOLID_HIGH**)**
void **FillArc(**BPoint *center*, float *xRadius*, float *yRadius*, float *angle*, float *span*,
pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw an arc, a portion of an ellipse. **StrokeArc()** strokes a line along the path of the arc. **FillArc()** fills the wedge defined by straight lines stretching from the center of the ellipse of which the arc is a part to the end points of the arc itself. For example:



The arc is a section of the ellipse inscribed in *rect*—or the ellipse located at *center*, where the horizontal distance from the center to the edge of the ellipse is measured by *xRadius* and the vertical distance from the center to the edge is measured by *yRadius*.

The arc starts at *angle* and stretches along the ellipse for *span* degrees, where angular coordinates are measured counterclockwise with 0° on the right, as shown below:



For example, if *angle* is 180.0° and *span* is 90.0°, the arc would be the lower left quarter of the ellipse. The same arc would be drawn if *angle* were 270.0° and *span* were –90.0°.
< Currently, *angle* and *span* measurements in fractions of a degree are not supported. >

The width of the line drawn by **StrokeArc()** is determined by the current pen size. Both functions draw using *aPattern*—or, if no pattern is specified, using the current high color. Neither function alters the current pen position.

See also: **StrokeEllipse()**

StrokeEllipse(), FillEllipse()

void **StrokeEllipse(**BRect *rect,* pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokeEllipse(**BPoint *center,* float *xRadius*, float *yRadius*,
pattern *aPattern* = B_SOLID_HIGH**)**

void **FillEllipse(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**
void **FillEllipse(**BPoint *center*, float *xRadius*, float *yRadius*,
pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw an ellipse. **StrokeEllipse()** strokes a line around the perimeter of the ellipse and **FillEllipse()** fills the area the ellipse encloses.

The ellipse has its center at *center*. The horizontal distance from the center to the edge of the ellipse is measured by *xRadius* and the vertical distance from the center to the edge is measured by *yRadius*. If *xRadius* and *yRadius* are the same, the ellipse will be a circle.

Alternatively, the ellipse can be described as one that's inscribed in *rect*. If the rectangle is a square, the ellipse will be a circle.

The width of the line drawn by **StrokeEllipse()** is determined by the current pen size. Both functions draw using *aPattern*—or, if no pattern is specified, using the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**

StrokeLine()

void **StrokeLine(**BPoint *start*, BPoint *end*, pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokeLine(**BPoint *end*, pattern *aPattern* = B_SOLID_HIGH**)**

Draws a straight line between the *start* and *end* points—or, if no starting point is given, between the current pen position and *end* point—and leaves the pen at the end point.

This function draws the line using the current pen size and the specified pattern. If no pattern is specified, the line is drawn in the current high color. The points are specified in the BView's coordinate system.

See also: **SetPenSize()**, **BeginLineArray()**

StrokePolygon(), FillPolygon()

void **StrokePolygon(**BPolygon **polygon*,
bool *isClosed* = TRUE, pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*,
bool *isClosed* = TRUE, pattern *aPattern* = B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
bool *isClosed* = TRUE, pattern *aPattern* = B_SOLID_HIGH**)**

void **FillPolygon(**BPolygon **aPolygon*,
pattern *aPattern* = B_SOLID_HIGH**)**
void **FillPolygon(**BPoint **pointList*, long *numPoints*,
pattern *aPattern* = B_SOLID_HIGH**)**
void **FillPolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a polygon with an arbitrary number of sides. **StrokePolygon()** strokes a line around the edge of the polygon using the current pen size. If a *pointList* is specified rather than a BPolygon object, this function strokes a line from point to point, connecting the first and last points if they aren't identical. However, if the *isClosed* flag is **FALSE**, **StrokePolygon()** won't stroke the line connecting the first and last points that define the BPolygon (or the first and last points in the *pointList*). This leaves the polygon open—making it not appear to be a polygon at all, but rather a series of straight lines connected at their end points. If *isClosed* is **TRUE**, as it is by default, the polygon will appear to be a polygon, a closed figure.

FillPolygon() is a simpler function; it fills in the entire area enclosed by the polygon.

Both functions must calculate the frame rectangle of a polygon constructed from a point list—that is, the smallest rectangle that contains all the points in the polygon. If you know what this rectangle is, you can make the function somewhat more efficient by passing it as the *rect* parameter.

Both functions draw using the specified pattern—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**, the BPolygon class

StrokeRect(), FillRect()

void **StrokeRect(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**

void **FillRect(**BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a rectangle. **StrokeRect()** strokes a line around the edge of the rectangle; the width of the line is determined by the current pen size. **FillRect()** fills in the entire rectangle.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**, **StrokeRoundRect()**

StrokeRoundRect(), FillRoundRect()

> void **StrokeRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*,
> pattern *aPattern* = B_SOLID_HIGH**)**
>
> void **FillRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*,
> pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a rectangle with rounded corners. The corner arc is one-quarter of an ellipse, where the ellipse would have a horizontal radius equal to *xRadius* and a vertical radius equal to *yRadius*.

Except for the rounded corners of the rectangle, these functions work exactly like **StrokeRect()** and **FillRect()**.

Both functions draw using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **StrokeRect()**, **StrokeEllipse()**

StrokeTriangle(), FillTriangle()

> void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
> pattern *aPattern* = B_SOLID_HIGH**)**
> void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
> BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**
>
> void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
> pattern *aPattern* = B_SOLID_HIGH**)**
> void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
> BRect *rect*, pattern *aPattern* = B_SOLID_HIGH**)**

These functions draw a triangle, a three-sided polygon. **StrokeTriangle()** strokes a line the width of the current pen size from the first point to the second, from the second point to the third, then back to the first point. **FillTriangle()** fills in the area that the three points enclose.

Each function must calculate the smallest rectangle that contains the triangle. If you know what this rectangle is, you can make the function marginally more efficient by passing it as the *rect* parameter.

Both functions do their drawing using the pattern specified by *aPattern*—or, if no pattern is specified, in the current high color. Neither function alters the current pen position.

See also: **SetPenSize()**

Sync() ***see*** **Flush()**

Window()

BWindow ***Window(**void**)** const

Returns the BWindow to which the BView belongs, or **NULL** if the BView isn't attached to a window. This function returns the same object that **Looper()** (inherited from the BHandler class) does—except that **Window()** returns it more specifically as a pointer to a BWindow and **Looper()** returns it more generally as a pointer to a BLooper.

See also: **BHandler::Looper()** in the Application Kit, **AddChild()**, **BWindow::AddChild()**, **AttachedToWindow()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Implemented by derived classes to take whatever steps are necessary when the BView's window becomes the active window, or when the window gives up that status. If *active* is **TRUE**, the window has become active. If *active* is **FALSE**, it no longer is the active window.

All objects in the view hierarchy receive **WindowActivated()** notifications when the status of the window changes.

BView's version of this function is empty.

See also: **BWindow::WindowActivated()**

BWindow

Derived from:	public BLooper
Declared in:	<interface/Window.h>

Overview

The BWindow class defines an application interface to windows. Each BWindow object corresponds to one window in the user interface.

At the most basic level, it's the Application Server's responsibility to provide an application with the windows it needs. The Server allocates the memory each window requires, renders images in the window on instructions from the application, and manages the user interface. It equips windows with all the accouterments that let users activate, move, resize, reorder, hide, and close them. These user actions are not mediated by the application; they're handled within the Application Server alone. However, the Server sends the application messages notifying it of user actions that affect the window. A class derived from BWindow can implement virtual functions such as **FrameResized()**, **QuitRequested()**, and **WindowActivated()** to respond to these messages.

BWindow objects are the application's interface to the Server's windows:

- Creating a BWindow object instructs the Application Server to produce a window that can be displayed to the user. The BWindow constructor determines what kind of window it will be and how it will behave. The window is initially hidden; the **Show()** function makes it visible on-screen.
- BWindow functions give the application the ability to manipulate the window programmatically—to activate, move, resize, reorder, hide, and close it just as a user might.
- Classes derived from BWindow can implement functions that respond to interface messages affecting the window.

BWindow objects communicate directly with the Server. However, before this communication can take place, the constructor for the BApplication object must establish an initial connection to the Server. You must construct the BApplication object before the first BWindow.

View Hierarchy

A window can display images, but it can't produce them. To draw within a window, an application needs a collection of various BView objects. For example, a window might have several check boxes or radio buttons, a list of names, some scroll bars, and a scrollable display of pictures or text—all provided by objects that inherit from the BView class.

These BViews are created by the application and are associated with the BWindow by arranging them in a hierarchy under a *top view*, a view that fills the entire content area of the window. Views are added to the hierarchy by making them children of views already in the hierarchy, which at the outset means children of the top view.

A BWindow doesn't reveal the identity of its top view, but it does have functions that act on the top view's behalf. For example, BWindow's **AddChild()** function adds a view to the hierarchy as a child of the top view. Its **FindView()** function searches the view hierarchy beginning with the top view.

Window Threads

Each window runs in its own thread—both in the Application Server and in the application. When it's constructed, a BWindow object spawns a *window thread* for the application and begins running a message loop where it receives reports of user actions associated with the window. You don't have to call **Run()** to get the message loop going, as you do for other BLoopers; **Run()** is called for you at construction time.

Actions initiated from a BWindow's message loop are executed in the window's thread. This, of course, includes all actions that are spun off from the original message. For example, if the user clicks a button in a window and this initiates a series of calculations involving a variety of objects, those calculations will be executed in the thread of the window where the button is located (unless the calculation explicitly spawns other threads or posts messages to other BLoopers).

Quitting

To "close" a window is to remove the window from the screen, quit the message loop, kill the window thread, and delete the BWindow object. As is the case for other BLoopers, this process is initiated by a request to quit—a **B_QUIT_REQUESTED** message.

For a BWindow, a request to quit is an event that might be reported from the Application Server (as when the user clicks a window's close button) or from within the application (as when the user clicks a "Close" menu item).

To respond to quit-requested messages, classes derived from BWindow implement **QuitRequested()** functions. **QuitRequested()** can prevent the window from closing, or take whatever action is appropriate before the window is destroyed. It typically interacts with the user, asking, for example, whether recent changes to a document should be saved.

QuitRequested() is a hook function declared in the BLooper class; it's not documented here. See the BLooper class in the Application Kit for information on the function and on how classes derived from BWindow might implement it.

Hook Functions

FrameMoved()	Can be implemented to take note of the fact that the window has moved.
FrameResized()	Can be implemented to take note of the fact that the window has been resized.
MenusWillShow()	Can be implemented to make sure menu data structures are up-to-date before the menus are displayed to the user.
Minimize()	Removes the window from the screen and replaces it with its minimized representation, or restores the window if it was previously minimized; can be reimplemented to provide a different representation for a minimized window.
SavePanelClosed()	Can be implemented to take note when the window's save panel closes.
SaveRequested()	Can be implemented to save the document displayed in the window when the user requests it in the save panel.
ScreenChanged()	Makes sure the window stays visible on-screen when the size of the pixel grid changes; can be implemented to make other adjustments when the screen changes its depth or dimensions.
WindowActivated()	Can be implemented to take whatever action is necessary when the window becomes the active window, or when it loses that status.
WorkspaceActivated()	Can be implemented to take remedial steps when the workspace where the window lives becomes the active workspace, or when it loses that status.
WorkspacesChanged()	Can be implemented to respond when the set of workspaces where the window can be displayed changes.
Zoom()	Zooms the window to a larger size, or from the larger size to its previous state; can be reimplemented to modify the target window size or make other adjustments.

Constructor and Destructor

BWindow()

BWindow(BRect *frame*, const char **title*, window_type *type*, ulong *flags*,
ulong *workspaces* = B_CURRENT_WORKSPACE**)**

Produces a new window with the *frame* content area, spawns a new thread of execution for the window, and begins running a message loop in that thread.

The first argument, *frame*, measures only the content area of the window; it excludes the border and the title tab at the top. The window's top view will be exactly the same size and shape as its frame rectangle—though the top view is located in the window's coordinate system and the window's frame rectangle is specified in the screen coordinate system.

For the window to become visible on-screen, the frame rectangle you assign it must lie within the frame rectangle of the screen. You can find the current dimensions of the screen by calling **get_screen_info()**. In addition, both the width and height of *frame* must be greater than 0.

Since a window is always aligned on screen pixels, the sides of its frame rectangle must have integral coordinate values. Any fractional coordinates that are passed in *frame* will be rounded to the nearest whole number.

The second argument, *title*, does two things: It sets the title the window will display if it has a tab, and it determines the name of the window thread. The thread name is a string that prefixes "w>" to the title in the following format:

```
"w>title"
```

If the *title* is long, only as many characters will be used as will fit within the limited length of a thread name. (Only the thread name is limited, not the window title.) The title (and thread name) can be changed with the **SetTitle()** function.

The *title* can be **NULL** or an empty string.

The *type* of window is set by one of the following constants:

B_MODAL_WINDOW	A modal window, one that disables other activity in the application until the user dismisses it. It has a border but no tab to display a title.
B_BORDERED_WINDOW	An ordinary (nonmodal) window with a border but no title tab.
B_TITLED_WINDOW	A window with a tab that displays its title and a narrow border that's the same on all sides.
B_DOCUMENT_WINDOW	A window with a title tab and a border. The border on the right and bottom sides is a thin line

that's designed to look good with vertical and horizontal scroll bars.

The tab and border are drawn around the window's frame rectangle.

The fourth argument, *flags*, is a mask that determines the behavior of the window. It's formed by combining constants from the following set:

B_NOT_MOVABLE	Prevents the user from being able to move the window. By default, a window with a tab at the top is movable.
B_NOT_H_RESIZABLE	Prevents the user from resizing the window horizontally. A window is horizontally resizable by default.
B_NOT_V_RESIZABLE	Prevents the user from resizing the window vertically. A window is vertically resizable by default.
B_NOT_RESIZABLE	Prevents the user from resizing the window in any direction. This constant is a shorthand that you can substitute for the combination of **B_NOT_H_RESIZABLE** and **B_NOT_V_RESIZABLE**. A window is resizable by default.
B_NOT_CLOSABLE	Prevents the user from closing the window (eliminates the close button from its tab). Windows with title tabs have a close button by default.
B_NOT_ZOOMABLE	Prevents the user from zooming the window larger or smaller (eliminates the zoom button from the window tab). Windows with tabs are zoomable by default.
B_NOT_MINIMIZABLE	Prevents the user from collapsing the window to its minimized form. Windows can be minimized by default.
B_WILL_ACCEPT_FIRST_CLICK	Enables the BWindow to receive mouse-down and mouse-up messages even when it isn't the active window. By default, a click in a window that isn't the active window brings the window to the front and makes it active, but doesn't get reported to the application. If a BWindow accepts the first click, the event gets reported to the application, but it doesn't make the window active. The BView that responds to the mouse-

	down message must take responsibility for activating the window.
B_WILL_FLOAT	Causes the window to float in front of other windows.

If *flags* is 0, the window will be one the user can move, resize, close, and zoom. It won't float or accept the first click.

The final argument, *workspaces*, associates the window with a set of one or more workspaces. Each workspace is identified by a specific bit in a **long** integer; the *workspaces* mask can name up to 32 workspaces. The mask can even name workspaces that don't yet exist. The window will live in those workspaces when and if the user creates them.

Two special values can be passed as the *workspaces* parameter:

B_CURRENT_WORKSPACE	Associates the window with the workspace that's currently displayed on-screen (the active workspace), whatever workspace that happens to be. This is the default choice.
B_ALL_WORKSPACES	Associates the window with all workspaces. The window will show up in all workspaces the user has created and in all future workspaces that will be created.

The window's message loop reads messages delivered to the window and dispatches them by calling a virtual function of the responsible object. The responsible object is usually one of the BViews in the window's view hierarchy. Views are notified of system messages through **MouseDown()**, **KeyDown()**, **MouseMoved()** and other virtual function calls. However, sometimes the responsible object is the BWindow itself. It handles **FrameMoved()**, **QuitRequested()**, **WindowActivated()** and other notifications.

The message loop begins to run when the BWindow is constructed and continues until the window is told to quit and the BWindow object is deleted. Everything the window thread does is initiated by a message of some kind.

See also: **SetFlags()**, **SetTitle()**

~BWindow()

virtual **~BWindow(**void**)**

Frees all memory that the BWindow allocated for itself.

Call the **Quit()** function to destroy the BWindow object; don't use the **delete** operator. **Quit()** does everything that's necessary to shut down the window—such as remove its connection to the Application Server and get rid of its views—and invokes **delete** at the appropriate time.

See also: **Quit()**

Member Functions

Activate()

void **Activate(**bool *flag* = TRUE**)**

Makes the BWindow the active window (if *flag* is **TRUE**), or causes it to relinquish that status (if *flag* is **FALSE**). When this function activates a window, it reorders the window to the front <of its tier>, highlights its tab, and makes it the window responsible for handling subsequent keyboard events. When it deactivates a window, it undoes all these things. It reorders the window to the back <of its tier> and removes the highlighting from its tab. Another window (the new active window) becomes the target for keyboard events.

When a BWindow is activated or deactivated (whether programmatically through this function or by the user), it and all the BViews in its view hierarchy receive **WindowActivated()** notifications.

This function will not activate a window that's hidden.

See also: **WindowActivated()**, **BView::WindowActivated()**

AddChild()

virtual void **AddChild(**BView **aView***)**

Adds *aView* to the hierarchy of views associated with the window, making it a child of the window's top view. However, if *aView* already has a parent, it won't be forcibly removed from that family and adopted into this one. A view can live with but one parent at a time.

This function calls *aView*'s **AttachedToWindow()** function to inform it that it now belongs to the BWindow. Every view that descends from *aView* also becomes attached to the window and receives its own **AttachedToWindow()** notification.

When a BView is attached to a window, it also is added to the BWindow's list of BHandler objects, making it eligible to receive messages the BWindow dispatches. In addition, this function assigns the BWindow as *aView*'s next handler.

See also: **BView::AddChild()**, **BView::AttachedToWindow()**, **RemoveChild()**, **BHandler::SetNextHandler()**

AddShortcut(), RemoveShortcut()

void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message***)**
void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message*,
BHandler **target***)**

void **RemoveShortcut(**ulong *aChar*, ulong *modifiers***)**

These functions set up, and tear down, keyboard shortcuts for the window. A shortcut is a character (*aChar*) that the user can type, in combination with the Command key and possibly one or more other *modifiers* to issue an instruction to the application. For example, Command-*r* might rotate what's displayed within a particular view. The instruction is issued by posting a BMessage to the window thread.

Keyboard shortcuts are commonly associated with menu items. However, *do not* use these functions to set up shortcuts for menus; use the BMenuItem constructor instead. These BWindow functions are for shortcuts that aren't associated with a menu.

AddShortcut() registers a new window-specific keyboard shortcut. The first two arguments, *aChar* and *modifiers*, specify the character and the modifier states that together will issue the instruction. *modifiers* is a mask that combines any of the usual modifier constants (see the **modifiers()** function for the full list). Typically, it's one or more of these four (or it's 0):

B_SHIFT_KEY
B_CONTROL_KEY
B_OPTION_KEY
B_COMMAND_KEY

B_COMMAND_KEY is assumed; it doesn't have to be specified. The character value that's passed as an argument should reflect the modifier keys that are required. For example, if the shortcut is Command-Shift-*C*, *aChar* should be 'C', not 'c'.

The instruction that the shortcut issues is embodied in a model *message* that the BWindow will copy and post whenever it's notified of a key-down event matching the *aChar* and *modifiers* combination (including **B_COMMAND_KEY**).

Before posting the message, it adds one data entry to the copy:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the key-down event occurred, as measured in microseconds from the time the machine was last booted.

The model *message* shouldn't contain an entry of the same name.

The message is posted to the BWindow. If a *target* BHandler object is specified, it will be designated to respond to the message. If a *target* isn't specified, the current focus view will be designated to handle it. If there is no focus view, the BWindow will act as the handler.

The message is dispatched by calling the handler's **MessageReceived()** function. If you add a keyboard shortcut to a window, you must implement a **MessageReceived()** function that can respond to the message the shortcut generates.

(Note, however, that if the *message* has **B_QUIT_REQUESTED** or the constant for another interface message as its **what** data member, the *target* will be ignored and it will be dispatched by calling a specific function, like **QuitRequested()**, not **MessageReceived()**.)

RemoveShortcut() unregisters a keyboard shortcut that was previously added.

See also: **MessageReceived()**, **FilterKeyDown()**, the BMenuItem constructor

Bounds()

BRect **Bounds(**void**)** const

Returns the current bounds rectangle of the window. The bounds rectangle encloses the content area of the window and is stated in the window's coordinate system. It's exactly the same size as the frame rectangle, but its left and top sides are always 0.0.

See also: **Frame()**

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

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

These first of these functions returns the child BView at *index*, or **NULL** if there's no such child of the BWindow's top view. Indices begin at 0 and there are no gaps in the list. The second function returns the number of children the top view has.

See also: **BView::Parent()**

Close() *see* Quit()

CloseSavePanel() *see* RunSavePanel()

ConvertToScreen(), ConvertFromScreen()

BPoint **ConvertToScreen(**BPoint *windowPoint***)** const
void **ConvertToScreen(**BPoint **windowPoint***)** const

BRect **ConvertToScreen(**BRect *windowRect***)** const
void **ConvertToScreen(**BRect **windowRect***)** const

BPoint **ConvertFromScreen(**BPoint *screenPoint***)** const
void **ConvertFromScreen(**BPoint **screenPoint***)** const

BRect **ConvertFromScreen(**BRect *screenRect***)** const
void **ConvertFromScreen(**BRect **screenRect***)** const

These functions convert points and rectangles to and from the global screen coordinate system. **ConvertToScreen()** converts *windowPoint* or *windowRect* from the window coordinate system to the screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts *screenPoint* or *screenRect* from the screen coordinate system to the window coordinate system.

If the point or rectangle is passed by value, the function returns the converted value. If a pointer is passed, the conversion is done in place.

The window coordinate system has its origin, (0.0, 0.0), at the left top corner of the window's content area. The origin of the screen coordinate system is at the left top corner of the main screen.

See also: **BView::ConvertToScreen()**

CurrentFocus(), PreferredHandler()

BView ***CurrentFocus(**void**)** const

virtual BHandler ***PreferredHandler(**void**)** const

Both these functions return the current focus view for the BWindow, or **NULL** if no view is currently in focus. **CurrentFocus()** returns the object as a BView, and **PreferredHandler()** overrides the BLooper function to return it as a BHandler.

The focus view is the BView that's responsible for showing the current selection and handling keyboard messages when the window is the active window.

Various other objects in the Interface Kit, such as BButtons and BMenuItems, call **PreferredHandler()** to discover where they should target messages posted to the BWindow when a specific target hasn't been designated. This mechanism permits these objects to name the current focus view. Thus, a menu item or a control device can be set up to always act on whatever BView happens to be displaying the current selection.

See also: **BView::MakeFocus()**, **BControl::SetTarget()**, **BMenuItem::SetTarget()**, **BLooper::PreferredHandler()**

DefaultButton() *see* SetDefaultButton()

DisableUpdates(), EnableUpdates()

void **DisableUpdates(**void**)**

void **EnableUpdates(**void**)**

These function disable automatic updating within the window, and re-enable it again. Updating is enabled by default, so every user action that changes a view and every program action that invalidates a view's contents causes the view to be automatically redrawn.

This may be inefficient when there are a number of changes to a view, or to a group of views within a window. In this case, you can temporarily disable the updating mechanism by calling **DisableUpdates()**, make the changes, then call **EnableUpdates()** to re-enable updating and have all the changes displayed at once.

See also: **BView::Invalidate()**, **UpdateIfNeeded()**

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BHandler **handler***)**

Overrides the BLooper function to dispatch messages as they're received by the window thread. This function is called for you each time the BWindow takes a message from its queue. It dispatches the message by calling the virtual function that's designated to begin the application's response.

- It dispatches system messages by calling a message-specific virtual function implemented for the BWindow or the responsible BView. See "Hook Functions for Interface Messages" on page 44 in the introduction to this chapter for a list of these functions.
- It defers to the BLooper version of this function to dispatch **B_QUIT_REQUESTED** and **B_HANDLERS_REQUESTED** messages.
- It dispatches other messages by calling the targeted *handler*'s **MessageReceived()** function.

Whenever it's called, **DispatchMessage()** locks the BWindow. The lock remains in place until the window thread's response to the message is complete.

Derived classes can override this function to make it dispatch specific kinds of messages in other ways. For example:

```
void MyWindow::DispatchMessage(BMessage *message)
{
    Lock();
    if ( message->what == MAKE_PREDICTIONS )
        predictor->GuessAbout(message);
    else
        BWindow::DispatchMessage(message);
    Unlock();
}
```

The message loop deletes every message it receives when the function that **DispatchMessage()** calls, and **DispatchMessage()** itself, return. The message should not be deleted in application code (unless **DetachCurrentMessage()** is first called to detach it from the message loop).

See also: the BMessage class, **BLooper::DispatchMessage()**, **BLooper::CurrentMessage()**

EnableUpdates() *see* DisableUpdates()

FindView()

BView ***FindView(**BPoint *point***)** const
BView ***FindView(**const char **name***)** const

Returns the view located at *point* within the window, or the view tagged with *name*. The point is specified in the window's coordinate system (the coordinate system of its top view), which has the origin at the upper left corner of the window's content area.

If no view is located at the point given, or no view within the window has the name given, this function returns **NULL**.

See also: **BView::FindView()**

Flush()

void **Flush(**void**)** const

Flushes the window's connection to the Application Server, sending whatever happens to be in the out-going buffer to the Server. The buffer is automatically flushed on every update and after each message.

This function has the same effect as the **Flush()** function defined for the BView class.

See also: **BView::Flush**

Frame()

BRect **Frame(**void**)** const

Asks the Application Server for the current frame rectangle for the window and returns it. The frame rectangle encloses the content area of the window and is stated in the screen coordinate system. It's first set by the BWindow constructor, and is modified as the window is resized and moved.

See also: **MoveBy()**, **ResizeBy()**, the BWindow constructor

FrameMoved()

virtual void **FrameMoved(**BPoint *screenPoint***)**

Implemented by derived classes to respond to a notification that the window has moved. The move—which placed the left top corner of the window's content area at *screenPoint* in the screen coordinate system—could be the result of the user dragging the window or of the program calling **MoveBy()** or **MoveTo()**. If the user drags the window, **FrameMoved()** is called repeatedly as the window moves. If the program moves the window, it's called just once to report the new location.

The default version of this function does nothing.

See also: **MoveBy()**, "**B_WINDOW_MOVED**" on page 16 in the *Message Protocols* appendix

FrameResized()

virtual void **FrameResized(**float *width*, float *height***)**

Implemented by derived classes to respond to a notification that the window's content area has been resized to a new *width* and *height*. The resizing could be the result of the program calling **ResizeTo()**, **ResizeBy()**, or **Zoom()**—in which case **FrameResized()** is called just once to report the window's new size—or of a user action—in which case it's called repeatedly as the user drags a corner of the window to resize it.

The default version of this function does nothing.

See also: **ResizeBy()**, "**B_WINDOW_RESIZED**" on page 16 in the *Message Protocols* appendix

GetSizeLimits() *see* SetSizeLimits()

HandlersRequested()

virtual void **HandlersRequested(**BMessage **message***)**

Responds to a request for information identifying the BHandlers associated with the BWindow. This function sends a **B_HANDLERS_INFO** reply to the **B_HANDLERS_REQUESTED** *message* it's passed as an argument. The reply has an entry named "handlers" with BMessenger objects corresponding to the requested BHandlers, or one named "error" with an error code.

If the **B_HANDLERS_REQUESTED** *message* has an entry called "class" and that entry contains the string "BView", this function interprets the request as one that concerns the BView objects that are the children of its top view. It limits its search for BHandlers accordingly. Otherwise, the scope of the request is not limited and encompasses all BHandlers that have been added to the window, including all BViews (except the top view).

If the *message* asks for a particular BView with an entry named "index", the BWindow puts a BMessenger in the reply message for the child BView (or the associated BHandler) at the requested index. If not, and if the *message* asks for a particular BView with an entry labeled "name" and the string in the entry matches the name of one of the top view's children (or one of the window's BHandlers), it puts a BMessenger for that object in the reply message.

However, if the *message* doesn't specify a particular object, it supplies BMessengers for all the top view's children (or all the BWindow's BHandlers).

If this function can't supply BMessengers for the specified BHandlers, it doesn't add any BMessengers to the **B_HANDLERS_INFO** message, but places an appropriate error code—**B_BAD_INDEX**, **B_NAME_NOT_FOUND**, or **B_ERROR**—in the message under the name "error".

You can override this function to respond to different protocols for requesting handlers, or to prevent the BWindow's BViews (and BHandlers) from being revealed.

See also: **BView::HandlersRequested()**, **BApplication::HandlersRequested()**

Hide(), Show()

virtual void **Hide(**void**)**

virtual void **Show(**void**)**

These functions hide the window so it won't be visible on-screen, and show it again.

Hide() removes the window from the screen. If it happens to be the active window, **Hide()** also deactivates it. Hiding a window hides all the views attached to the window. While the window is hidden, its BViews respond **TRUE** to **IsHidden()** queries.

Show() puts the window back on-screen. It places the window in front of other windows and makes it the active window.

Calls to **Hide()** and **Show()** can be nested; if **Hide()** is called more than once, you'll need to call **Show()** an equal number of times for the window to become visible again.

A window begins life hidden (as if **Hide()** had been called once); it takes an initial call to **Show()** to display it on-screen.

See also: **IsHidden()**

IsActive()

bool **IsActive(**void**)** const

Returns **TRUE** if the window is currently the active window, and **FALSE** if it's not.

See also: **Activate()**

IsFront()

bool **IsFront(**void**)** const

Returns **TRUE** if the window is currently the frontmost window on-screen, and **FALSE** if it's not.

IsHidden()

bool **IsHidden(**void**)** const

Returns **TRUE** if the window is currently hidden, and **FALSE** if it isn't.

Windows are hidden at the outset. The **Show()** function puts them on-screen, and **Hide()** can be called to hide them again.

If **Show()** has been called to unhide the window, but the window is totally obscured by other windows or occupies coordinates that don't intersect with the physical screen, **IsHidden()** will nevertheless return **FALSE**, even though the window isn't visible.

See also: **Hide()**

IsSavePanelRunning() *see* RunSavePanel()

KeyMenuBar() *see* SetKeyMenuBar()

MenusWillShow()

virtual void **MenusWillShow(**void**)**

Implemented by derived classes to make sure menus are up-to-date before they're placed on-screen. This function is called just before menus belonging to the window are about to be shown to the user. It gives the BWindow a chance to make any required alterations—for example, disabling or enabling particular items—so that the menus are in synch with the current state of the window.

See also: the BMenu and BMenuItem classes

MessageReceived()

virtual bool **MessageReceived(**BMessage **message***)**

Augments the BHandler version of **MessageReceived()** to ensure that **B_KEY_DOWN** messages that find their way to the BWindow object (in the absence of a focus view, for example), are not lost and can contribute to keyboard navigation.

See also: **BHandler::MessageReceived()**

Minimize()

virtual void **Minimize(**bool *minimize***)**

Removes the window from the screen and replaces it with a token representation, if the *minimize* flag is **TRUE**—or restores the window to the screen and removes the token, if *minimize* is **FALSE**.

This function can be called to minimize or unminimize the window. It's also called by the BWindow to respond to **B_MINIMIZE** messages, which are posted automatically when the user double-clicks the window tab to minimize the window, and when the user double-clicks the token to restore the window. It can be reimplemented to provide a different minimal representation for the window.

See also: "**B_MINIMIZE**" on page 9 in the *Message Protocols* appendix, **Zoom()**

MoveBy(), MoveTo()

void **MoveBy(**float *horizontal*, float *vertical***)**

void **MoveTo(**BPoint *point***)**
void **MoveTo(**float *x*, float *y***)**

These functions move the window without resizing it. **MoveBy()** adds *horizontal* coordinate units to the left and right components of the window's frame rectangle and *vertical* units to the frame's top and bottom. If *horizontal* and *vertical* are negative, the window moves upward and to the left. If they're positive, it moves downward and to the right. **MoveTo()** moves the left top corner of the window's content area to *point*—or (*x*, *y*)—in the screen coordinate system; it adjusts all coordinates in the frame rectangle accordingly.

None of the values passed to these functions should specify fractional coordinates; a window must be aligned on screen pixels. Fractional values will be rounded to the closest whole number.

Neither function alters the BWindow's coordinate system or bounds rectangle.

When these functions move a window, a window-moved event is reported to the window. This results in the BWindow's **FrameMoved()** function being called.

See also: **FrameMoved()**

NeedsUpdate()

bool **NeedsUpdate(**void**)** const

Returns **TRUE** if any of the views within the window need to be updated, and **FALSE** if they're all up-to-date.

See also: **UpdateIfNeeded()**

PreferredHandler() *see* CurrentFocus()

PulseRate() *see* SetPulseRate()

Quit(), Close()

virtual void **Quit(**void**)**

inline void **Close(**void**)**

Quit() gets rid of the window and all its views. This function removes the window from the screen, deletes all the BViews in its view hierarchy, destroys the window thread, removes the window's connection to the Application Server, and, finally, deletes the BWindow object.

Use this function, rather than the **delete** operator, to destroy a window. **Quit()** applies the operator after it empties the BWindow of views and severs its connection to the application and Server. It's dangerous to apply **delete** while these connections remain intact.

BWindow's **Quit()** works much like the BLooper function it overrides. When called from the BWindow's thread, it doesn't return. When called from another thread, it returns after all previously posted messages have been responded to and the BWindow and its thread have been destroyed.

Close() is a synonym of **Quit()**. It simply calls **Quit()** so if you override **Quit()**, you'll affect how both functions work.

See also: **BLooper::QuitRequested()**, **BLooper::Quit()**, **BApplication::QuitRequested()**

RemoveChild()

virtual bool **RemoveChild(**BView **aView***)**

Removes *aView* from the BWindow's view hierarchy, but only if *aView* was added to the hierarchy as a child of the window's top view (by calling BWindow's version of the **AddChild()** function).

If *aView* is successfully removed, **RemoveChild()** returns **TRUE**. If not, it returns **FALSE**.

See also: **AddChild()**

RemoveShortcut() *see* AddShortcut()

ResizeBy(), ResizeTo()

void **ResizeBy(**float *horizontal*, float *vertical***)**

void **ResizeTo(**float *width*, float *height***)**

These functions resize the window, without moving its left and top sides. **ResizeBy()** adds *horizontal* coordinate units to the width of the window and *vertical* units to its height. **ResizeTo()** makes the content area of the window *width* units wide and *height* units high. Both functions adjust the right and bottom components of the frame rectangle accordingly.

Since a BWindow's frame rectangle must line up with screen pixels, only integral values should be passed to these functions. Values with fractional components will be rounded to the nearest whole number.

When a window is resized, either programmatically by these functions or by the user, the BWindow's **FrameResized()** virtual function is called to notify it of the change.

See also: **FrameResized()**

RunSavePanel(), CloseSavePanel(), IsSavePanelRunning()

```
long RunSavePanel(const char *tentativeName = NULL,
                  const char *windowTitle = NULL,
                  const char *saveButtonLabel = NULL,
                  const char *cancelButtonLabel = NULL,
                  BMessage *message = NULL)

void CloseSavePanel(void)

bool IsSavePanelRunning(void)
```

RunSavePanel() requests the Browser to display a panel where the user can choose how to save the document displayed in the window. The panel permits the user to navigate the file system and type in file and directory names.

The arguments to this function are all optional. They're used to configure the panel:

- If passed a *tentativeName* for the document displayed in the window, the save panel will place it in a text field where the user can type a name for the file. The name might designate an existing file, or it might simply be a placeholder name like "UNNAMED" or "UNTITLED–3". If a *tentativeName* isn't passed, the text field will be empty.

- If another *windowTitle* is not specified, the title of the window will include the tentative file name. It will be "Save *tentativeName* As..." preceded by the name of the application. The file name is enclosed in quotes. For example:

  ```
  WishMaker : Save “UNTITLED-3” As...
  ```

  If a *tentativeName* isn't passed, the quotes will be empty.

- If a *saveButtonLabel* isn't provided, the principal button in the panel (the default button) will be labeled "Save".

- If a *cancelButtonLabel* isn't provided, the other button in the panel (to the left of the principal button) will be labeled "Cancel".

- If a *message* is passed, it can contain entries that further configure the panel. It also serves as a model for the message that reports the directory and file name the user selected. If a *message* isn't provided, this information will be reported in a standard **B_SAVE_REQUESTED** message.

If the *message* has one or both of the following entries, they will be used to help configure the panel:

Data name	Type code	Description
“directory”	**B_REF_TYPE**	The **record_ref** for the directory that the panel should display when it first comes on-screen. If this entry is absent, the panel will initially display the current directory of the current volume.
“frame”	**B_RECT_TYPE**	A BRect that sets the size and position of the panel in screen coordinates. If this entry is absent, the Browser will choose an appropriate frame rectangle for the panel.

When the user finishes choosing where to save the file and operates the “Save” (or *saveButtonLabel*) button, the file panel sends a message to the BWindow (through the BApplication object). If a customized *message* is provided, it’s used as the model for the message that’s sent. If a *message* isn’t provided, a standard **B_SAVE_REQUESTED** message is sent instead. In either case, it has two data entries:

Data name	Type code	Description
“name”	**B_STRING_TYPE**	The name of the file in which the document should be saved.
“directory”	**B_REF_TYPE**	A **record_ref** reference to the directory where the file should reside.

A **B_SAVE_REQUESTED** message is dispatched by calling the **SaveRequested()** hook function; the “name” and “directory” are passed as arguments to **SaveRequested()**. This function should be implemented to create the file, if necessary, and save the document. **RunSavePanel()** doesn’t do this work; it simply delivers a BMessage object with the information you need to do the job.

A customized *message* works much like the model messages assigned to BControl objects and BMenuItems. The save panel makes a copy of the model, adds the “name” and “directory” entries (as described above) to the copy, and delivers the copy to the BWindow. Other entries in the message remain unchanged.

The *message* can have any command constant you choose. If it’s **B_SAVE_REQUESTED**, the “name” and “directory” will be extracted from the message and passed to **SaveRequested()**. Otherwise, nothing is extracted and the message is dispatched by calling **MessageReceived()**.

The save panel disappears when the user operates the “Save” (or *saveButtonLabel*) button—provided that the message has **B_SAVE_REQUESTED** as the command constant. If it has a customized constant, it remains open until **CloseSavePanel()** is called (or until the application quits). You can choose to leave the panel on-screen if the user hasn’t chosen a valid file name. **IsSavePanelRunning()** will report whether the save panel is currently displayed on-screen. A BWindow can run only one save panel at a time.

The save panel is automatically closed when user operates the "Cancel" (or *cancelButtonLabel*) button. Whenever it's closed, by the user or the application, a **B_PANEL_CLOSED** message is sent to the application and the **SavePanelClosed()** hook function is called.

RunSavePanel() returns **B_NO_ERROR** if it succeeds in getting the Browser to put the panel on-screen. If the Browser isn't running or the save panel already is, it returns **B_ERROR**. If the Browser is running but the application can't communicate with it, it returns an error code that indicates what went wrong; these codes are the same as those documented for the BMessenger class in the Application Kit.

See also: **SaveRequested()**, **SavePanelClosed()**

SavePanelClosed()

virtual void **SavePanelClosed(**BMessage **message***)**

Implemented by derived classes to take note when the save panel is closed. The *message* argument contains information about how the panel was closed and its state at the time it was closed. It has entries under the names "frame" (the panel's frame rectangle), "directory" (the directory the panel displayed), and "canceled" (whether the user closed the panel). Some of this information can be retained to configure the panel the next time it runs.

See also: "**B_PANEL_CLOSED**" on page 12 in the *Message Protocols* appendix, **RunSavePanel()**

SaveRequested()

virtual void **SaveRequested(**record_ref *directory*, const char **filename***)**

Implemented by derived classes to save the document displayed in the window. This function is called when the BWindow receives a **B_SAVE_REQUESTED** message from the save panel. It reports that the user has asked for the file to be saved in the *directory* indicated and assigned the specified *filename*. The file may already exist, or the application may need to create it to carry out the request.

There's no guarantee that the *directory* and *filename* are valid.

If the file can be saved as requested, you may want this function to call **CloseSavePanel()** to remove the panel from the screen. If the file can't be saved, **SaveRequested()** should notify the user. In some cases, you may want to leave the panel on-screen so the user can try again with a different directory or file name.

See also: **RunSavePanel()**

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Implemented by derived classes to respond to a notification that the screen configuration has changed. This function is called for all affected windows when:

- The number of pixels the screen displays (the size of the pixel grid) is altered,
- < The screen changes its location in the screen coordinate system >, or
- The color mode of the screen changes.

frame is the new frame rectangle of the screen, and *mode* is its new color space.

< Currently, there can be only one monitor per machine, so the screen can't change where it's located in the screen coordinate system. >

See also: **set_screen_size()**, "**B_SCREEN_CHANGED**" on page 14 in the *Message Protocols* appendix

SetDefaultButton(), DefaultButton()

void **SetDefaultButton(**BButton **button***)**

BButton ***DefaultButton(**void**)** const

SetDefaultButton() makes *button* the default button for the window—the button that the user can operate by pressing the Enter key even if another BView is the focus view. **DefaultButton()** returns the button that currently has that status, or **NULL** if there is no default button.

At any given time, only one button in the window can be the default. **SetDefaultButton()** may, therefore, affect two buttons: the one that's forced to give up its status as the default button, and the one that acquires that status. Both buttons are redisplayed, so that the user can see which one is currently the default, and both are notified of their change in status through **MakeDefault()** virtual function calls.

If the argument passed to **SetDefaultButton()** is **NULL**, there will be no default button for the window. The current default button loses its status and is appropriately notified with a **MakeDefault()** function call.

The Enter key can operate the default button only while the window is the active window. However, the BButton doesn't have to be the focus view. Normally, the focus view is notified of key-down messages the window receives. But if the character reported is **B_ENTER**, the default button is notified instead (provided there is a default button).

See also: **BButton::MakeDefault()**

SetDiscipline()

void **SetDiscipline(**bool *flag***)**

Sets a *flag* that determines how much programming discipline the system will enforce. When *flag* is **TRUE**, as it is by default, Kit functions will check to be sure various rules are adhered to. For example, most BView functions will require the caller to first lock the window. < Currently, this is the only rule that comes under the discipline flag. > When *flag* is **FALSE**, these rules are not enforced.

The discipline *flag* should be set to **TRUE** while an application is being developed. However, once it has matured, and it's clear that none of the rules are being disobeyed, the *flag* can be set to **FALSE**. This will eliminate various checking operations and improve performance.

See also: "Locking the Window" in the BView class overview

SetKeyMenuBar(), KeyMenuBar()

void **SetKeyMenuBar(**BMenuBar **menuBar***)**

BMenuBar ***KeyMenuBar(**void**)** const

SetKeyMenuBar() makes the specified BMenuBar object the "key" menu bar for the window—the object that's at the root of the menu hierarchy that users can navigate using the keyboard. **KeyMenuBar()** returns the object with key status, or **NULL** if the window doesn't have a BMenuBar object in its view hierarchy.

If a window contains only one BMenuBar view, it's automatically designated the key menu bar. If there's more than one BMenuBar in the window, the last one added to the window's view hierarchy is considered to be the key one.

If there's a "true" menu bar displayed along the top of the window, its menu hierarchy is the one that users should be able to navigate with the keyboard. **SetKeyMenuBar()** can be called to make sure that the BMenuBar object at the root of that hierarchy is the "key" menu bar.

See also: the BMenuBar class

SetPulseRate(), PulseRate()

void **SetPulseRate(**double *microseconds***)**

double **PulseRate(**void**)**

These functions set and return how often **Pulse()** is called for the BWindow's views (how often **B_PULSE** messages are posted to the window). All BViews attached to the same window share the same pulse rate.

By turning on the **B_PULSE_NEEDED** flag, a BView can request periodic **Pulse()** notifications. By default, **B_PULSE** messages are posted every 500,000.0 microseconds, as long as no other messages are pending. Each message causes **Pulse()** to be called once for every BView that requested the notification. There are no pulses if no BViews request them.

SetPulseRate() permits you to set a different interval. The interval set should not be less than 100,000.0 microseconds; differences less than 50,000.0 microseconds may not be noticeable. A finer granularity can't be guaranteed.

Setting the pulse rate to 0.0 disables pulsing for all views in the window.

See also: **BView::Pulse()**, the BView constructor

SetSizeLimits(), GetSizeLimits(), SetZoomLimits()

void **SetSizeLimits(**float *minWidth*, float *maxWidth*,
float *minHeight*, float *maxHeight***)**

void **GetSizeLimits(**float **minWidth*, float **maxWidth*,
float **minHeight*, float **maxHeight***)**

void **SetZoomLimits(**float *maxWidth*, float *maxHeight***)**

These functions set and report limits on the size of the window. The user won't be able to resize the window beyond the limits set by **SetSizeLimits()**—to make it have a width less than *minWidth* or greater than *maxWidth*, nor a height less than *minHeight* or greater than *maxHeight*. By default, the minimums are sufficiently small and the maximums sufficiently large to accommodate any window within reason.

SetSizeLimits() constrains the user, not the programmer. It's legal for an application to set a window size that falls outside the permitted range. The limits are imposed only when the user attempts to resize the window; at that time, the window will jump to a size that's within range.

GetSizeLimits() writes the current limits to the variables provided.

SetZoomLimits() sets the maximum size that the window will zoom to (when the **Zoom()** function is called). The maximums set by **SetSizeLimits()** also apply to zooming; the window will zoom to the screen size or to the smaller of the maximums set by these two functions.

Since the sides of a window must line up on screen pixels, the values passed to both **SetSizeLimits()** and **SetZoomLimits()** should be whole numbers.

See also: the BWindow constructor, **Zoom()**

SetTitle(), Title()

> void **SetTitle(**const char **newTitle***)**
>
> const char ***Title(**void**)** const

These functions set and return the window's title. **SetTitle()** replaces the current title with *newTitle*. It also renames the window thread in the following format:

```
"w>newTitle"
```

where as many characters of the *newTitle* are included in the thread name as will fit.

Title() returns a pointer to the current title. The returned string is null-terminated. It belongs to the BWindow object, which may alter the string or free the memory where it resides without notice. Applications should ask for the title each time it's needed and make a copy for their own purposes.

A window's title and thread name are originally set by an argument passed to the BWindow constructor.

See also: the BWindow constructor

SetWorkspaces(), Workspaces()

> void **SetWorkspaces(**ulong *workspaces***)**
>
> ulong **Workspaces(**void**)** const

These functions set and return the set of workspaces where the window can be displayed. The *workspaces* argument passed to **SetWorkspaces()** and the value returned by **Workspaces()** is a bitfield with one bit set for each workspace in which the window can appear. Usually a window appears in just one workspace.

SetWorkspaces() can associate a window with workspaces that don't exist yet. The window will appear in those workspaces if and when the user creates them.

You can pass **B_CURRENT_WORKSPACE** as the *workspaces* argument to place the window in the workspace that's currently displayed (the active workspace) and remove it from all others, or **B_ALL_WORKSPACES** to make sure the window shows up in all workspaces, including any new ones that the user might create. **Workspaces()** may return **B_ALL_WORKSPACES**, but will identify the current workspace rather than return **B_CURRENT_WORKSPACE**.

Changing a BWindow's set of workspaces causes it to be notified with a **WorkspacesChanged()** function call.

See also: the BWindow constructor, **WorkspacesChanged()**

SetZoomLimits() *see* SetSizeLimits()

Show() *see* **Hide()**

Title() *see* **SetTitle()**

UpdateIfNeeded()

void **UpdateIfNeeded(**void**)**

Causes the **Draw()** virtual function to be called immediately for each BView object that needs updating. If no views in the window's hierarchy need to be updated, this function does nothing.

BView's **Invalidate()** function generates an update message that the BWindow receives just as it receives other messages. Although update messages take precedence over other kinds of messages the BWindow receives, the window thread can respond to only one message at a time. It will update the invalidated view as soon as possible, but it must finish responding to the current message before it can get the update message.

This may not be soon enough for a BView that's engaged in a time-consuming response to the current message. **UpdateIfNeeded()** forces an immediate update, without waiting to return the BWindow's message loop. However, it works only if called from within the BWindow's thread.

(Because the message loop expedites the handling of update messages, they're never considered the current message and are never returned by BLooper's **CurrentMessage()** function.)

See also: **BView::Draw()**, **BView::Invalidate()**, **NeedsUpdate()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Implemented by derived classes to make any changes necessary when the window becomes the active window, or when it ceases being the active window. If *active* is **TRUE**, the window has just become the new active window, and if *active* is **FALSE**, it's about to give up that status to another window.

The BWindow receives a **WindowActivated()** notification whenever its status as the active window changes. Each of its BViews is also notified.

See also: **BView::WindowActivated()**

WindowType()

inline window_type **WindowType(**void**)** const

Returns what type of window it is. The type is set at construction as one of the following constants:

B_MODAL_WINDOW
B_BORDERED_WINDOW
B_TITLED_WINDOW
B_DOCUMENT_WINDOW

See also: the BWindow constructor

Workspaces() *see* SetWorkspaces()

WorkspaceActivated()

virtual void **WorkspaceActivated(**long *workspace*, bool *active***)**

Implemented by derived classes to respond to a notification that the workspace displayed on the screen has changed. All windows in the newly activated workspace as well as those in the one that was just deactivated get this notification.

The *workspace* argument identifies the workspace in question and the *active* flag conveys its current status. If *active* is **TRUE**, the workspace has just become the active workspace. If *active* is **FALSE**, it has just stopped being the active workspace.

The default (BWindow) version of this function is empty.

See also: "**B_WORKSPACE_ACTIVATED**" on page 16 in the *Message Protocols* appendix

WorkspacesChanged()

virtual void **WorkspacesChanged(**ulong *oldWorkspaces*, ulong *newWorkspaces***)**

Implemented by derived classes to respond to a notification the the window has just changed the set of workspaces in which it can be displayed from *oldWorkspaces* to *newWorkspaces*. This typically happens when the user moves a window from one workspace to another, but it may also happen when a programmatic change is made to the set of permitted workspaces.

The default (BWindow) version of this function is empty.

See also: "**B_WORKSPACES_CHANGED**" on page 17 in the *Message Protocols* appendix, **SetWorkspaces()**

Zoom()

void **Zoom(**void**)**
virtual void **Zoom(**BPoint *leftTop*, float *width*, float *height***)**

Zooms the window to a larger size—or, if already zoomed larger, restores it to its previous size.

The simple version of this function can be called to simulate the user operating the zoom button in the window tab. It resizes the window to the full size of the screen, or to the size previously set by **SetSizeLimits()** and **SetZoomLimits()**. However, if the width and height of the window are both within five coordinate units of the fully zoomed size, it restores the window to the size it had before being zoomed.

To actually change the window's size, the simple version of **Zoom()** calls the virtual version. The virtual version is also called by the system in response to a **B_ZOOM** system message. The system generates this message when the user clicks the zoom button in the window's title tab.

The arguments to the virtual version propose a *width* and *height* for the window and a location for the left top corner of its content area in the screen coordinate system. It can be overridden to change these dimensions or to resize the window differently.

Zoom() may both move and resize the window, resulting in **FrameMoved()** and **FrameResized()** notifications.

See also: **SetSizeLimits()**, **ResizeBy()**

Global Functions

This section describes the global (nonmember) functions defined in the Interface Kit. All these functions deal with aspects of the system-wide environment for the user interface—the keyboard and mouse, the screen, workspaces, installed fonts and symbol sets, the list of possible colors, and various user preferences.

The Application Server maintains this environment (with just a few exceptions). Therefore, for a global Interface Kit function to work, your application must be connected to the Server. The connection these functions depend on is the one that's established when the BApplication object is constructed. Consequently, none of them should be called before a BApplication object is present in your application.

activate_app()

> <interface/InterfaceDefs.h>
>
> void **activate_app(**team_id *app***)**

Activates the *app* application < by bringing one of its windows to the front and making it the active window >. This function works only if the target application has a window on-screen. The newly activated application is notified with a **B_APP_ACTIVATED** message.

< This function is an alternative to sending the application a **B_ACTIVATE** message. It accomplishes the same thing, except that it communicates directly with the Application Server to do its work. >

See also: **BApplication::Activate()** in the Application Kit

activate_workspace(), current_workspace()

> <interface/InterfaceDefs.h>
>
> void **activate_workspace(**long *workspace***)**
>
> long **current_workspace(**void**)**

These functions set and return the active workspace, the one that's currently displayed on-screen. Each workspace is represented by a bit in a **long** integer.

See also: **BWindow::WorkspaceActivated()**

adjust_crt() *see* get_screen_info()

count_fonts() *see* get_font_name()

count_screens() *see* get_screen_info()

count_symbol_sets() *see* get_symbol_set_name()

count_workspaces() *see* set_workspace_count()

current_workspace() *see* activate_workspace()

desktop_color() *see* set_desktop_color()

get_click_speed() *see* set_click_speed()

get_dock_width()

<interface/InterfaceDefs.h>

long **get_dock_width(**float **width***)**

Writes the current width of the dock into the variable referred to by *width*. Since the dock floats on top of other windows, this function can help determine how much usable screen space is actually available. It returns **B_NO_ERROR** if successful and **B_ERROR** if not.

See also: **get_screen_info()**

get_font_name(), count_fonts()

<interface/InterfaceDefs.h>

void **get_font_name(**long *index*, font_name **name***)**

long **count_fonts(**void**)**

These two functions are used in combination to get the names of all installed fonts. For example:

```
long numFonts = count_fonts();
font_name buf;

for ( long i = 0; i < numFonts; i++ ) {
    get_font_name(i, &buf);
    . . .
}
```

The names of all installed fonts are kept in an alphabetically ordered list. **get_font_name()** reads one of the names from the list, the name at *index*, and copies it into the *name* buffer. Font names can be up to 64 characters long, plus a null terminator. Indices begin at 0.

count_fonts() returns the number of fonts currently installed, the number of names in the list.

See also: **BView::GetFontInfo()**, **BView::SetFontName()**

get_key_repeat_delay() *see* set_key_repeat_rate()

get_key_repeat_rate() *see* set_key_repeat_rate()

get_keyboard_id()

<interface/InterfaceDefs.h>

long **get_keyboard_id(**ushort **theId***)**

Obtains the keyboard identifier from the Application Server and writes it into the variable referred to by *theId*. This number reveals what kind of keyboard is currently attached to the computer.

The identifier for the standard 101-key keyboard—and for keyboards with a similar set of keys—is 0x83ab. < Currently, this is the only value this function can provide. > See “Key Codes” on page 48 for illustrations showing the keys found on a standard keyboard.

If unsuccessful for any reason, **get_keyboard_id()** returns **B_ERROR**. If successful, it returns **B_NO_ERROR**.

get_menu_info() *see* set_menu_info()

get_mouse_map() *see* set_mouse_map()

get_mouse_speed() *see* set_mouse_map()

get_mouse_type() *see* set_mouse_map()

get_screen_info(), count_screens()

<interface/InterfaceDefs.h>

void **get_screen_info(**screen_info **theInfo***)**
void **get_screen_info(**long *index*, screen_info **theInfo***)**

long **count_screens(**void**)**

long **set_screen_space(**long *index*, ulong *space*, bool *makeDefault* = TRUE**)**

long **set_screen_refresh_rate(**long *index*, float *rate*, bool *makeDefault* = TRUE**)**

long **adjust_crt(**long *index*, uchar *hPosition*, uchar *vPosition*,
uchar *hSize*, uchar *vSize*, bool *makeDefault* = TRUE**)**

These functions provide information about the screens (monitors) that are currently hooked up to the BeBox, and alter screen parameters.

Each screen that's attached to the machine is identified by an index into a system-wide screen list. The screen at index 0 is the one that has the origin of the screen coordinate system at its left top corner. Other screens in the list are unordered; they're located elsewhere in the coordinate system that the first screen defines. < Currently, multiple screens are not supported, so the screen at index 0 is the only one in the list. Therefore, the *index* passed to these functions should always be 0. >

count_screens() returns the number of screens (monitors) that are attached to the computer. < Since no more than one screen can be attached, this function currently always returns 1. >

get_screen_info() writes information about the screen at *index* into the **screen_info** structure referred to by *theInfo*. If no index is mentioned, it assumes the screen at index 0. The **screen_info** structure contains the following fields:

color_space **mode**	The depth and color interpretation of pixel data in the screen's frame buffer; currently, the mode will be either **B_COLOR_8_BIT** or **B_RGB_32_BIT**. (See "Colors" on page 25 of the chapter introduction for an explanation of the various **color_space** modes.)
BRect **frame**	The frame rectangle of the screen—the rectangle that defines the size and location of the screen in the screen coordinate system.
ulong **spaces**	A mask that enumerates all the possible configurations of the screen space. The consonant values that can contribute to the mask are listed below.
float **min_refresh_rate**	The maximum possible refresh rate in cycles per second.
float **max_refresh_rate**	The minimum possible refresh rate (which may be the same as the maximum).

float **refresh_rate**	The current refresh rate.
uchar **h_position**	The current horizontal position of the CRT display on the monitor, a value between 0 (as far to the left as possible) and 100 (as far to the right as possible) with 50 as the default.
uchar **v_position**	The current vertical position of the CRT display on the monitor, a value between 0 (as close to the top as possible) and 100 (as close to the bottom as possible) with 50 as the default.
uchar **h_size**	The current horizontal size of the CRT display on the monitor, a value between 0 (as narrow as possible) and 100 (as wide as possible) with 50 as the default.
uchar **v_size**	The current vertical size of the CRT display on the monitor, a value between 0 (as short as possible) and 100 (as tall as possible) with 50 as the default.

If the color space **mode** is **B_COLOR_8_BIT**, each pixel value in the frame buffer for the screen is an 8-bit color index. In **B_RGB_32_BIT** mode, each value is a set of four 8-bit color components (red, green, blue, and alpha). The components will be arranged in the most natural order for the display device—typically blue, green, red, and alpha. You can access the frame buffer only through the BWindowScreen class in the Game Kit.

The **spaces** field is a mask that enumerates all the possible configurations of the screen space (its depth and dimensions). It's formed from the following constants:

B_8_BIT_640x400		
B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

For example, if the mask includes **B_32_BIT_1280x1024**, the frame buffer can be 32 bits deep (the **B_RGB_32_BIT** color space) while the screen grid is 1,280 pixels wide and 1,024 pixels high. Not all configurations are possible for all graphics cards. < The operating system currently doesn't support depths of 16 bits. >

The current screen configuration can be read from the **mode** and **frame** fields. To change the configuration, you can pass one of the **spaces** constants to **set_screen_space()**. When the configuration of the screen changes, every affected BWindow object is notified with a **ScreenChanged()** function call. < Since there's currently only one screen, all windows are affected and all, whether on-screen or hidden, receive **ScreenChanged()** notifications. >

The refresh rate for the screen can be changed by passing a new *rate* to **set_screen_refresh_rate()**. The rate should lie between the minimum and maximum reported by **get_screen_info()**. The requested change is made to the best of the ability of the graphics card driver; exact compliance is not promised.

The **h_position**, **v_position**, **h_size**, and **v_size** fields of the **screen_info** structure record the placement of the CRT display on the physical monitor, as set by software controls—not the hardware controls on the monitor itself. If the monitor and the driver for the graphics card permit CRT adjustments through software, **adjust_crt()** can be called to change any setting. Its *hPosition*, *vPosition*, *hSize*, and *vSize* arguments have the same meaning as the corresponding fields of **screen_info**.

The three functions that alter screen parameters—**adjust_crt()**, **set_screen_space()**, and **set_screen_refresh_rate()**—all make changes that take effect immediately. If the *makeDefault* flag is **TRUE**, the new setting also becomes the default and will be used the next time the machine is turned on. If *makeDefault* is **FALSE**, the setting is in effect for the current session only. Each function returns **B_NO_ERROR** if successful, and **B_ERROR** if not.

These three functions are designed for preferences applications—like the Screen application—that permit users to make system-wide choices. Other applications should respect those choices and refrain from modifying them.

get_screen_info() reports on the screen as it is known to the Application Server. If you bypass the Server with the Game Kit, it may not provide accurate information.

See also: **BWindow::ScreenChanged()**, *The Game Kit* chapter

get_scroll_bar_info() *see* set_scroll_bar_info()

get_symbol_set_name(), count_symbol_sets()

<interface/InterfaceDefs.h>

void **get_symbol_set_name(**long *index*, symbol_set_name **name***)**

long **count_symbol_sets(**void**)**

These functions are used to get the names of all available symbol sets. They work much like the parallel font functions **get_font_name()** and **count_fonts()**.

A symbol set associates character symbols (glyphs) with character codes (ASCII values). They differ mainly in how they extend the standard ASCII set—how they assign characters to codes above 0x7f.

get_symbol_set_name() gets one name from the list of symbol sets, the name at *index*, and copies it into the *name* buffer. **count_symbol_sets()** returns the total number of symbol sets (the number of names in the list).

Unlike font names, the names of symbol sets are not arranged alphabetically.

Every font implements every symbol set. However, some fonts implement particular sets more fully than others—that is, some characters in a symbol set may not be available in some fonts. But the position of each character in the set (its character code) remains the same across all fonts.

See also: **BView::SetSymbolSet()**, **get_font_name()**

idle_time()

double **idle_time(**void**)** const

Returns the number of microseconds since the user last manipulated the mouse or keyboard. This information isn't specific to a particular application; **idle_time()** tells you when the user last directed an action at *any* application, not just yours.

index_for_color()

<interface/InterfaceDefs.h>

uchar **index_for_color(**rgb_color *aColor***)**
uchar **index_for_color(**uchar *red*, uchar *green*, uchar *blue*, uchar *alpha* = 0**)**

Returns an index into the list of 256 colors that comprise the **B_COLOR_8_BIT** color space. The value returned picks out the listed color that most closely matches a full 32-bit color—specified either as an **rgb_color** value, *aColor*, or by its *red*, *green*, and *blue* components. < (The *alpha* component is currently ignored.) >

The returned index identifies a color in the **B_COLOR_8_BIT** color space. It can, for example, be passed to BBitmap's **SetBits()** function.

To find the fully specified color that an index picks out, you have to get the color list from the system color map. For example, if you first obtain the index for the "best fit" color that most closely matches an arbitrary color,

```
uchar index = index_for_color(134, 210, 6);
```

you can then use the index to locate that color in the color list:

```
rgb_color bestFit = system_colors()->color_list[index];
```

See also: **system_colors()**, the BBitmap class

modifiers()

<interface/InterfaceDefs.h>

ulong **modifiers(**void**)**

Returns a mask that has a bit set for each modifier key the user is holding down and for each keyboard lock that's set. The mask can be tested against these constants:

B_SHIFT_KEY	**B_COMMAND_KEY**	**B_CAPS_LOCK**
B_CONTROL_KEY	**B_MENU_KEY**	**B_SCROLL_LOCK**
B_OPTION_KEY		**B_NUM_LOCK**

No bits are set (the mask is 0) if no locks are on and none of the modifiers keys are down.

If it's important to know which physical key the user is holding down, the one on the right or the one on the left, the mask can be further tested against these constants:

B_LEFT_SHIFT_KEY	**B_RIGHT_SHIFT_KEY**
B_LEFT_CONTROL_KEY	**B_RIGHT_CONTROL_KEY**
B_LEFT_OPTION_KEY	**B_RIGHT_OPTION_KEY**
B_LEFT_COMMAND_KEY	**B_RIGHT_COMMAND_KEY**

By default, on a 101-key keyboard, the keys labeled "Alt(ernate)" function as the Command modifiers, the key on the right labeled "Control" functions as the right Option key, and only the left "Control" key is available to function as a Control modifier. However, users can change this configuration with the Keymap application.

See also: "Modifier Keys" on page 51 of the introduction to the chapter, **system_key_map()**, **BView::GetKeys()**

restore_key_map() *see* system_key_map()

set_click_speed(), get_click_speed()

<interface/InterfaceDefs.h>

long **set_click_speed(**double *interval***)**

long **get_click_speed(**double **interval***)**

These functions set and report the timing for multiple-clicks. For successive mouse-down events to count as a multiple-click, they must occur within the *interval* set by **set_click_speed()** and provided by **get_click_speed()**. The interval is measured in microseconds; it's usually set by the user in the Mouse preferences application. The smallest possible interval is 100,000 microseconds (0.1 second).

If successful, these functions return **B_NO_ERROR**; if unsuccessful, they return an error code, which may be just **B_ERROR**.

See also: **set_mouse_map()**

set_desktop_color(), desktop_color()

<interface/InterfaceDefs.h>

void **set_desktop_color(**rgb_color *color*, bool *makeDefault* = TRUE**)**

rgb_color **desktop_color(**void**)**

These functions set and return the color of the so-called "desktop"—the bare backdrop against which windows are displayed. The color is the same for all screens attached to the same machine (however, the Workspaces application can arrange for each workspace to have a different background color). **set_desktop_color()** makes an immediate change in the desktop color displayed on-screen; **desktop_color()** returns the color currently displayed.

If the *makeDefault* flag is **TRUE**, the *color* that's set becomes the default color for the desktop; it's the color that will be shown the next time the machine is booted. If the flag is **FALSE**, the color is set only for the current session.

Users can change the default color with the Screen application found in **/preferences**.

set_key_repeat_rate(), get_key_repeat_rate(), set_key_repeat_delay(), get_key_repeat_delay()

<interface/InterfaceDefs.h>

long **set_key_repeat_rate(**int *rate***)**

long **get_key_repeat_rate(**int **rate***)**

long **set_key_repeat_delay(**double *delay***)**

long **get_key_repeat_delay(**double **delay***)**

These functions set and report the timing of repeating keys. When the user presses a character key on the keyboard, it produces an immediate **B_KEY_DOWN** message. If the user continues to hold the key down, it will, after an initial delay, continue to produce messages at regularly spaced intervals—until the user releases the key or presses another key. The delay and the spacing between messages are both preferences the user can set with the Keyboard application.

set_key_repeat_rate() sets the number of messages repeating keys produce per second. For a standard PC keyboard, the *rate* can be as low as 2 and as high as 30; **get_key_repeat_rate()** writes the current setting into the integer that *rate* refers to.

set_key_repeat_delay() sets the length of the initial delay before the key begins repeating. Acceptable values are 250,000.0, 500,000.0, 750,000.0 and 1,000,000.0 microseconds (.25, .5, .75, and 1.0 second); **get_key_repeat_delay()** writes the current setting into the variable that *delay* points to.

All four functions return **B_NO_ERROR** if they successfully communicate with the Application Server, and **B_ERROR** if not. It's possible for the **set...()** functions to communicate with the Server but not succeed in setting the *rate* or *delay* (for example, if the *delay* isn't one of the listed four values).

set_keyboard_locks()

<interface/InterfaceDefs.h>

void **set_keyboard_locks(**ulong *modifiers***)**

Turns the keyboard locks—Caps Lock, Num Lock, and Scroll Lock—on and off. The keyboard locks that are listed in the *modifiers* mask passed as an argument are turned on; those not listed are turned off. The mask can be 0 (to turn off all locks) or it can contain any combination of the following constants:

B_CAPS_LOCK
B_NUM_LOCK
B_SCROLL_LOCK

See also: **system_key_map()**, **modifiers()**

set_menu_info(), get_menu_info()

<interface/Menu.h>

void **set_menu_info(**menu_info **info***)**

void **get_menu_info(**menu_info **info***)**

These functions set and get the user's preferences for how menus should look and work. User's express their preferences with the Menu application, which calls **set_menu_info()**. **get_menu_info()** writes the current preferences into the **menu_info** structure that into refers to. This structure contains the following fields:

float **font_size**	The size of the font that will be used to display menu items.
font_name **font**	The name of the font that's used to display menu items.
rgb_color **background_color**	The background color of the menu.
long **separator**	The style of horizontal line that separates groups of items in a menu. The value is an index ranging from 0 through 2; there are three possible separators.
bool **click_to_open**	Whether it's possible to open a menu by clicking in the item that controls it. The default value is **TRUE**.
bool **triggers_always_shown**	Whether trigger characters are always marked in menus and menu bars, regardless of whether the

menu hierarchy is the target for keyboard actions. The default value is **FALSE**.

< At present, both functions always return **B_NO_ERROR**. >

See also: the BMenu class

set_modifier_key()

<interface/InterfaceDefs.h>

void **set_modifier_key(**ulong *modifier*, ulong *key***)**

Maps a *modifier* role to a particular *key* on the keyboard, where *key* is a key identifier and *modifier* is one of the these constants:

B_CAPS_LOCK	**B_LEFT_SHIFT_KEY**	**B_RIGHT_SHIFT_KEY**
B_NUM_LOCK	**B_LEFT_CONTROL_KEY**	**B_RIGHT_CONTROL_KEY**
B_SCROLL_LOCK	**B_LEFT_OPTION_KEY**	**B_RIGHT_OPTION_KEY**
B_MENU_KEY	**B_LEFT_COMMAND_KEY**	**B_RIGHT_COMMAND_KEY**

The *key* in question serves as the named modifier key, unmapping any key that previously played that role. The change remains in effect until the default key map is restored. In general, the user's preferences for modifier keys—expressed in the Keymap application—should be respected.

Modifier keys can also be mapped by calling **system_key_map()** and altering the **key_map** structure directly. This function is merely a convenient alternative for accomplishing the same thing.

See also: **system_key_map()**

set_mouse_map(), get_mouse_map(), set_mouse_type(), get_mouse_type(), set_mouse_speed(), get_mouse_speed()

<interface/InterfaceDefs.h>

long **set_mouse_map(**mouse_map **map***)**

long **get_mouse_map(**mouse_map **map***)**

long **set_mouse_type(**long *numButtons***)**

long **get_mouse_type(**long **numButtons***)**

long **set_mouse_speed(**long *acceleration***)**

long **get_mouse_speed(**long **acceleration***)**

These functions configure the mouse and supply information about the current configuration. The configuration should usually be left to the user and the Mouse preferences application.

set_mouse_map() maps the buttons of the mouse to their roles in the user interface, and **get_mouse_map()** writes the current map into the variable referred to by *map*. The **mouse_map** structure has a field for each button on a three-button mouse:

ulong **left**	The button on the left of the mouse
ulong **right**	The button on the right of the mouse
ulong **middle**	The button in the middle, between the other two buttons

Each field is set to one of the following constants:

B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

The same role can be assigned to more than one physical button. If all three buttons are set to **B_PRIMARY_MOUSE_BUTTON**, they all function as the primary button; if two of them are set to **B_SECONDARY_MOUSE_BUTTON**, they both function as the secondary button; and so on.

set_mouse_type() informs the system of how many buttons the mouse actually has. If it has two buttons, only the **left** and **right** fields of the **mouse_map** are operative. If it has just one button, only the **left** field is operative. **set_mouse_type()** writes the current number of buttons into the variable referred to by *numButtons*.

set_mouse_speed() sets the speed of the mouse—the acceleration of the cursor image on-screen relative to the actual speed at which the user moves the mouse on its pad. An *acceleration* value of 0 means no acceleration. The maximum acceleration is 20, though even 10 is too fast for most users. **set_mouse_speed()** writes the current acceleration into the variable referred to by *acceleration*.

All six functions return **B_NO_ERROR** if successful, and an error code, typically **B_ERROR**, if not.

set_screen_refresh_rate() *see* get_screen_info()

set_screen_space() *see* get_screen_info()

set_scroll_bar_info(), get_scroll_bar_info()

long **set_scroll_bar_info(**scroll_bar_info **info***)**

long **get_scroll_bar_info(**scroll_bar_info **info***)**

These functions set and report preferences that the BScrollBar class uses when it creates a new scroll bar. **set_scroll_bar_info()** reads the values contained in the **scroll_bar_info** structure that *info* refers to and sets the system-wide preferences accordingly; **get_scroll_bar_info()** writes the current preferences into the structure provided.

The **scroll_bar_info** structure contains the following fields:

bool **proportional**	**TRUE** if scroll bars should have a knob that grows and shrinks to show what proportion of the document is currently visible on-screen, and **FALSE** if not. Scroll knobs are proportional by default.
bool **double_arrows**	**TRUE** if a set of double arrows (for scrolling in both directions) should appear at each end of the scroll bar, or **FALSE** if only single arrows (for scrolling in one direction only) should be used. Double arrows are the default.
long **knob**	An index that picks the pattern for the knob. Only values of 0, 1, and 2 are currently valid. The patterns can be seen in the ScrollBar preferences application. The pattern at index 1 is the default.
long **min_knob_size**	The length of the scroll knob, in pixels. This is the minimum size for a proportional knob and the fixed size for one that's not proportional. The default is 15.

The user can set these preferences with the ScrollBar application. Applications can call **get_scroll_bar_info()** to find out what choices the user made, but should refrain from calling **set_scroll_bar_info()**. That function is desigined for utilities, like the ScrollBar application, that enable users to set preferences that are respected system-wide.

If successful, these functions return **B_NO_ERROR**; if not, they return **B_ERROR**.

See also: the BScrollBar class

set_workspace_count(), count_workspaces()

<interface/InterfaceDefs.h>

void **set_workspace_count(**long *numWorkspaces***)**

long **count_workspaces(**void**)**

These functions set and return the number of workspaces the user has available. There can be as many as 32 workspaces and as few as 1. The choice of how many there should be is usually left to the user and the Workspaces application.

See also: **activate_workspace()**

system_colors()

<interface/InterfaceDefs.h>

color_map ***system_colors(**void**)**

Returns a pointer to the system's *color map*. The color map defines the set of 256 colors that can be displayed in the **B_COLOR_8_BIT** color space. A single set of colors is shared by all applications connected to the Application Server.

The **color_map** structure is defined in **interface/InterfaceDefs.h** and contains the following fields:

long **id**	An identifier that the Server uses to distinguish one color map from another.
rgb_color **color_list**[256]	A list of the 256 colors, expressed as **rgb_color** structures. Indices into the list can be used to specify colors in the **B_COLOR_8_BIT** color space. See the **index_for_color()** function above.
uchar **inversion_map**[256]	A mapping of each color in the **color_list** to its opposite color. Indices are mapped to indices. An example of how this map might be used is given below.
uchar **index_map**[32768]	An array that maps RGB colors—specified using five bits per component—to their nearest counterparts in the color list. An example of how to use this map is also given below.

The **inversion_map** is a list of indices into the **color_list** where each index locates the "inversion" of the original color. The inversion of the *n*'th color in **color_list** would be found as follows:

```
uchar inversionIndex = system_colors()->inversion_map[n];
rgb_color inversionColor =
                system_colors()->color_list[inversionIndex];
```

Inverting an inverted index returns the original index, so this code

```
uchar color = system_colors()->inversion_map[inversionIndex];
```

would return *n*. < Inverted colors are used, primarily, for highlighting. Given a color, its highlight complement is its inversion. >

The **index_map** maps every RGB combination that can be expressed in 15 bits (five bits per component) to a single **color_list** index that best approximates the original RGB data.

The following example demonstrates how to squeeze 24-bit RGB data into a 15-bit number that can be used as an index into the **index_map**:

```
long rgb15 = ( ((red & 0xf8) << 7) |
               ((green & 0xf8) << 2) |
               ((blue & 0xf8) >> 3) );
```

Most applications won't need to use the index map directly; the **index_for_color()** function performs the same conversion with less fuss (no masking and shifting required). However, applications that implement repetitive graphic operations, such as dithering, may want to access the index map themselves, and thus avoid the overhead of an additional function call.

You should never modify or free the **color_map** structure returned by this function.

See also: **index_for_color()**

system_key_map(), restore_key_map()

<interface/InterfaceDefs.h>

key_map ***system_key_map(**void**)**

void **restore_key_map(**void**)**

The first of these functions returns a pointer to the system key map—the structure that describes the role of each key on the keyboard. The second function restores the default map, in case any of its fields have been changed.

The system key map is shared by all applications. An application can alter values in the structure that **system_key_map()** returns—and thus alter the roles that the keys play—but it should make sure that those changes are local to itself and don't affect other, unsuspecting applications. In particular, it should:

- Modify the key map only when one of its windows becomes the active window, and
- Restore the default key map when it no longer has the active window.

Through the Keymap preferences application, users can configure the keyboard to their liking. The user's preferences affect all applications; they're captured in the default key map and stored in a file (**/system/settings/Key_map**).

When the machine reboots or when **restore_key_map()** is called, the key map is read from this file. If the file doesn't exist, the original map encoded in the Application Server is used.

The **key_map** structure contains a large number of fields, but it can be broken down into these six parts:

- A version number.
- A series of fields that determine which keys will function as modifier keys—such as Shift, Control, or Num Lock.
- A field that sets the initial state of the keyboard locks in the default key map.
- A series of ordered tables that assign character values to keys. Keys assigned a value other than –1 produce key-down events when pressed. This includes almost all the keys on the keyboard (all except for a handful of modifier keys).
- A series of tables that locate the dead keys for diacritical marks and determine how a combination of a dead key plus another key is mapped to a particular character.
- A set of masks that determine which modifier keys are required for a key to be considered dead.

The following sections describe each part of the **key_map** structure in turn.

Version. The first field of the key map is a version number:

ulong **version**	An internal identifier for the key map.

The version number doesn't change when the user configures the keyboard, and shouldn't be changed programmatically either. You can ignore it.

Modifiers. Modifier keys set states that affect other user actions on the keyboard and mouse. Eight modifier states are defined—Shift, Control, Option, Command, Menu, Caps Lock, Num Lock, and Scroll Lock. These states are discussed under "Modifier Keys" on page 51 of the introduction. They overlap, but don't exactly match the key caps found on a standard keyboard—which generally has a set of Alt(ernate) keys, rarely Option keys, and only sometimes Command and Menu keys. Because of these differences, the mapping of keys to modifiers is the area of the key map most open to the user's personal judgement and taste, and consequently to changes in the default configuration. Applications are urged to respect the user's preferences.

Since two keys, one on the left and one on the right, can be mapped to the Shift, Control, Option, and Command modifiers, the keyboard can have as many as twelve modifier keys. The **key_map** structure has one field for each key:

ulong **caps_key**	The key that functions as the Caps Lock key—by default, this is the key labeled "Caps Lock," key 0x3b.

ulong **scroll_key**	The key that functions as the Scroll Lock key—by default, this is the key labeled “Scroll Lock,” key 0x0f.
ulong **num_key**	The key that functions as the Num Lock key—by default, this is the key labeled “Num Lock,” key 0x22.
ulong **left_shift_key**	A key that functions as a Shift key—by default, this is the key on the left labeled “Shift,” key 0x4b.
ulong **right_shift_key**	Another key that functions as a Shift key—by default, this is the key on the right labeled “Shift,” key 0x56.
ulong **left_command_key**	A key that functions as a Command key—by default, this is the left “Alt” key, key 0x5d.
ulong **right_command_key**	Another key that functions as a Command key—by default, this is the right “Alt” key, key 0x5f.
ulong **left_control_key**	A key that functions as a Control key—by default, this is the key labeled “Control” on the left, key 0x5c.
ulong **right_control_key**	Another key that functions as a Control key—by default, this key is not mapped. (The value of the field is set to 0.)
ulong **left_option_key**	A key that functions as an Option key—by default, this is the key that’s labeled “Command” (or that has a command symbol) on the left of some keyboards, key 0x66. This key doesn’t exist on, and therefore isn’t mapped for, a standard 101-key keyboard.
ulong **right_option_key**	A key that functions as an Option key—by default, this is the key labeled “Control” on the right, key 0x60.
ulong **menu_key**	A key that initiates keyboard navigation of the menu hierarchy—by default, this is the key labeled “Menu,” key 0x68. This key doesn’t exist on, and therefore isn’t mapped for, a standard 101-key keyboard.

Each field names the key that functions as that modifier. For example, when the user holds down the key whose code is set in the **right_option_key** field, the **B_OPTION_KEY** and **B_RIGHT_OPTION_KEY** bits are turned on in the modifiers mask that the **modifiers()** function returns. When the user then strikes a character key, the **B_OPTION_KEY** state influences the character that’s generated.

If a modifier field is set to a value that doesn't correspond to an actual key on the keyboard (including 0), that field is not mapped. No key fills that particular modifier role.

Keyboard locks. One field of the key map sets initial modifier states:

ulong **lock_settings**	A mask that determines which keyboard locks are turned on when the machine reboots or when the default key map is restored.

The mask can be 0 or may contain any combination of these three constants:

B_CAPS_LOCK
B_SCROLL_LOCK
B_NUM_LOCK

It's 0 by default; there are no initial locks.

Altering the **lock_settings** field has no effect unless the altered key map is made the default (by writing it to a file that replaces **/system/settings/Key_map**).

Character maps. The principal job of the key map is to assign character values to keys. This is done in a series of nine tables:

ulong **control_map**[128]	The characters that are produced when a Control key is down but both Command keys are up.
ulong **option_caps_shift_map**[128]	The characters that are produced when Caps Lock is on and both a Shift key and an Option key are down.
ulong **option_caps_map**[128]	The characters that are produced when Caps Lock is on and an Option key is down.
ulong **option_shift_map**[128]	The characters that are produced when both a Shift key and an Option key are down.
ulong **option_map**[128]	The characters that are produced when an Option key is down.
ulong **caps_shift_map**[128]	The characters that are produced when Caps Lock is on and a Shift key is down.
ulong **caps_map**[128]	The characters that are produced when Caps Lock is on.
ulong **shift_map**[128]	The characters that are produced when a Shift key is down.

ulong **normal_map**[128]	The characters that are produced when none of the other tables apply.

Each of these tables is an array of 128 characters (declared as **ulong**s). Key codes are used as indices into the arrays. The value stored at any particular index is the character associated with that key. For example, the code assigned to the *M* key is 0x52; the characters to which the *M* key is mapped are recorded at index 0x52 in the various arrays.

The tables are ordered. Character values from the first applicable array are used, even if another array might also seem to apply. For example, if Caps Lock is on and a Control key is down (and both Command keys are up), the **control_map** array is used, not **caps_map**. If a Shift key is down and Caps Lock is on, the **caps_shift_map** is used, not **shift_map** or **caps_map**.

Notice that the last eight tables (all except **control_map**) are paired, with a table that names the Shift key (**..._shift_map**) preceding an equivalent table without Shift:

- **option_caps_shift_map** is paired with **option_caps_map**,
- **option_shift_map** with **option_map**,
- **caps_shift_map** with **caps_map**, and
- **shift_map** with **normal_map**.

These pairings are important for a special rule that applies to keys on the numerical keypad when Num Lock is on:

- If the Shift key is down, the non-Shift table is used.
- However, if the Shift key is *not* down, the Shift table is used.

In other words, Num Lock inverts the Shift and non-Shift tables for keys on the numerical keypad.

Not every key needs to be mapped to a character. If the value recorded in a table is –1, the key corresponding to that index is not mapped to a character given the particular modifier states the table represents. Generally, modifier keys are not mapped to characters, but all other keys are, at least for some tables. Key-down events are not generated for –1 character values.

Dead keys. Next are the tables that map combinations of keys to single characters. The first key in the combination is “dead”—it doesn’t produce a key-down event until the user strikes another character key. When the user hits the second key, one of two things will happen: If the second key is one that can be used in combination with the dead key, a single key-down event reports the combination character. If the second key doesn’t combine with the dead key, two key-down events occur, one reporting the dead-key character and one reporting the second character.

There are five dead-key tables:

ulong **acute_dead_key**[32]	The table for combining an acute accent (´) with other characters.
ulong **grave_dead_key**[32]	The table for combining a grave accent (`) with other characters.
ulong **circumflex_dead_key**[32]	The table for combining a circumflex (ˆ) with other characters.
ulong **dieresis_dead_key**[32]	The table for combining a dieresis (¨) with other characters.
ulong **tilde_dead_key**[32]	The table for combining a tilde (˜) with other characters

The tables are named after diacritical marks that can be placed on more than one character. However, the name is just a mnemonic; it means nothing. The contents of the table determine what the dead key is and how it combines with other characters. It would be possible, for example, to remap the **tilde_dead_key** table so that it had nothing to do with a tilde.

Each table consists of a series of up to 16 character pairs, where each character is declared as a **ulong**. The first character in the pair is the one that must be typed immediately after the dead key. The second character is the resulting character, the character that's produced by the combination of the dead key plus the first character in the pair. For example, if the first character is 'o', the second might be 'ô'—meaning that the combination of a dead key plus the character 'o' produces a circumflexed 'ô'.

The character pairs in the default **grave_dead_key** array look something like this:

```
' ', '´',
'A', 'À',
'E', 'È',
'I', 'Ì',
'O', 'Ò',
'U', 'Ù',
'a', 'à',
'e', 'è',
'i', 'ì',
'o', 'ò',
'u', 'ù',
. . .
```

By convention, the first pair in each array is a space followed by the dead-key character itself. This pair does double duty: It states that the dead key plus a space yields the dead-key character, and it also names the dead key. The system understands what the dead key is from the second character in the array.

Character tables for dead keys. As mentioned above, for a key to be dead, it must be mapped to the second character in a dead-key array. However, it's not typical for every key that's mapped to the character to be dead. Usually, there's a requirement that the user must hold down certain modifier keys (often the Option key). In other words, a key is dead only if selected character-map tables map it to the requisite character.

Five additional fields of the **key_map** structure specify what those character-map tables are—which modifiers are required for each of the dead keys:

ulong **acute_tables**	The character tables that cause a key to be dead when they map it to the second character in the **acute_dead_key** array.
ulong **grave_tables**	The character tables that cause a key to be dead when they map it to the second character in the **grave_dead_key** array.
ulong **circumflex_tables**	The character tables that cause a key to be dead when they map it to the second character in the **circumflex_dead_key** array.
ulong **dieresis_tables**	The character tables that cause a key to be dead when they map it to the second character in the **dieresis_dead_key** array.
ulong **tilde_tables**	The character tables that cause a key to be dead when they map it to the second character in the **tilde_dead_key** array.

Each of these fields contains a mask formed from the following constants:

B_CONTROL_TABLE
B_OPTION_CAPS_SHIFT_TABLE
B_OPTION_CAPS_TABLE
B_OPTION_SHIFT_TABLE
B_OPTION_TABLE
B_CAPS_SHIFT_TABLE
B_CAPS_TABLE
B_SHIFT_TABLE
B_NORMAL_TABLE

The mask designates the character-map tables that permit a key to be dead. For example, if the mask for the **grave_tables** field is,

```
B_OPTION_TABLE | B_OPTION_CAPS_SHIFT_TABLE
```

a key would be dead whenever either of those tables mapped the key to the second character in the **grave_dead_key** array ('`' in the example above). A key mapped to the same character by another table would not be dead.

See also: **BView::GetKeys()**, **modifiers()**, "Keyboard Information" in the chapter introduction, **set_modifier_key()**

Constants and Defined Types

This section lists the various constants and types that the Interface Kit defines to support the work done by its principal classes. The Kit is a framework of cooperating classes; almost all of its programming interface can be found in the class descriptions presented in previous sections of this chapter. Most of the constants and types listed here have already been explained in the descriptions of class member functions and global nonmember functions. Only one or two have not yet been mentioned in full detail. All of them are noted here and briefly described. If a more lengthy discussion is to be found under a class or a function, you'll be referred to that location.

Constants are listed first, followed by defined types. Constants that are defined as part of an enumeration type are presented with the other constants, rather than with the type. They're listed in the "Constants" section under the type name.

Constants

alert_type Constants

<interface/Alert.h>

Enumerated constant

B_EMPTY_ALERT
B_INFO_ALERT
B_IDEA_ALERT
B_WARNING_ALERT
B_STOP_ALERT

These constants designate the various types of alert panels that are recognized by the system. The type corresponds to an icon that's displayed in the alert window.

See also: the BAlert constructor

alignment Constants

<interface/InterfaceDefs.h>

Enumerated constant

B_ALIGN_LEFT
B_ALIGN_RIGHT
B_ALIGN_CENTER

These constants define the **alignment** data type. They determine how lines of text are aligned by BTextView and BStringView objects.

See also: **BTextView::SetAlignment()**

button_width Constants

<interface/Alert.h>

Enumerated constant

B_WIDTH_AS_USUAL
B_WIDTH_FROM_LABEL
B_WIDTH_FROM_WIDEST

These constants define the **button_width** type. They determine how the width of the buttons in an alert panel will be set—whether they're set to an standard (minimal) width, a width just sufficient to accommodate the button's own label, or a width sufficient to accommodate the widest label of all the buttons.

See also: the BAlert constructor

Character Constants

<interface/InterfaceDefs.h>

Enumerated constant	Character value
B_BACKSPACE	0x08 (same as '\b')
B_ENTER	0x0a (same as '\n')
B_RETURN	0x0a (synonym for **B_ENTER**)
B_SPACE	0x20 (same as ' ')
B_TAB	0x09 (same as '\t')
B_ESCAPE	0x1b
B_LEFT_ARROW	0x1c
B_RIGHT_ARROW	0x1d
B_UP_ARROW	0x1e
B_DOWN_ARROW	0x1f
B_INSERT	0x05
B_DELETE	0x7f

B_HOME	0x01
B_END	0x04
B_PAGE_UP	0x0b
B_PAGE_DOWN	0x0c
B_FUNCTION_KEY	0x10

These constants stand for the ASCII characters they name. Constants are defined only for characters that normally don't have visible symbols.

See also: "Function Key Constants" below

color_space Constants

<interface/InterfaceDefs.h>

Enumerated constant	Meaning
B_MONOCHROME_1_BIT	One bit per pixel, where 1 is black and 0 is white.
B_GRAYSCALE_8_BIT	256 gray values, where 255 is black and 0 is white.
B_COLOR_8_BIT	Colors specified as 8-bit indices into the color map.
B_RGB_16_BIT	< undefined for the current release >
B_RGB_32_BIT	Colors as 8-bit red, green, and blue components.

These constants define the **color_space** data type. A color space describes two properties of bitmap images:

- How many bits of information there are per pixel (the depth of the image), and
- How those bits are to be interpreted (whether as colors or on a grayscale, what the color components are, and so on).

See the "Colors" section in the chapter introduction for a fuller explanation of the color spaces currently defined for this type, particularly **B_RGB_32_BIT**.

See also: "Colors" on page 25, the BBitmap class

Control Values

<interface/Control.h>

Enumerated constant	Value
B_CONTROL_ON	1
B_CONTROL_OFF	0

These constants define the bipolar states of a typical control device.

See also: **BControl::SetValue()**

Cursor Transit Constants

<interface/View.h>

Enumerated constant	Meaning
B_ENTERED_VIEW	The cursor has just entered a view.
B_INSIDE_VIEW	The cursor has moved within the view.
B_EXITED_VIEW	The cursor has left the view

These constants describe the cursor's transit through a view. Each **MouseMoved()** notification includes one of these constants as an argument, to inform the BView whether the cursor has entered the view, moved while inside the view, or exited the view.

See also: **BView::MouseMoved()**

Dead-Key Mapping

<interface/InterfaceDefs.h>

Enumerated constants
B_CONTROL_TABLE
B_OPTION_CAPS_SHIFT_TABLE
B_OPTION_CAPS_TABLE
B_OPTION_SHIFT_TABLE
B_OPTION_TABLE
B_CAPS_SHIFT_TABLE
B_CAPS_TABLE
B_SHIFT_TABLE
B_NORMAL_TABLE

These constants determine which combinations of modifiers can cause a key to be the "dead" member of a two-key combination.

See also: **system_key_map()**

drawing_mode Constants

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
B_OP_COPY	**B_OP_ADD**
B_OP_OVER	**B_OP_SUBTRACT**
B_OP_ERASE	**B_OP_MIN**
B_OP_INVERT	**B_OP_MAX**
B_OP_BLEND	

These constants define the **drawing_mode** data type. The drawing mode is a BView graphics parameter that determines how the image being drawn interacts with the image already in place in the area where it's drawn. The various modes are explained under "Drawing Modes" in the chapter introduction.

See also: "Drawing Modes" on page 27, **BView::SetDrawingMode()**

Font Name Length

<interface/InterfaceDefs.h>

Defined constant	Value
B_FONT_NAME_LENGTH	64

This constant defines the maximum length of a font name. It's used in the definition of the **font_name** type.

See also: **font_name** under "Defined Types" below

Function Key Constants

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
B_F1_KEY	**B_F9_KEY**
B_F2_KEY	**B_F10_KEY**
B_F3_KEY	**B_F11_KEY**
B_F4_KEY	**B_F12_KEY**
B_F5_KEY	**B_PRINT_KEY** (the "Print Screen" key)
B_F6_KEY	**B_SCROLL_KEY** (the "Scroll Lock" key)
B_F7_KEY	**B_PAUSE_KEY**
B_F8_KEY	

These constants stand for the various keys that are mapped to the **B_FUNCTION_KEY** character. When the **B_FUNCTION_KEY** character is reported in a key-down event, the application can determine which key produced the character by testing the key code against these constants. (Control-*p* also produces the **B_FUNCTION_KEY** character.)

See also: "Character Mapping" on page 53 of the introduction to this chapter

Interface Messages

<app/AppDefs.h>

Enumerated constant	Enumerated constant
B_ZOOM	**B_KEY_DOWN**
B_MINIMIZE	**B_KEY_UP**
B_WINDOW_RESIZED	**B_MOUSE_DOWN**
B_WINDOW_MOVED	**B_MOUSE_UP**
B_WINDOW_ACTIVATED	**B_MOUSE_MOVED**
B_QUIT_REQUESTED	**B_VIEW_RESIZED**
B_SCREEN_CHANGED	**B_VIEW_MOVED**
B_WORKSPACE_ACTIVATED	**B_VALUE_CHANGED**
B_WORKSPACES_CHANGED	**B_PULSE**
B_SAVE_REQUESTED	**B_PANEL_CLOSED**

These constants identify interface messages—system messages that are delivered to BWindow objects. Each constant conveys an instruction to do something in particular (**B_ZOOM**) or names a type of event (**B_KEY_DOWN**).

See also: “Interface Messages” on page 41 in the introduction to this chapter

menu_bar_border Constants

<interface/MenuBar.h>

Enumerated constant	Meaning
B_BORDER_FRAME	Put a border around the entire frame rectangle.
B_BORDER_CONTENTS	Put a border around the group of items only.
B_BORDER_EACH_ITEM	Put a border around each item.

These constants can be passed as an argument to BMenuBar’s **SetBorder()** function.

See also: **BMenuBar::SetBorder()**

menu_layout Constants

<interface/Menu.h>

Enumerated constant	Meaning
B_ITEMS_IN_ROW	Menu items are arranged horizontally, in a row.
B_ITEMS_IN_COLUMN	Menu items are arranged vertically, in a column.
B_ITEMS_IN_MATRIX	Menu items are arranged in a custom fashion.

These constants define the **menu_layout** data type. They distinguish the ways that items can be arranged in a menu or menu bar—they can be laid out from end to end in a row like a typical menu bar, stacked from top to bottom in a column like a typical menu, or arranged in some custom fashion like a matrix.

See also: the BMenu and BMenuBar constructors

Modifier States

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
B_SHIFT_KEY	**B_OPTION_KEY**
B_LEFT_SHIFT_KEY	**B_LEFT_OPTION_KEY**
B_RIGHT_SHIFT_KEY	**B_RIGHT_OPTION_KEY**
B_CONTROL_KEY	**B_COMMAND_KEY**
B_LEFT_CONTROL_KEY	**B_LEFT_COMMAND_KEY**
B_RIGHT_CONTROL_KEY	**B_RIGHT_COMMAND_KEY**
B_CAPS_LOCK	**B_MENU_KEY**
B_SCROLL_LOCK	
B_NUM_LOCK	

These constants designate the Shift, Option, Control, Command, and Menu modifier keys and the lock states set by the Caps Lock, Scroll Lock, and Num Lock keys. They're typically used to form a mask that describes the current, or required, modifier states.

For each variety of modifier key, there are constants that distinguish between the keys that appear at the left and right of the keyboard, as well as one that lumps both together. For example, if the user is holding the left Control key down, both **B_CONTROL_KEY** and **B_LEFT_CONTROL_KEY** will be set in the mask.

See also: **modifiers()**, **BWindow::AddShortcut()**, the BMenu constructor

Mouse Buttons

<interface/View.h>

Enumerated constant
B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

These constants name the mouse buttons. Buttons are identified, not by their physical positions on the mouse, but by their roles in the user interface.

See also: **BView::GetMouse()**, **set_mouse_map()**

orientation Constants

<interface/InterfaceDefs.h>

<u>Enumerated constant</u>

B_HORIZONTAL
B_VERTICAL

These constants define the **orientation** data type that distinguishes between the vertical and horizontal orientation of graphic objects. It's currently used only to differentiate scroll bars.

See also: the BScrollBar and BScrollView classes

Pattern Constants

<interface/InterfaceDefs.h>

const pattern **B_SOLID_HIGH** = { 0xff, 0xff, 0xff, 0xff, 0xff,0xff, 0xff, 0xff }

const pattern **B_SOLID_LOW** = { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 }

const pattern **B_MIXED_COLORS**
= { 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55 }

These constants name the three standard patterns defined in the Interface Kit.

B_SOLID_HIGH is a pattern that consists of the high color only. It's the default pattern for all BView drawing functions that stroke lines and fill shapes.

B_SOLID_LOW is a pattern with only the low color. It's used mainly to erase images (to replace them with the background color).

B_MIXED_COLORS alternates pixels between the high and low colors in a checkerboard pattern. The result is a halftone midway between the two colors. This pattern can produce fine gradations of color, especially when the high and low colors are set to two colors that are already quite similar.

See also: "Patterns" on page 26 of the chapter introduction, the **pattern** defined type below

Resizing Modes

<interface/View.h>

Defined constants

B_FOLLOW_LEFT
B_FOLLOW_RIGHT
B_FOLLOW_LEFT_RIGHT
B_FOLLOW_H_CENTER

B_FOLLOW_TOP
B_FOLLOW_BOTTOM
B_FOLLOW_TOP_BOTTOM
B_FOLLOW_V_CENTER

B_FOLLOW_ALL
B_FOLLOW_NONE

These constants are used to set the behavior of a view when its parent is resized. They're explained under the BView constructor.

See also: the BView constructor, **BView::SetResizingMode()**

Screen Spaces

<interface/InterfaceDefs.h>

Enumerated constant	Enumerated constant
B_8_BIT_640x480	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_32_BIT_1600x1200**
B_16_BIT_640x480	**B_8_BIT_640x400**
B_16_BIT_800x600	
B_16_BIT_1024x768	
B_16_BIT_1152x900	
B_16_BIT_1280x1024	
B_16_BIT_1600x1200	

These constants are used to configure the screen—to set its depth and the size of the pixel grid it displays—as well as to report which configurations are possible. < 16-bit depths are not currently supported. >

See also: **set_screen_space()**, **get_screen_info()**

Scroll Bar Constants

<interface/ScrollBar.h>

Defined constant
B_H_SCROLL_BAR_HEIGHT
B_V_SCROLL_BAR_WIDTH

These constants record the recommended thickness of scroll bars. They should be used to help define the frame rectangles passed to the BScrollBar constructor.

See also: the BScrollBar class

Tracking Constants

<interface/View.h>

Enumerated constant	Meaning
B_TRACK_WHOLE_RECT	Drag the whole rectangle around.
B_TRACK_RECT_CORNER	Drag only the left bottom corner of the rectangle.

These constants determines how BView's **BeginRectTracking()** function permits the user to drag (or drag out) a rectangle.

See also: **BView::BeginRectTracking()**

Transparency Constants

<interface/InterfaceDefs.h>

const uchar **B_TRANSPARENT_8_BIT**
const rgb_color **B_TRANSPARENT_32_BIT**

These constants set transparent pixel values in a bitmap image. **B_TRANSPARENT_8_BIT** designates a transparent pixel in the **B_COLOR_8_BIT** color space, and **B_TRANSPARENT_32_BIT** designates a transparent pixel in the **B_RGB_32_BIT** color space.

Transparency is explained the "Drawing Modes" section of the chapter introduction. Drawing modes other than **B_OP_COPY** preserve the destination image where a source bitmap is transparent.

See also: "Drawing Modes" on page 27, the BBitmap class, **BView::SetViewColor()**

View Flags

<interface/View.h>

Enumerated constant	Meaning
B_FULL_UPDATE_ON_RESIZE	Include the entire view in the clipping region.
B_WILL_DRAW	Allow the BView to draw.
B_PULSE_NEEDED	Report pulse events to the BView.
B_FRAME_EVENTS	Report view-resized and view-moved events.

These constants can be combined to form a mask that sets the behavior of a BView object. They're explained in more detail under the class constructor. The mask is passed to the constructor, or to the **SetFlags()** function.

See also: the BView constructor, **BView::SetFlags()**

Window Areas

<interface/Window.h>

Enumerated constant
B_UNKNOWN_AREA
B_TITLE_AREA
B_CONTENT_AREA
B_RESIZE_AREA
B_CLOSE_AREA
B_ZOOM_AREA

These constants name the various parts of a window. They're used to designate the area where the cursor is located in messages that report the cursor's movement over a window.

See also: "**B_MOUSE_MOVED**" on page 10 in the *Message Protocols* appendix

Window Flags

<interface/Window.h>

Enumerated constant	Enumerated constant
B_NOT_MOVABLE	**B_NOT_CLOSABLE**
B_NOT_H_RESIZABLE	**B_NOT_ZOOMABLE**
B_NOT_V_RESIZABLE	**B_NOT_MINIMIZABLE**
B_NOT_RESIZABLE	**B_WILL_FLOAT**
B_WILL_ACCEPT_FIRST_CLICK	

These constants set the behavior of a window. They can be combined to form a mask that's passed to the BWindow constructor.

See also: the BWindow constructor

window_type Constants

<interface/Window.h>

Enumerated constant	Meaning
B_MODAL_WINDOW	The window is a modal window.
B_BORDERED_WINDOW	The window has a border but no title tab.
B_TITLED_WINDOW	The window has a border and a title tab.
B_DOCUMENT_WINDOW	The window has a tab and borders fit for scroll bars.

These constants describe the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor

Workspace Constants

<interface/Window.h>

Defined constant
B_CURRENT_WORKSPACE
B_ALL_WORKSPACES

These constants are used—along with designations of specific workspaces—to associate a set of one or more workspaces with a BWindow.

See also: the BWindow constructor, **BWindow::SetWorkspaces()**

Defined Types

alert_type

<interface/Alert.h>

typedef enum {. . .} **alert_type**

These constants name the various types of alert panel.

See also: "**alert_type** Constants" on page 335 above, the BAlert constructor

alignment

<interface/InterfaceDefs.h>

typedef enum {. . .} **alignment**

Alignment constants determine where lines of text are placed in a view.

See also: “**alignment** Constants” on page 336 above, **BTextView::SetAlignment()**

button_width

<interface/Alert.h>

typedef enum {. . .} **button_width**

These constants name the methods that can be used to determine how wide to make the buttons in an alert panel.

See also: “**button_width** Constants” on page 336 above, the BAlert constructor

color_map

<interface/InterfaceDefs.h>

```
typedef struct {
        long id;
        rgb_color color_list[256];
        uchar inversion_map[256];
        uchar index_map[32768];
} color_map
```

This structure contains information about the color context provided by the Application Server. There’s one and only one color map for all applications connected to the Server. Applications can obtain a pointer to the color map by calling the global **system_colors()** function. See that function for information on the various fields.

See also: **system_colors()**

color_space

<interface/InterfaceDefs.h>

typedef enum {. . .} **color_space**

Color space constants determine the depth and interpretation of bitmap images. They’re described under “Colors” in the introduction.

See also: “**color_space** Constants” on page 337 above, “Colors” on page 25, the BBitmap class

drawing_mode

<interface/InterfaceDefs.h>

typedef enum {. . .} **drawing_mode**

The drawing mode determines how source and destination images interact. The various modes are explained in the chapter introduction under "Drawing Modes".

See also: "Drawing Modes" on page 27, "**drawing_mode** Constants" on page 338 above

edge_info

<interface/View.h>

```
typedef struct {
        float left;
        float right;
} edge_info
```

This structure records information about the location of a character outline within the horizontal space allotted to the character. Edges separate one character from adjacent characters on the left and right. They're explained under the **GetCharEdges()** function in the BView class.

See also: **BView::GetCharEscapements()**, **BView::GetFontInfo()**

font_info

<interface/View.h>

```
typedef struct {
        font_name name;
        float size;
        float shear;
        float rotation;
        float ascent;
        float descent;
        float leading;
} font_info
```

This structure holds information about a BView's current font. Its fields are explained under the **GetFontInfo()** function in the BView class.

See also: **BView::GetFontInfo()**, **BView::SetFontName()**

font_name

<interface/InterfaceDefs.h>

typedef char **font_name**[FONT_NAME_LENGTH + 1]

This type defines a string long enough to hold the name of a font—64 characters plus the null terminator.

See also: **BView::SetFontName()**, **get_font_name()**

key_info

<interface/View.h>

```
typedef struct {
        ulong char_code;
        ulong key_code;
        ulong modifiers;
        uchar key_states[16];
} key_info
```

This structure is used by BView's **GetKeys()** function to return all known information about what the user is currently doing on the keyboard.

See also: **BView::GetKeys()**, "Keyboard Information" on page 47 in the introduction to this chapter

key_map

```
<interface/InterfaceDefs.h>

typedef struct {
        ulong version;
        ulong caps_key;
        ulong scroll_key;
        ulong num_key;
        ulong left_shift_key;
        ulong right_shift_key;
        ulong left_command_key;
        ulong right_command_key;
        ulong left_control_key;
        ulong right_control_key;
        ulong left_option_key;
        ulong right_option_key;
        ulong menu_key;
        ulong lock_settings;
        ulong control_map[128];
        ulong option_caps_shift_map[128];
        ulong option_caps_map[128];
        ulong option_shift_map[128];
        ulong option_map[128];
        ulong caps_shift_map[128];
        ulong caps_map[128];
        ulong shift_map[128];
        ulong normal_map[128];
        ulong acute_dead_key[32];
        ulong grave_dead_key[32];
        ulong circumflex_dead_key[32];
        ulong dieresis_dead_key[32];
        ulong tilde_dead_key[32];
        ulong acute_tables;
        ulong grave_tables;
        ulong circumflex_tables;
        ulong dieresis_tables;
        ulong tilde_tables;
} key_map
```

This structure maps the physical keys on the keyboard to their functions in the user interface. It holds the tables that assign characters to key codes, set up dead keys, and determine which keys function as modifiers. There's just one key map shared by all applications running on the same machine. It's returned by the **system_key_map()** function.

See also: **system_key_map()**

menu_bar_border

> <interface/MenuBar.h>
>
> typedef enum {. . .} **menu_bar_border**

This type enumerates the ways that a menu bar can be bordered.

See also: **BMenuBar::SetBorder()**, "**menu_bar_border** Constants" above

menu_info

> <interface/Menu.h>

```
typedef struct {
        float font_size;
        font_name font;
        rgb_color background_color;
        long separator;
        bool click_to_open;
        bool triggers_always_shown;
} menu_info
```

This structure records the user's menu preferences.

See also: **set_menu_info()** , the BMenu class

menu_layout

> <interface/Menu.h>
>
> typedef enum {. . .} **menu_layout**

This type distinguishes the various ways that items can arranged in a menu or menu bar.

See also: the BMenu class, "**menu_layout** Constants" above

mouse_map

> <interface/InterfaceDefs.h>

```
typedef struct {
        ulong left;
        ulong right;
        ulong middle;
} mouse_map
```

This structure maps mouse buttons to their roles as the **B_PRIMARY_MOUSE_BUTTON**, **B_SECONDARY_MOUSE_BUTTON**, or **B_TERTIARY_MOUSE_BUTTON**.

See also: **set_mouse_map()**

orientation

```
<interface/InterfaceDefs.h>
typedef enum {. . .} orientation
```

This type distinguishes between the **B_VERTICAL** and **B_HORIZONTAL** orientation of scroll bars.

See also: the BScrollBar and BScrollView classes

pattern

```
<interface/InterfaceDefs.h>
typedef struct {
        uchar data[8];
} pattern
```

A pattern is a arrangement of two colors—the high color and the low color—in an 8-pixel by 8-pixel square. Pixels are specified in rows, with one byte per row and one bit per pixel. Bits marked 1 designate the high color; those marked 0 designate the low color. An example and an illustration are given under “Patterns” on page 26 of the introduction to this chapter.

See also: “Pattern Constants” above, “Patterns” in the chapter introduction

print_file_header

```
<interface/PrintJob.h>
typedef struct {
        long version;
        long page_count;
        long _reserved_1_;
        long _reserved_2_;
        long _reserved_3_;
        long _reserved_4_;
        long _reserved_5_;
} print_file_header
```

This structure defines the header information for a print job. < Although declared publicly, it currently is used only internally by the BPrintJob class. >

rgb_color

<interface/InterfaceDefs.h>

```
typedef struct {
    uchar red;
    uchar green;
    uchar blue;
    uchar alpha;
} rgb_color
```

This type specifies a full 32-bit color. Each component can have a value ranging from a minimum of 0 to a maximum of 255.

< The **alpha** component, which is designed to specify the coverage of the color (how transparent or opaque it is), is currently ignored. However, an **rgb_color** can be made completely transparent by assigning it the special value, **B_TRANSPARENT_32_BIT**. >

See also: **BView::SetHighColor()**

screen_info

<interface/InterfaceDefs.h>

```
typedef struct {
    color_space mode;
    BRect frame;
    ulong spaces;
    float min_refresh_rate;
    float max_refresh_rate;
    float refresh_rate;
    uchar h_position;
    uchar v_position;
    uchar h_size;
    uchar v_size;
} screen_info
```

This structure holds information about a screen. Its fields are explained under the **get_screen_info()** global function.

See also: **get_screen_info()**

scroll_bar_info

<interface/InterfaceDefs.h>

```
typedef struct {
        bool proportional;
        bool double_arrows;
        long knob;
        long min_knob_size;
} scroll_bar_info
```

This structure captures the user's preferences for how scroll bars should behave and appear.

See also: **set_scroll_bar_info()**, the BScrollBar class

symbol_set_name

<interface/InterfaceDefs.h>

typedef font_name **symbol_set_name**

This type defines a string long enough to hold the name of a symbol set—64 characters plus the null terminator. The names of symbol sets are subject to the same length constraint as the names of fonts, which is why this type is a redefinition of **font_name**.

See also: **get_symbol_set_name()**

window_type

<interface/Window.h>

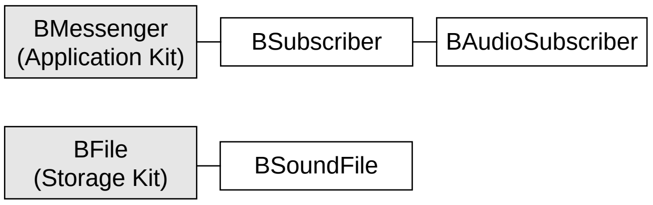
typedef enum {. . .} **window_type**

This type describes the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor, "**window_type** Constants" on page 346 above

5 The Media Kit

Media Kit Inheritance Hierarchy



5 The Media Kit

The Media Kit gives you tools that let you generate, examine, manipulate, and realize (or *render*) medium-specific data in real-time. It also lets you synchronize the transmission of data to different media devices, allowing you to build applications that can easily incorporate and coordinate audio and video (for example).

There are three layers in the Media Kit:

- Through the classes provided by the *module* layer, you create data-generating and -manipulating modules that can be plugged into each other to create an ever-narrowing data-processing tree. The tree terminates at a global scheduling object. Every application can have its own processing tree, or it can share branches or even individual modules with other applications. Synchronization between data from different media is handled by the scheduler: All you have to do is define and hook up the data-processing modules.

- At the *subscriber* layer are classes that let you talk directly to the media servers that are provided by the Kit. For each distinct medium there's a distinct server—but there's only one server per medium per computer. Corresponding to each server is a BSubscriber-derived class. Through instances of these classes you can receive and send data to the server.

- The *stream* layer lets you access the "data-streaming" facilities of the Kit. A data stream (as used by the Kit) is a sequence of programming entities that each get access to a set of data buffers. There are no servers or other media-specific constraints at this layer; you can actually use the classes in the stream layer to design a streamlined, intra-computer, data-transmission application (currently, streams can't broadcast over a network).

These three layers are interconnected: The module layer is built on top of the subscriber layer, which is built on top of the stream layer. Most high-level media applications will want to use the module layer exclusively. If you need more control or greater efficiency, head for the subscriber layer. The stream layer is the least useful to media applications, but, as mentioned above, it may find a home in applications—media-specific or not—that want to set up an efficient, real-time data pipeline.

Currently, only the subscriber and stream layers of the Media Kit are implemented, and, in this release, only the subscriber layer is documented.

At the subscriber layer, the Kit provides two classes:

- BSubscriber defines the basic rules to which all subscribers must adhere. If you want to use the subscriber layer, this is where you start to learn about it.

- BAudioSubscriber provides additional functionality that speaks directly to the *Audio Server.* The Audio Server is a background application that manages sound data that arrives through the microphone or line-in jacks, and that sends sound data to the internal speaker and line-out jacks. All subscribers that you create, for now, will be instances of BAudioSubscriber.

The Kit also provides a BSoundFile class that lets you read the data in a sound file, and global functions that let you play sound files.

BAudioSubscriber

Derived from:	public BSubscriber
Declared in:	<media/AudioSubscriber.h>

Overview

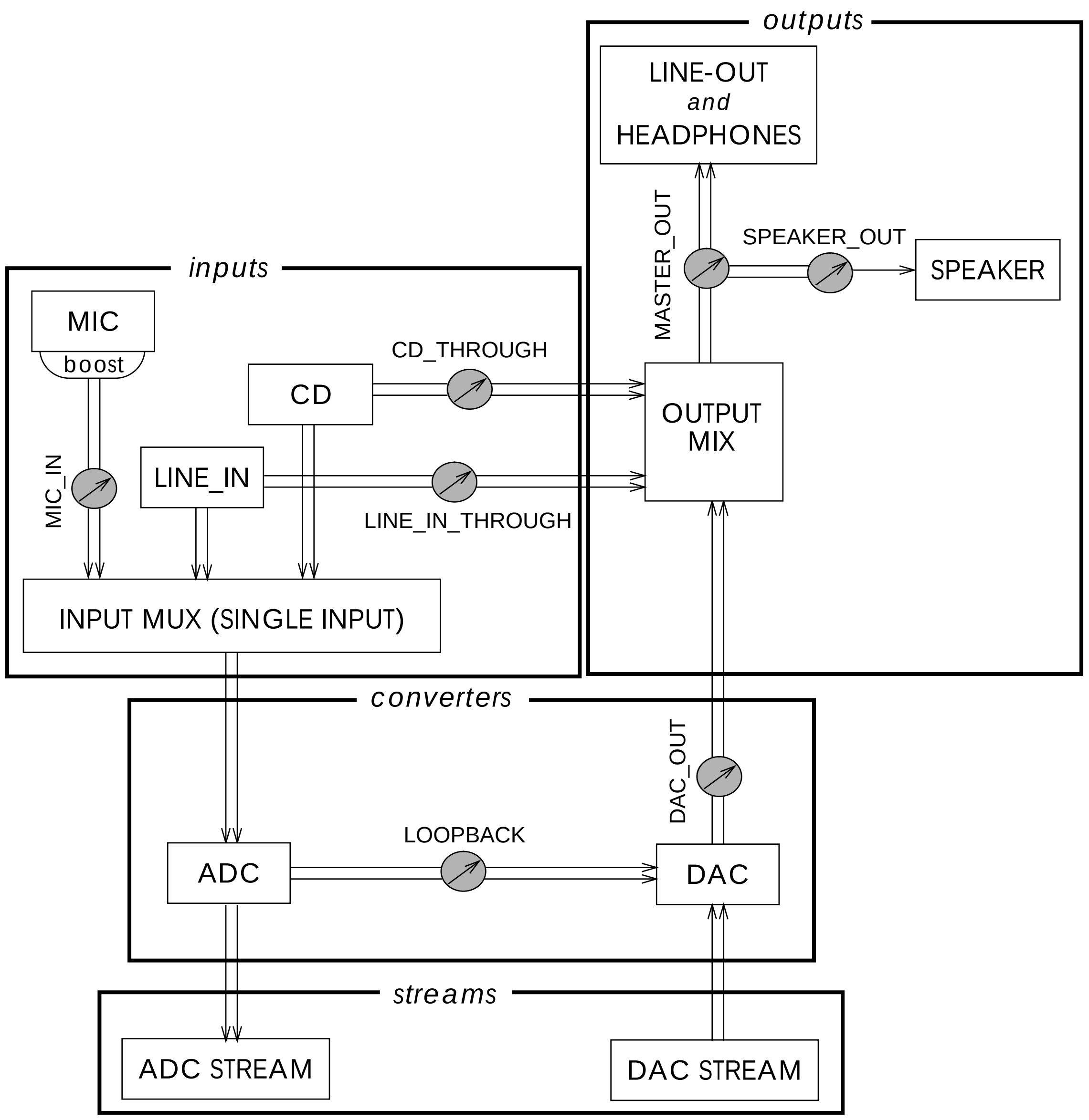
BAudioSubscriber objects perform two functions:

- They let your application receive, process, and broadcast sound data.
- They let you control certain parameters—such as volume and muting—of the sound hardware.

Ultimately, the first point is the more interesting of the two: Recording, generating, and manipulating sound data is a bit more amusing than simply setting the volume levels of the hardware devices. But to understand how and what data is received by your BAudioSubscriber objects, and what happens when you broadcast data through an object, you should first understand how the hardware is configured. The next section examines the sound hardware; following that is a description of the sound data that appears in your application.

Sound Hardware

The sound hardware consists of a number of physical devices (jacks, converters, and the like), a signal path that routes audio data between these devices, and “control points” along the signal path that let you adjust the format and flow of the audio data. These elements are depicted in the following illustration.



- The four large boxes ("inputs," "converters," "streams," and "outputs") divide the signal path into manageable territories; each territory is examined in separate sections, below.

- The smaller boxes ("MIC," "CD," and so on) are actual or virtual sound devices.

- The long arrowed lines show how the devices are connected. A single line indicates a single channel, a double line means stereo. The arrowhead at the end of each line indicates the direction of the signal.

- The circled arrows show where the software can exhibit gain control over a device. Each control point is labelled as it's known to the Media Kit. Every control point has a volume control and a mute.

Inputs

There are three analog audio input devices:

- *The microphone.* The microphone jack at the back of the computer accepts a stereo mini-phone (1/8") plug. The analog microphone signal has its own volume control and mute, and also allows a 20 dB boost. The microphone signal then feeds into the input MUX.

- *Line-in*. The stereo line-in jacks at the back of the computer bring a line-level analog signal into the computer. This signal can be routed directly to the audio output devices, and fed to the MUX. The direct-to-output, or "through," path has its own volume control and mute; this control point is called **B_LINE_IN_THROUGH** by the Kit.

- *CD input*. The CD (analog) input has the same features as line-in: The CD signal can be sent through to the output (**B_CD_THROUGH**), and it can be fed to the MUX.

Note that the microphone signal *doesn't* have a through path.

To bring an analog signal into your application (so you can record it, for example), the signal must pass through the input MUX:

- The MUX is a "mutually exclusive" device that lets you choose a single (analog) input from among the three sources listed above. In other words, you can bring in the microphone signal *or* the line-in signal *or* the CD signal, but you can't bring in any two or all of them at the same time. The MUX passes the input signal to its output without conversion to digital representation or other modification.

Converters

There are two sound data converters, the analog-to-digital converter (ADC) and the digital-to-analog converter (DAC):

- The ADC takes the analog signal that it reads from the MUX and converts it to digital representation. It does this by producing a series of *samples*, or instantaneous measurements of the signal's amplitude. The ADC control point is called **B_ADC_IN.**

- The DAC converts digital sound data into a continuous analog signal. The DAC control point is called **B_DAC_OUT**.

Acting as a sort of "short-circuit" between these two devices is the loopback:

- The loopback path takes the digital signal straight out of the ADC and sends it to the DAC. This path is intended, primarily, to simulate a "through" path for the microphone signal. There's little reason to send the line-in or CD signal down the loopback path since they have actual through paths built in.

Streams

The ADC stream and DAC stream are the centerpieces of the BAudioSubscriber class. By subscribing to the ADC stream you can receive the samples that are emitted by the ADC; and by subscribing to the DAC stream, you can send buffers of digital sound data to the DAC.

To enter the ADC stream you must create a BAudioSubscriber, subscribe to the stream (by passing **B_ADC_STREAM** as the first argument to **Subscribe()**), and then call **EnterStream()**. At that point, your object will begin receiving buffers of ADC-converted data from the Audio Server. The buffers show up as arguments to the object's stream function.

Similarly, the DAC stream universe is broached by subscribing to and entering the **B_DAC_STREAM**.

If you're unfamiliar with the concepts of subscription, entering a stream, and the stream function, take a break and read the BSubscriber specification.

Outputs

The output devices take analog signals and broadcast them to hardware that can turn the signals into sound.

- The output mixer mixes the signal from the DAC with the signals from the line-in and CD through paths. You can control the output of this mix at the **B_MASTER_OUT** control point.
- The mixed signal is presented at the stereo line-out jacks at the back of the computer. This is the same signal that's presented at the headphone jack.
- The stereo signal is mixed to mono (and attenuated by 6 dB) and sent to the abysmal internal speaker. The speaker has its own volume and mute control (**B_SPEAKER_OUT**).

Controlling the Hardware

The BAudioSubscriber class defines a number of functions that control the sound hardware and that query the state of the hardware. Note that you can call these functions without first subscribing to one or the other of the audio streams.

Volume and Mute

To set the volume level of a particular sound device, you use BAudioSubscriber's **SetVolume()** function. The function takes three arguments:

- A constant that represents the device you want to control.
- A float that sets the volume level of the left channel of the device.
- A float that does the same for the right channel.

The device constants are listed below; they correspond to the named control points shown in the hardware diagram:

- **B_CD_THROUGH**
- **B_LINE_IN_THROUGH**
- **B_ADC_IN**
- **B_LOOPBACK**
- **B_DAC_OUT**
- **B_MASTER_OUT**
- **B_SPEAKER_OUT**

All volume levels are floating-point numbers in the range [0.0, 1.0], where 0.0 is inaudible, and 1.0 is maximum volume. If you're setting a single-channel device (the speaker), the left channel level is used—the value you pass as the right channel level is ignored. If you want to set one channel of a stereo device but leave the other at its present level, pass the **B_NO_CHANGE** constant for the no-change channel.

In the example below, a BAudioSubscriber is used to set the volume of the CD-through signal:

```
BAudioSubscriber *setter = BAudioSubscriber("setter");

/* Set the right channel of the CD through signal
 * to half the maximum volume, and leave the left channel
 * alone.
 */
setter->SetVolume(B_CD_THROUGH, B_NO_CHANGE, 0.5);
```

To mute a device, you disable it; or, more precisely, you set it to be not enabled. This is done through the **EnableDevice()** function. As with **SetVolume()**, the function's first argument is the constant that represents the device you want to control. The second argument is a boolean that states whether you want to enable (**TRUE**) or disable (**FALSE**) the device. For example:

```
/* Mute the internal speaker. */
setter->EnableDevice(B_SPEAKER_OUT, FALSE);
```

The **GetVolume()** and **IsDeviceEnabled()** functions retrieve the current volume and enabled state of a given device. (As a convenience, **GetVolume()** returns volume *and* enabled status; see the function description for details.)

The MUX and the Mic

To select the analog device that will feed into the MUX, you use the **SetADCInput()** function (the signal into the MUX goes to the ADC, hence the name of the function). The input devices are represented by these constants:

- **B_MIC_IN**
- **B_CD_IN**
- **B_LINE_IN**

The **ADCInput()** function returns the current input device.

The microphone's 20 dB boost is toggled through the **BoostMic()** function. The state of the boost is retrieved by **IsMicBoosted()**.

Sound Data

Sounds are propagated by the continuous fluctuation of air pressure. This fluctuation is called a sound wave. The digital representation of a sound wave consists of a series of discrete measurements of the instantaneous pressure (or amplitude) of the wave at precise, (typically) equally-spaced points in time. Each measurement is called a *sample*. There are five attributes that characterize a digital sound sample:

- The size of a single sound sample (the Media Kit expresses this measurement in bytes-per-sample).

- The order of bytes in a multiple-byte sample.

- The number of samples in a "frame" of sound, where each sample in the frame is meant to be played at the same time. For example, a stereo sound would have two samples-per-frame. Samples-per-frame is commonly called the *channel count*.

- The number of frames that should be played in a second. This is commonly called the *sampling rate*.

- The mapping from the value of a digital sample to a specific sound wave amplitude. The Media Kit calls this the *sample format*. Usually, the mapping is linear: When you double the value of a sample, you double the amplitude to which it corresponds.

The Be sound hardware (both the ADC and the DAC) allows the following sound attribute settings:

- Sample size can be one or two bytes-per-sample.

- Byte-ordering is either most-significant-byte first (**B_BIG_ENDIAN**), or least-significant-byte first (**B_LITTLE_ENDIAN**).

- The channel count can be one (mono) or two (stereo).

- The sampling rates, expressed as frames-per-second, that are supported by the hardware are: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.
- There are two sample formats: The linear format, represented by the constant **B_LINEAR_SAMPLES**, can be used with either one- or two-byte samples. The "mu-law" format (**B_MULAW_SAMPLES**) can only be used with one-byte samples. Mu-law is a quasi-exponential mapping that attempts to minimize quantization noise by dedicating more bits, proportionally, to low amplitude values than to high amplitude values.

The ADC and DAC use the same sampling rate. You can set the sampling rate through BAudioSubscriber's **SetSamplingRate()** function, but you can't specify which device you intend the setting to apply to: It always applies to both.

As for the other sound data parameters (sample size, byte order, channel count, and sample format), the ADC and the DAC maintain independent settings. For example, you can set the DAC to expect two-byte linear samples while the ADC produces one-byte mu-law samples. The functions that set these sound format attributes are **SetDACSampleInfo()** and **SetADCSampleInfo()**. Your BAudioSubscriber needn't subscribe before setting the DAC or ADC parameters.

Receiving and Broadcasting Sound Data

A BAudioSubscriber object receives buffers of sound data from one of the Audio Server's two buffer streams:

- The buffers that flow through the ADC stream are filled with sound data that's been brought into the computer, passed through the MUX, and converted by the ADC. Data buffers that are received by your objects will already be filled with this data. Although it's not forbidden, you usually don't modify the data in the ADC stream's buffers. BAudioSubscribers that enter the ADC stream do so, typically, to record or examine the data that they find there.
- The buffers that flow through the DAC stream are ultimately dumped into the DAC. The DAC stream's buffers are zeroed at the start of their journey; if a BAudioSubscriber wants to broadcast a sound, it enters the DAC stream and adds its sound data into the buffers as they flow past.

The ADC stream isn't automatically connected to the DAC stream. If you want to grab data from the ADC and send it to the DAC, you have to subscribe to both streams through two separate BAudioSubscriber objects, and then coordinate the hand off of data from the ADC subscriber to the DAC subscriber.

Constructor and Destructor

BAudioSubscriber()

BAudioSubscriber(const char **name***)**

Creates and returns a new BAudioSubscriber object. The object is given the name that you pass as *name*; the name is provided as a convenience and needn't be unique.

After creating a BAudioSubscriber, you typically do the following (in this order):

- Subscribe the object to one of the Audio Server's streams (either **B_ADC_STREAM** or **B_DAC_STREAM**) by calling **Subscribe()**.
- Allow the object to begin receiving buffers by calling **EnterStream()**.

See also: **BSubscriber::Subscribe()**, **BSubscriber::EnterStream()**

~BAudioSubscriber()

virtual **~BAudioSubscriber(**void**)**

Destroys the BAudioSubscriber.

Member Functions

ADCInput(), SetADCInput

long **ADCInput(**void**)**
long **SetADCInput(**long *device***)**

These functions get and set the device that feeds into the MUX (and so to the ADC, hence the name). Valid *device* constants are:

- **B_MIC_IN**
- **B_CD_IN**
- **B_LINE_IN**

You don't need to be subscribed to the ADC stream in order to call these functions.

BoostMic(), IsMicBoosted()

long **BoostMic(**bool *boost***)**
bool **IsMicBoosted(**void**)**

BoostMic() enables or disables the 20 dB boost on the microphone signal. **IsMicBoosted()** returns the state of the boost.

GetADCSampleInfo(), GetDACSampleInfo(), SamplingRate()

long **GetADCSampleInfo(**long **bytesPerSample,*
long **channelCount,*
long **byteOrder,*
long **sampleFormat***)**

long **GetDACSampleInfo(**long **bytesPerSample,*
long **channelCount,*
long **byteOrder,*
long **sampleFormat***)**

long **SamplingRate(**void**)**

These functions return the values of the various sound data parameters. **GetADC...** returns (by reference) the sound parameters that are used in the ADC stream. **GetDAC...** does the same for the DAC stream. **SamplingRate()** returns the sampling rate directly; the sampling rate is held in common by the two streams.

See the description of **SetADCSampleInfo()** for a list of parameter values that you can expect to see.

See also: **SetADCSampleInfo()**

GetDACSampleInfo() *see* GetADCSampleInfo()

SetADCSampleInfo(), SetDACSampleInfo(), SetSamplingRate()

long **SetADCSampleInfo(**long *bytesPerSample,*
long *channelCount,*
long *byteOrder,*
long *sampleFormat***)**

long **SetDACSampleInfo(**long *bytesPerSample,*
long *channelCount,*
long *byteOrder,*
long *sampleFormat***)**

long **SetSamplingRate(**long *samplingRate***)**

These functions set the values of the sound data attributes used by (respectively) the ADC stream (**SetADC...**), DAC stream (**SetDAC...**), and both streams (**SetSamplingRate()**). The arguments to the **SetADC...** and **SetDAC...** functions are:

- *bytesPerSample* is the size of a single sound sample measured in bytes. Acceptable values are 1 and 2.
- *channelCount* is the number of samples in a “frame” of sound. Acceptable values are 1(mono) and 2 (stereo).

- *byteOrder* is the order of bytes in a multiple-byte sample. The ordering is either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**.
- *sampleFormat* is the data format of a single sample. Linear format (**B_LINEAR_SAMPLES**) can be used for one- or two-byte samples; mu-law format (**B_MULAW_SAMPLES**) can be used for 1-byte samples.

The **SetSamplingRate()** function sets the sampling rate for both the ADC stream and the DAC stream:

- The following sampling rates are supported by the sound hardware: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.

These functions don't flinch at wildly inappropriate parameter settings. The values of the arguments that you pass in are always rounded to the nearest acceptable value for the particular parameter.

See also: **GetADCSampleInfo()**

SetDACSampleInfo() *see* SetADCSampleInfo()

SetVolume(), GetVolume(), EnableDevice(), IsDeviceEnabled()

long **SetVolume(**long *device,*
float *leftVolume,*
float *rightVolume***)**
long **GetVolume(**long *device,*
float **leftVolume,*
float **rightVolume***,**
bool *isEnabled***)**

long **EnableDevice(**long *device*, bool *enable***)**
bool **IsDeviceEnabled(**long *device***)**

These functions set and return (by reference) the left and right volume levels, and the enabled status of the device that's identified by the first argument. Valid device constants are:

- **B_ADC_IN**
- **B_CD_THROUGH**
- **B_LINE_IN_THROUGH**
- **B_LOOPBACK**
- **B_DAC_OUT**
- **B_MASTER_OUT**
- **B_SPEAKER_OUT**

Volume values are floating-point numbers that are clipped within the range [0.0, 1.0]. Across this range, the amplitude of a sound waveform is increased logarithmically; this results, perceptually, in a linear increase in volume.

Note that the speaker is monophonic; when you set or retrieve the volume of the **B_SPEAKER_OUT** device, only the *leftVolume* argument is used.

You needn't be subscribed to call these functions.

BSoundFile

Derived from:	public BFile
Declared in:	<media/SoundFile.h>

Overview

BSoundFile objects give you access to files that contain sound data. The BSoundFile functions let you examine the format of the data in the sound file, read the data, and position a "frame pointer" in the file. Notably absent from the list of a BSoundFile's talents is the ability to play itself and to record into itself. You *can* play a BSoundFile's data, but this requires the assistance of a BAudioSubscriber (as explained later). Currently, you can't record into a BSoundFile.

To use a BSoundFile, you set its ref (using the methods that are described in the BFile class), and then you open the file through a call to **Open()**. Since you can't record into a BSoundFile, you almost always open such files in **B_READ_ONLY** mode. None of the BSoundFile-defined functions work on an unopened file.

Sound File Formats

The BSoundFile class understands AIFF, WAVE, and "standard" UNIX sound files (**.snd** and **.au**). When you tell a BSoundFile object to open its file, the object figures out the format of the file—you can't force it to assume a particular format. If it encounters a file that's in a format that it doesn't understand ("unknown" format), it assumes that the file contains 44.1 kHz, 16-bit stereo data, and that the file doesn't have a header (it assumes that the entire file is sound data). The admission of the unknown format means that *any* file can act as sound data. BSoundFile doesn't know the meaning of "inappropriate data."

The file formats are represented by the constants **B_AIFF_FILE**, **B_WAVE_FILE**, **B_UNIX_FILE**, and **B_UNKNOWN_FILE**. You can retrieve the file format from an open BSoundFile through the **FileFormat()** function.

Note: 8-bit WAVE data is, by definition, unsigned. However, when you read such data (through the **ReadFrames()** function, which will be discussed later), it's automatically shifted so that it *is* signed. This automatic conversion allows an 8-bit WAVE file to be mixed with other sound sources.

Sound Data Parameters

After opening your BSoundFile, you can ask for the parameters of its data by calling the various parameter-retrieving functions (**SamplingRate()**, **ChannelCount()**, **SampleSize()**, and so on). There's also a set of parameter-setting functions (**SetSamplingRate()**, **SetChannelCount()**, **SetSampleSize()**, ...), but note that these functions don't actually modify the data in the file (or in the BSoundFile object); they simply set the object's impression of the sort of data that it contains so other objects that act on your BSoundFile will interpret the data correctly. This should only be necessary if the file format is unknown.

For example, let's say you have your own sound file format. Your format defines a header that lists the usual information—the size of the samples, where the data starts, and so on. BSoundFile won't recognize your format, of course, but through a combination of the **Read()** function (so you can read the header yourself) and the sample parameter-setting functions defined by BSoundFile, you can tell the object what sort of data it contains.

If you're creating your own BSoundFile-derived class to encapsulate your own sound file format, you would put the header-reading code in your implementation of the **Open()** function. For example:

```
long MySoundFile::Open(long mode)
{
    long result;

    if ((result = BSoundFile::Open(mode)) < B_NO_ERROR)
        return result;

    /* ReadHeader() is assumed to be implemented
     * by MySoundFile--it isn't a BSoundFile function.
     */
    if (FileFormat() == B_UNKNOWN_FILE)
        result = ReadHeader());
    return result;
}
```

By invoking the BSoundFile version of **Open()**, you allow your object to represent the standard file formats in addition to your own. (Keep in mind that BSoundFile sets the file format in its **Open()** implementation.)

Playing a Sound File

There are two methods for playing a sound file:

- The easy way is to call the **play_sound()** function. The function takes a **record_ref** argument (in addition to others), and plays the data that it finds in the referred to file—you don't need to create a BSoundFile object in order to call **play_sound()**. (The complete documentation for **play_sound()** and related functions can be found in the final section of this chapter.)

- A much more amusing approach is to create a BSoundFile object, open it, read the data contained within, and add the data into the DAC stream. Obviously, this is a bit more involved than the simple **play_sound()** (which a dead dog would have no trouble using), but it gives you more control over the sound: Since you're reading the sound data yourself, the BSoundFile approach lets you manipulate the sound as you're throwing it into the stream.

A demonstration of the second approach is given below. To understand the example, you must be familiar with the subscription and stream-entering mechanisms described in the BSubscriber class.

An Example

In this example, we show how to read data from a BSoundFile and add it to the DAC stream for playback. In addition, we'll allow dynamic amplitude control of the sound. For the sake of brevity, we'll restrict the example to 16-bit data.

First, we define an object called SoundPlayer that will be used to coordinate the Media Kit objects. Notice that SoundPlayer needn't derive from a Kit class:

```
class SoundPlayer : public BObject
{
    public:
        long SetSoundFile(record_ref ref);
        void Play(void);
        void SetAmpScale(double value);

    private:
        static bool _play_back(void *arg, char *sound,
            long size);
        bool Playback(short *sound, long sample_count);

        BAudioSubscriber *a_sub;
        BSoundFile s_file;
        char transfer_buf[B_PAGE_SIZE];
        double amp_scale;
};
```

There are three public functions: **SetSoundFile()** let's you set the soundfile that you want to play, **Play()** plays it, and **SetAmpScale()** will control the amplitude. In this implementation, the file is always allowed to play to completion—aborting the playback is left as an exercise for the reader.

The private **_play_back()** function will be the BAudioSubscriber's literal stream function. **Playback()** will be called from within **_play_back()**; it will do the actual stream work. The private **transfer_buf** will be used to transfer data between the file and the audio stream (a page at a time), and **amp_scale** will hold the amplitude scaling value.

Opening the File and Subscribing

In the implementation of **SetSoundFile()**, we set the BSoundFile's ref and open the file...

```
long SoundPlayer::SetSoundFile(record_ref ref)
{
    /* Set the BSoundFile's ref and open the object. */
    s_file.SetRef(ref);
    s_file.Open(B_READ_ONLY);
    if (s_file.Error() < B_NO_ERROR)
        return B_ERROR;

    /* Check for 16-bit data (given in bytes). */
    if (s_file.SampleSize() != 2)
        return B_ERROR;
```

...and then we create the BAudioSubscriber and subscribe it to the DAC stream and set the stream's sample parameters to match the data that's in the file:

```
    a_sub = new BAudioSubscriber("SoundFile Player");
    if(!a_sub->Subscribe(B_DAC_STREAM, B_SHARED_SUBSCRIBER_ID,
        FALSE) < B_NO_ERROR)
        return B_ERROR;

    a_sub->SetSamplingRate(s_file.SamplingRate());
    a_sub->SetDACSampleInfo(s_file.SampleSize(),
                           s_file.CountChannels(),
                           s_file.ByteOrder(),
                           s_file.SampleFormat());
```

Next, we set the size of the stream's buffers to match that of our transfer buffer. The arguments to **SetStreamBuffers()** are buffer size, buffer count. The buffer count we use here (8, the same as the Audio Server default) is unimportant in this example:

```
    a_sub->SetStreamBuffers(B_PAGE_SIZE, 8);
```

Finally, we initialize the amp scaler and return:

```
    amp_scale = 1.0;
    return B_NO_ERROR;
}
```

By setting the DAC stream's sample info and buffer size as shown in here, we make the stream function's job quite a bit easier—it won't have to convert the samples as it reads them from the file, or keep track of how many samples it has read. However, you should be aware that some other BAudioSubscriber could come along and reset the DAC stream at any time, thus screwing up the playback. For now, we'll live with the danger.

Entering the Stream

The **Play()** function enters the BAudioSubscriber into the DAC stream. This causes buffers to be sent to the stream function, which we'll implement in the next section.

```
void SoundPlayer::Play(void)
{
    a_sub->EnterStream(NULL, /* no neighbor */
                      TRUE, /* head of the stream */
                      this, /* arg for the stream function */
                      _play_back, /* the stream function */
                      NULL, /* no completion function */
                      TRUE); /* run in the background */
}
```

While we're at it, we'll implement the **SetAmpScale()** function:

```
void SoundPlayer::SetAmpScale(double scale)
{
    amp_scale = min(1.0, max(0.0, scale));
}
```

Reading and Playing the File

Now comes the fun part. First we implement the literal stream function, **_play_back()**:

```
bool SoundPlayer::_play_back(void *arg, char *sound, long size)
{
    return (((SoundPlayer *)arg)->Playback((short *)sound,
                                          size/2));
}
```

As **_play_back()** receives buffers from the DAC stream, it forwards them (cast as 16-bit data) to the guts of the operation, **Playback()**. At each invocation, **Playback()** reads the correct number of frames from the sound file, scales their amplitudes, and adds the samples into the DAC stream buffer. First, we set up some variables:

```
bool SoundPlayer::Playback(short *sound, long sample_count)
{
    long frames_read, counter;
    long channel_count = s_file.CountChannels();
    long frame_count = sample_count / channel_count;
    short *tb_ptr = (short *)transfer_buf;
```

Now we read **frame_count** sample frames from the file and place them in the transfer buffer. (We should check to make sure that the transfer buffer can accommodate the number of frames read—but, for this example, we'll assume that the stream's buffer size hasn't changed since we set it to be the same size as the transfer buffer.) If **ReadFrames()** returns less than the number of frames that we asked for, we're at the end of the file. **ReadFrames()** returns the number of frames that it actually read, or an error code (as usual, a negative number) if something went wrong:

```
    frames_read = s_file.ReadFrames(transfer_buf, frame_count);

    if (frames_read <= 0)
        return FALSE;
```

Finally, we get to write into the sound buffer. We loop over the samples in the transfer buffer, scale each by the **amp_scale** value, and then write the scaled value into the sound buffer:

```
        for (counter = 0; counter < frames_read; counter++) {
            *sound++ += *tb_ptr++ * amp_scale; /* left or mono */
            if (channel_count == 2)
                *sound++ += *tb_ptr++ * amp_scale; /* right */
        }
```

Once again we examine the **frames_read** count. If it's less than what we expected to have read, we've reached the end of the file, and so return **FALSE**. Otherwise we return **TRUE**:

```
        if (frames_read < frame_count)
            return FALSE;
        else
            return TRUE;
    }
```

Obviously, this example is neither robust nor efficient. In particular, the file-reading mechanism should probably read more than one page at a time—if you were to play more than a couple files simultaneously with this code, the constant file seeking could cause your hard disk to burn a hole right through to Australia. Or to California, if you live in Perth. The point of this exercise was to demonstrate the basic procedures of playing a sound file.

Constructor and Destructor

BSoundFile()

BSoundFile(void**)**
BSoundFile(record_ref *ref***)**

Creates and returns a new BSoundFile object. The first version of the constructor must be followed by a call to **SetRef()**.

~BSoundFile()

virtual **~BSoundFile(**void**)**

Closes the BSoundFile's sound file and destroys the object. The data in the sound file isn't affected.

Member Functions

CountFrames()

long **CountFrames(**void**)**

Returns the number of frames of sound that are in the object's file. If the object's file isn't open, this returns **B_ERROR**.

FileFormat()

long **FileFormat(**void**)**

Returns a constant that identifies the type of sound file that this object is associated with. Currently, three types of sound files are recognized: **B_AIFF_FILE, B_WAVE_FILE, B_UNIX_FILE** and **B_UNKNOWN_FILE**. AIFF is the Apple-defined sound format, WAVE is a popular PC format, the **B_UNIX_FILE** constant represents the sound file format that's used on many UNIX-based computers. **B_UNKNOWN_FILE** is returned for all other formats.

B_UNKNOWN_FILE isn't as useless as it sounds: Any file that is so identified is considered to contain "raw" sound data. You can accept the default values of the data format parameters (see **SamplingRate()** for a list of these values), or you can shape the data into a recognizable format by setting the data format parameters directly, through calls to **SetSamplingRate()**, **SetChannelCount()**, and so on. In this case, you'll need to position the frame pointer to the first frame—in other words, you have to read past the file's header, if any—yourself. Thus primed, subsequent calls to **ReadFrames()** will read the proper sequences of samples.

If the BSoundFile isn't open, this returns **B_ERROR**.

FrameIndex() *see* SeekToFrame()

FramesRemaining()

long **FramesRemaining(**void**)**

Returns the number of unread frames in the file, or **B_ERROR** if the object isn't open.

ReadFrames()

virtual long **ReadFrames(**char **buffer***,** long *frameCount***)**

Reads (as many as) *frameCount* frames of data into *buffer.* The function returns the number of frames that were actually read and increments the frame pointer by that amount. When you hit the end of the file, the function returns 0.

Note that *buffer* shouldn't be the sound buffer that's passed to you in a stream function. If you read directly into a stream function's sound buffer, you'll be clobbering the data that's already there. If you're calling **ReadFrames()** from within a stream function, you must first read into a "transfer buffer", and then add the contents of this buffer into the sound buffer.

If the BSoundFile object isn't open, this returns **B_ERROR**.

SamplingRate(), CountChannels(), SampleSize(), FrameSize(), ByteOrder(), SampleFormat()

long **SamplingRate(**void**)**
long **CountChannels(**void**)**
long **SampleSize(**void**)**
long **FrameSize(**void**)**
long **ByteOrder(**void**)**
long **SampleFormat(**void**)**

These functions return information about the format of the data that's found in the object's sound file:

- **SamplingRate()** returns the sampling rate.
- **CountChannels()** returns the number of channels of sound.
- **SampleSize()** returns the size, in bytes, of a single sample.
- **FrameSize()** is a convenience function that give the number of bytes in a single frame of sound (it's the same as **CountChannels()** * **SampleSize()**).
- **ByteOrder()** returns a constant that represents the order of samples within a frame. It's either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**.
- **SampleFormat()** returns a constant that represents the data format of a single sample. It's one of: **B_LINEAR_SAMPLES**, **B_MULAW_SAMPLES**, **B_FLOAT_SAMPLES**, or **B_UNDEFINED_SAMPLES**.

These functions returns default values if the object isn't associated with a file. The defaults are:

- 44100 frames per second
- 2 channels
- 2 bytes per sample (16-bit samples)
- 4 bytes per frame
- Bytes are ordered MSB first (**B_BIG_ENDIAN**)
- The sample format is **B_LINEAR_SAMPLES**

If the BSoundFile object isn't open, these functions return **B_ERROR**.

SeekToFrame(), FrameIndex()

virtual long **SeekToFrame(**ulong *index***)**
long **FrameIndex(**void**)**

Theses function set and return the location of the "frame pointer." The frame pointer points to the next frame that will be read from the file. The first frame in a file is frame 0.

If you try to set the frame pointer to a location that's outside the bounds of the data, the pointer is set to the frame at the nearest extreme.

If the BSoundFile object isn't open, this returns **B_ERROR**.

SetSamplingRate(),SetChannelCount(), SetSampleSize() SetByteOrder(), SetSampleFormat()

virtual long **SetSamplingRate(**long *samplingRate***)**
virtual long **SetChannelCount(**long *channelCount***)**
virtual long **SetSampleSize(**long *bytesPerSample***)**
virtual long **SetByteOrder(**long *byteOrder***)**
virtual long **SetSampleFormat(**long *sampleFormat***)**

If the file format of your BSoundFile is **B_UNKNOWN_FILE**, you can use these functions to tell the object how to interpret the format of its data. These functions don't change the actual data—neither as it's represented within the object, nor as it resides in the file—they simply prime the object for subsequent reads of the data.

The candidate values for the functions are:

- *samplingRate* can be any number, but will be rounded to the nearest hardware-supported sampling rate when the data is played. The sampling rates that the hardware supports are: 5510, 6620, 8000, 9600, 11025, 16000, 18900, 22050, 27420, 32000, 33075, 37800, 44100, 48000.
- *channelCount* is usually 1 (mono) or 2 (stereo). You can set the data to a higher count but the hardware can play no more than 2 channels at a time.
- *sampleSize* is usually 2 (16 bit samples). But it can also be 1 (the usual setting for mu-law encoding) or 4 (floating-point data).
- *byteOrder* is either **B_BIG_ENDIAN** or **B_LITTLE_ENDIAN**
- *sampleFormat* is one of **B_LINEAR_SAMPLES**, **B_MULAW_SAMPLES**, **B_FLOAT_SAMPLES**, or **B_UNDEFINED_SAMPLES.**

Each function returns the value that was actually implanted. If the BSoundFile object isn't open, they return **B_ERROR**.

BSubscriber

Derived from:	public BObject
Declared in:	<media/Subscriber.h>

Overview

BSubscriber objects receive and process buffers of media-specific data. These buffers are allocated and sent (to the BSubscriber) by a media server; for example, buffers of audio data are sent by the *Audio Server.* Each server can control more than one *buffer stream* (the Audio Server has a sound-in stream and a sound-out stream). A BSubscriber can receive buffers from only one stream.

More than one BSubscriber can "subscribe" to the same stream. The collection of same-stream BSubscribers stand shoulder-to-shoulder and pass buffers down the stream, in the style of a bucket brigade. When a BSubscriber receives a buffer it does something to it—typically, it examines, adds to, or filters the data it finds there—and then passes it to the next BSubscriber (or, more accurately, lets the server pass it to the next BSubscriber).

The media servers take care of managing the data buffers in their streams—they allocate new buffers, pass them between BSubscribers, clear existing buffers for re-use, and so on. A BSubscriber's primary tasks are these (and in this order):

- Identifying the media server that it wants to get buffers from.
- Applying for aceptance into one of the server's streams (this is called "subscribing").
- Entering the stream. At the moment a BSubscriber enters a stream, the object begins receiving data buffers from the server.
- Processing the data that it finds in the buffers that it receives.

The BSubscribers that subscribe to the same stream needn't belong to the same application. This means that your BSubscriber may be examining, adding to, or filtering data that was generated in another application.

Most buffer streams need to "flow" quickly and uninterruptedly (this is especially true of the Audio Server's streams). The processing that a single BSubscriber performs when it receives a buffer from the server should be as brief and efficient as possible.

Identifying a Server

BSubscriber is an abstract class—you never construct instances of BSubscriber directly. Instead, you construct instances of one of its derived classes. Each BSubscriber-derived class provided by the Media Kit corresponds to a particular media server. Identifying a server, therefore, is implied by the act of choosing a BSubscriber-derived class with which you instantiate an object.

Currently, the only BSubscriber-derived class that's supplied by the Media Kit is BAudioSubscriber. Instances of this class receive buffers from, obviously enough, the Audio Server.

Subscribing

The first thing you do with your BSubscriber object, once you've constructed it, is to ask its server's permission to be sent buffers of data. This is performed through the **Subscribe()** function. Subscription doesn't cause buffers to actually be sent, but it does get the BSubscriber into the ballpark. The act by which a BSubscriber receives buffers (the **EnterStream()** function) depends on a successful subscription.

As part of a BSubscriber's subscription, it must tell the server which stream it wants to enter, which other BSubscribers it's willing to share the buffer stream with, and whether it's willing to wait for "undesirable" brethren to get out of the stream before it gets in. The object's opinions on these topics are registered through arguments to the **Subscribe()** function:

> long **Subscribe(**long *stream*, subscriber_id *clique*, bool *willWait***)**

The arguments are discussed in the following sections.

The Stream

A server can shepherd more than one stream. For example, the Audio Server controls access to two streams: The sound-out stream terminates at the DAC, the sound-in stream begins at the ADC. You identify the stream you want by using one of the stream constants defined by the server. The Audio Server defines the constants **B_DAC_STREAM** for sound-out and **B_ADC_STREAM** for sound-in.

A BSubscriber may only subscribe to one stream at a time.

The Clique

Note: The clique concept is being reconsidered. You can still set a clique value, but the mechanism may be removed in a subsequent release, or moved into a different level of the Kit's software. For now, it's recommended that you *always* set the clique to **B_SHARED_SUBSCRIBER_ID**.

A BSubscriber's clique (passed as the *clique* argument to **Subscribe()**) identifies the cabal of BSubscribers that the calling object is willing to share the server's buffer stream with. The value of *clique* acts as a "key" to the stream: To gain access to the stream, you have to have the proper key.

Here's how it works: The first BSubscriber that calls **Subscribe()** passes some value as the *clique* argument. This value becomes the key to the buffer stream; any other BSubscriber that wants to subscribe to that stream must pass the same clique value (unless you want to be an "invisible" subscriber, as described in the next section). The actual value that's used to represent the clique is irrelevant; matching is the only concern. A given clique is enforced until all subscribed objects have *unsubscribed* (through the **Unsubscribe()** function) at which point the next object that subscribes will establish a new clique value.

Note: The *clique* argument is type cast as a **subscriber_id**. Such values are tokens that uniquely identify BSubscriber objects among all extant BSubscribers of the same class (across all applications). That the clique is represented as a **subscriber_id** is primarily a convenience: Just as the actual clique value has no significance, neither does its type imply any special properties about the clique.

Choosing a Clique Value

With regard to cliques, there are four types of BSubscribers: Those that want utterly exclusive access to the buffer stream, those that are willing to share access with certain (but not all) other BSubscribers, those that will share with any other BSubscriber, and those that want to crash the party. The clique value that you choose depends on which of these characterizations describes your BSubscriber:

- *Exclusive access.* If a BSubscriber wants to have exclusive access to the stream—if it doesn't want any other BSubscriber to be able to enter the stream while it's subscribed—then the object passes some value as the *clique* argument, but keeps the value a secret. Typically, the object's own **subscriber_id** value is used as the argument; the **ID()** function supplies this value:

```
/* FirstSub is assumed to be a valid BSubscriber
 * object (currently, it must be an instance of
 * BAudioSubscriber).
 */
subscriber_id firstID = FirstSub->ID();
FirstSub->Subscribe(..., firstID, ...);
```

- *Selective sharing.* If the first subscriber wants to share the stream with subsequent subscribers, the initial clique value must be used in those subsequent subscriptions:

```
/* First... */
subscriber_id firstID = FirstSub->ID();
FirstSub->Subscribe(..., firstID, ...);
...

/* Notice that the second subscriber passes the
 * first subscriber's ID value as the clique argument.
```

```
         */
        SecondSub->Subscribe(..., firstID, ...);
```

To share the stream with BSubscribers in other applications, the first subscriber's application would have to broadcast the first subscriber's ID value (through a BMessage, for example).

- *Indiscriminate sharing.* To share the stream between all BSubscribers in all applications is easy: You pass the **B_SHARED_SUBSCRIBER_ID** constant as the value for *clique*:

  ```
  FirstSub->Subscribe(..., B_SHARED_SUBSCRIBER_ID, ...);
  ```

  Note, however, that the **B_SHARED_SUBSCRIBER_ID** clique doesn't *guarantee* that every BSubscriber will be allowed in the stream. If an unsharing BSubscriber has already set the clique to some other value, a BSubscriber that passes **B_SHARED_SUBSCRIBER_ID** will be turned down. Conversely, if the clique is set to **B_SHARED_SUBSCRIBER_ID** and a BSubscriber comes along that tries to subscribe with a less generous clique value, it's subscription will be denied.

- *Gate crashing.* If you just don't care who's in the stream or whether they like you or not, use the constant **B_INVISIBLE_SUBSCRIBER_ID** as the *clique* value. This will get you in regardless of—and without changing—the current clique setting. If you're the first subscriber, the next subscriber will be allowed in regardless of his clique specification (and the stream's clique will be set to this subsequent value).

Waiting for Access

If a BSubscriber is denied access to a server because it didn't pass the clique test, it can either give up immediately, or wait for the current clique members to unsubscribe. This is expressed in **Subscribe()**'s final argument, the boolean *willWait*:

- If *willWait* is **FALSE**, **Subscribe()** returns immediately, regardless of its success in gaining access to the server. (The measure of its success is given by the function's return value.)
- If it's **TRUE**, the function doesn't return until the BSubscriber has successfully subscribed. There's no time-out provision, so the wait is indefinite. (Yes, there is a **SetTimeout()** function; no, it doesn't apply to subscription.)

Entering the Stream

Having successfully subscribed to a server's stream, the BSubscriber's next task is to enter the stream. By this, the object will begin receiving buffers of data. You do this through the **EnterStream()** function:

```
virtual long EnterStream(subscriber_id neighbor,
                         bool before,
                         void *arg,
                         enter_stream_hook enterHook,
                         exit_stream_hook exitHook,
                         bool background)
```

The function's operations and arguments are described in the following sections.

Positioning your BSubscriber

The first two **EnterStream()** arguments position the BSubscriber with respect to the other BSubscriber objects that are already in the stream (if any):

EnterStream(subscriber_id *neighbor*, bool *before*, ...**)**

The *neighbor* argument identifies the BSubscriber (by its ID number, as returned by the **ID()** function) that you want the entering BSubscriber to stand next to; *before* places the entering object before (**TRUE**) or after (**FALSE**) the neighbor. The neighbor needn't belong to the same application as the entering object, but it must already have entered the stream.

If you want to place the BSubscriber at one or the other end of the stream (or to add the first BSubscriber to the stream), you pass **NULL** as the neighbor. A before value of **TRUE** thus places the BSubscriber at the "front" of the stream (the object will be the first to receive each buffer that flows through the stream), and a value of **FALSE** places it at the "back" (it's the last to receive buffers before they're realized or recycled).

A BSubscriber's position in the stream can't be locked. If, for example, you place your BSubscriber to stand at the back of the stream, some other BSubscriber—from some other application, possibly—can come along later and also claim the back. Your object will be bumped forward (towards the front of the stream) in deference to the newcomer.

Receiving and Processing Buffers

After your BSubscriber has entered the buffer stream, it will begin receiving buffers of data. The third, fourth, and last arguments to **EnterStream()** pertain to the means by which your object receives these buffers:

EnterStream(..., void **arg*, enter_stream_hook *entryHook*, ..., bool *background***)**

The arguments, taken out of order, are:

- *entryHook* is a pointer to a boolean function (the complete protocol is given below) that will be invoked once for each buffer that's received.
- *arg* is a pointer-sized value that will be passed as an argument to *entryrHook*.

- The value of *background* is used to determine whether *entryHook* will be executed in a separate thread (**TRUE**) or in the same thread (**FALSE**) as that in which **EnterStream()** was called. If you run in the background, **EnterStream()** returns immediately; if not, the function doesn't return until the object has exited the stream.

Of initial interest, here, is the "entry hook" that you must supply: This is global C function or static C++ member function that's invoked once for each buffer that the BSubscriber receives. The protocol for the function (which is **typedef**'d as **enter_stream_hook**) is:

bool ***stream_function*(**void *arg, char *buffer, long count**)**

- *arg* is the same as the *arg* argument that you passed to **EnterStream()**.
- *buffer* is a pointer to the buffer that has just arrived.
- *count* is the number of bytes of data in the buffer.

You have to implement the entry hook yourself; the Media Kit doesn't supply any entry hook candidates. From within your implementation of the function, you're expected to process the data in *buffer* as fits your intentions. As mentioned earlier, your processing should be designed with efficiency in mind. The only rule by which you should abide is this:

Don't Clear the Buffer

If you're generating data, you should *add* it into the data that you find in the buffer. Thank-you.

When you're done with your processing, you simply return from the entry hook. You don't have to do anything to send the buffer to the next BSubscriber in the stream; the Media Kit takes care of that for you. The value that the stream function returns is important: If it returns **TRUE**, the BSubscriber continues receiving buffers; if it returns **FALSE**, the object is removed from the stream.

Exiting the Stream

There are two ways to remove a BSubscriber from a stream. The first was mentioned above: Return **FALSE** from the stream function. The second method is to call **ExitStream()** directly. The **ExitStream()** function is particularly useful if you're running the stream function in the background and you want to pull the trigger from another thread.

Whichever method is used, the BSubscriber's "exit hook" is invoked upon exiting the stream. This is an optional call-back function, similar to the stream function in its application, that you supply as the fifth argument to **EnterStream()**:

EnterStream(..., exit_stream_hook *exitHook*, ...**)**

The protocol for the completion function is:

long ***completion_function*(**void **arg*, long *error***)**

- The *arg* value is, again, taken from the **EnterStream()** call.
- *error* is a code that explains why the BSubscriber is exiting the stream.

Normally, *error* is **B_NO_ERROR**. This means that the BSubscriber is exiting naturally: Either because the stream function returned **FALSE** or because **ExitStream()** was called. If error is **B_TIMED_OUT**, then the BSubscriber is exiting because of a delay in receiving the next buffer. (You set the time-out limit through BSubscriber's **SetTimeout()** function, specifying the limit in microseconds; by default the object will wait forever.) Any other error code will have been generated by a lower-level entity and can be lumped into the general category of "something went wrong."

The completion function is executed in the same thread as the stream function. If this isn't a background thread, the value returned by the completion function is then returned by **EnterStream()**. If you *are* using a background thread, the return value is lost.

You can perform whatever clean-up is necessary in your implementation of the completion function. The only thing that you mustn't do in the completion function is delete the BSubscriber itself.

Processing Data in a Member Function

Typically, the stream functions is implemented as a "dummy" static member function of some class. In this case, **EnterStream()**'s *arg* argument is a pointer to an instance of that class. In the implemention of the static function, the "real" stream function is invoked on the *arg* pointer that the function receives. The class that implements the functions derive from BSubscriber.

For example, in the (fictitious) SoundDuller class, a static function called **_dull_sound()** and a non-static function **DullSound()** are defined. Both of these functions are private. In addition, it defines public **Start()** and Stop() functions that will run the show, and some private variables—including a BAudioSubscriber object—that it requires to perform:

```
class SoundDuller : public BObject
{
    public:
        void Start(void);
        void Stop(void);

    private:
        static bool _dull_sound(void *arg,
                                char *buf,
                                long count);
        bool DullSound(char *buf, long count);

        BAudioSubscriber a_sub;
        short previous;
}
```

The implementation of **_dull_sound()** casts the arg pointer and then invokes **DullSound()**:

```
bool SoundDuller::_dull_sound(void *arg, char *buf, long count)
{
    return (((SoundDuller *)arg)->DullSound(buf,count));
}
```

DullSound() performs the actual stream data processing. The function shown here implements a simple low-pass filter (the "HelloWorld" of signal processing). The function assumes that the stream data is one channel of 16-bit sound:

```
bool SoundDuller::DullSound(char *buf, long count)
{
    long short_count = count/2;
    short *s_buf = (short *)buf;

    while (short_count-- > 0) {
        *s_buf += previous;
        previous = *s_buf++;
    }
}
```

The **Start()** function initializes the BAudioSubscriber and the **previous** variable, and then calls **EnterStream()**:

```
void SoundDuller::Start(void)
{
    if (a_sub.Subscribe(B_DAC_STREAM, B_SHARED_SUBSCRIBER_ID,
                        FALSE) < B_NO_ERROR)
        return;
    previous = 0;

    /* Enter at the stream's tail; run in the background. */
    a_sub.EnterStream(NULL, FALSE,
                        this, _make_dull, NULL, TRUE);
}
```

Stop() removes the subscriber from the stream by calling **ExitStream()**. The function's argument says whether we want to wait until the object is *really* out of the stream; it's always a good idea to re-synchronize if the subscriber is running in the background:

```
void SoundDuller::Stop(void)
{
    a_sub.ExitStream(TRUE);
    a_sub.Unsubscribe();
}
```

Sound details used in this example, such as the meaning of the **B_DAC_STREAM** constant, are explained in the BAudioSubscriber class. For another example of a stream function implementation, see the BSoundFile class.

Constructor and Destructor

BSubscriber()

BSubscriber(const char **name* = NULL**)**

Creates and returns a new BSubscriber object. The object can be given a name; the name needn't be unique.

After creating a BSubscriber, you typically do the following (in this order):

- Subscribe the object to a buffer stream by calling **Subscribe()**.
- Allow the object to begin receiving buffers by calling **EnterStream()**.

The construction of a BSubscriber never fails. This function doesn't set the object's **Error()** value.

See also: **Subscribe(), EnterStream()**

~BSubscriber()

virtual **~BSubscriber(**void**)**

Destroys the BSubscriber. You should never delete a BSubscriber from within an implementation of the object's stream function or completion function.

It isn't necessary to tell the object to exit the buffer stream or to unsubscribe it before deleting. These actions will happen automatically.

Member Functions

Clique()

subscriber_id **Clique(**void**)**

Returns the clique (a **subscriber_id** value) that this BSubscriber used in its most recent attempt to subscribe. The attempt need not have been successful, nor is there any guarantee that the object hasn't since unsubscribed. If the object hasn't attempted to subscribe, this returns **B_NO_SUBSCRIBER_ID**.

See also: **Subscribe()**

EnterStream()

```
virtual long EnterStream(subscriber_id neighbor,
                         bool before,
                         void *arg,
                         enter_stream_hook streamFunction,
                         exit_stream_hook completionFunction,
                         bool background)
```

Causes the BSubscriber to begin receiving buffers of data from its stream. The object must have successfully subscribed (through a call to **Subscribe()**) for this function to succeed.

The arguments to this function (and the function in general) is the topic of most of the overview to this class; look there for the whole story. Briefly, the arguments are:

- *neighbor* identifies the BSubscriber that this object will stand next to in the buffer stream. If neighbor is **NULL**, this BSubscriber will be positioned at the front or the back of the stream (depending on the value of the next argument).

- *before*, if **TRUE**, places this BSubscriber immediately before neighbor in the stream. If it's **FALSE**, this object is placed after neighbor. If neighbor was **NULL**, this object is placed at the front or back of the stream as *before* is **TRUE** or **FALSE**.

- *arg* is a pointer-sized value that's forwarded as an argument to the stream and completion functions (specified in the next two arguments to **EnterStream()**).

- *streamFunction* is a global function that's called once for every buffer that's sent to the BSubscriber. The protocol for the function is:

  bool ***stream_function*****(**void **arg*, char **buffer*, long *count***)**

  The *arg* argument, here, is taken literally as the *arg* value passed to **EnterStream()**. A pointer to the buffer itself is passed as *buffer*; *count* is the number of bytes of data in the buffer. If the stream function returns **TRUE**, the object continues to receive buffers; if it returns **FALSE**, it exits the stream.

- *completionFunction* is a global function that's called after the BSubscriber has finished processing its last buffer. Its protocol is:

  long ***completion_function*****(**void **arg*, long *error***)**

  arg, again, is taken from the argument to **EnterStream()**. *error* is a code that describes why the object is leaving the stream: **B_NO_ERROR** means that the object has received an **ExitStream()** call, or that the stream function returned **FALSE**; an error of **B_TIMED_OUT** means the time limit between buffer receptions (as set through **SetTimeout()**) has expired. If the function isn't running in the background (as described in the next argument), the value returned by the completion function becomes the value that's returned by **EnterStream().**

  The completion function is optional. A value of **NULL** is accepted.

- *background*, if **TRUE**, causes the stream and completion functions to be executed in a separate thread (the Kit spawns the thread for you). In this case, **EnterStream()** returns immediately. If it's **FALSE**, the functions are executed synchronously within the **EnterStream()** call.

If the designated neighbor isn't in the buffer stream, **EnterStream()** returns **B_SUBSCRIBER_NOT_FOUND**. If the BSubscriber is already in the stream, **B_BAD_SUBSCRIBER** is returned.

If *background* is **TRUE**, **EnterStream()** immediately returns **B_NO_ERROR**; if it's **FALSE**, **EnterStream()** returns the value returned by the completion function. If a completion function isn't supplied, **EnterStream()** returns a value that indicates the success of the communication with the server; unless something's gone wrong, it should return **B_NO_ERROR**. In all cases, the **Error()** value is set to the value returned here.

See also: **ExitStream()**

Error()

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

Returns an error code that reflects the success of the function that was most recently invoked upon this object. The error codes that a particular function uses are listed in that function's description.

ExitStream()

virtual long **ExitStream(**bool *andWait* = FALSE**)**

Causes the BSubscriber to leave the buffer stream after it completes the processing of its current buffer. If *andWait* is **TRUE**, the function doesn't return until the object has completed processing this final buffer and has actually left the stream. If a completion function was supplied in the **EnterStream()** invocation, it will run to completion before **ExitStream()** returns. If *andWait* is **FALSE** (the default), **ExitStream()** returns immediately.

If the object isn't in the stream, the **B_SUBSCRIBER_NOT_FOUND** is returned. Otherwise the function returns **B_NO_ERROR**.

Note: In release 1.1d7, **ExitStream()** doesn't return a reliable value—but it does set the error code properly.

See also: **EnterStream()**

ID()

subscriber_id **ID(**void**)**

Returns the **subscriber_id** value that uniquely identifies this BSubscriber. A subscriber ID is issued when the object subscribes to a stream; it's withdrawn when the object unsubscribes. ID values are used, primarily, to position a BSubscriber with respect to some other BSubscriber within a buffer stream.

If the BSubscriber isn't currently subscribed to a stream, **B_NO_SUBSCRIBER_ID** is returned.

IsInStream()

bool **IsInStream(**void**)**

Returns **TRUE** if the object is currently in a stream; otherwise it returns **FALSE**.

Name()

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

Returns a pointer to the name of the BSubscriber. The name is set through an argument to the BSubscriber constructor.

SetTimeout(), Timeout()

void **SetTimeout(**double *microseconds***)**
double **Timeout(**void**)**

These functions set and return the amount of time, measured in microseconds, that a BSubscriber that has entered the buffer stream is willing to wait from the time that it finishes processing one buffer till the time that it gets the next. If the time limit expires before the next buffer arrives, the BSubscriber exits the stream and the completion function is called with its *error* argument set to **B_TIMED_OUT**.

A time limit of 0 (the default) means no time limit—the BSubscriber will wait forever for its next buffer.

See also: **EnterStream()**

StreamParameters()

long **StreamParameters(**long **bufferSize***,**
long **bufferCount***,**
bool **isRunning***,**
long **subscriberCount***,**
subscriber_id **clique***)**

Returns information about the stream to which the BSubscriber is currently subscribed:

- *bufferSize* is the size, in bytes, of the buffers that the object will receive.
- *bufferCount* is the number of buffers that are used in the stream.
- *isRunning* is **TRUE** if the stream is currently running, and **FALSE** if it isn't.
- *subscriberCount* is the number of BSubscriber objects that are currently subscribed to the stream (whether or not they've actually entered).
- *clique* is the currently enforced clique value for the stream.

You can set the buffer size and buffer count parameters (and so fine-tune the latency of the stream) through the **SetStreamBuffers()** function. *isRunning* can be toggled through calls to **StartStreaming()** and **StopStreaming()**. The other two parameters (*subscriberCount* and *clique*) vary as subscribers come and go.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_BAD_SUBSCRIBER** is returned. Otherwise, the function returns **B_NO_ERROR**.

SetStreamBuffers()

long **SetStreamBuffers(**long *bufferSize*, long *bufferCount***)**

Sets the size (in bytes) and number of buffers that are used to transport data through the stream. Although it's up to the server to provide reasonable default values, you can fine-tune the performance of the stream by fiddling with this function:

- By decreasing the size and/or number of buffers, you can decrease the maximum latency of the stream (the time it takes for a buffer to get from one end of the stream to the other). However, if you go too far in this direction, you run the risk of falling out of real time.
- By increasing the buffer size and count, you help ensure the real-time integrity of the stream, but you increase its maximum latency.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_RESOURCE_UNAVAILABLE** is returned. Otherwise, the function returns **B_NO_ERROR**.

The Audio Server initializes its streams to use eight buffers (per stream), where each buffer is a single page (4096 bytes). Currently, there's no way to automatically restore these default values after you've mangled one of the audio streams.

StartStreaming(), StopStreaming()

long **StartStreaming(**void**)**
long **StopStreaming(**void**)**

Starts and stops the passing of buffers through the stream to which the BSubscriber is subscribed. By default, the stream begins running when the first BSubscriber enters it, and it stops when the final remaining BSubscriber exits. You should only need to call **StartStreaming()** or **StopStreaming()** if you want to interrupt this automation.

You must have successfully subscribed to the stream to call this function. If you haven't, **B_RESOURCE_UNAVAILABLE** is returned. Otherwise, the function returns **B_NO_ERROR**.

Subscribe()

virtual long **Subscribe(**long *stream*, subscriber_id *clique,* bool *willWait***)**

Asks for admission into the server's list of BSubscribers to which it (the server) will send buffers of data. Subscribing doesn't cause the BSubscriber to begin receiving buffers, it simply gives the object the *right* to do so. (To receive buffers, you must invoke **EnterStream()** on a BSubscriber that has successfully subscribed.)

The arguments are described fully in the overview to this class. Briefly, they are:

- *stream* is a constant that identifies the specific stream within the server that you wish to subscribe to. The Audio Server provides two stream constants: **B_DAC_STREAM** (sound-out), and **B_ADC_STREAM** (sound-in).

  The *clique* argument is used as a "key" to the server. If there are no other currently-subscribed objects, any clique value is accepted and the BSubscriber is admitted. Subsequent subscriptions (by other BSubscribers) are then denied if they don't match this clique value. Conversely, if some other object has successfully subscribed (and hasn't since unsubscribed) this object must pass the clique value by which the currently-subscribed object gained admittance. The special **B_INVISIBLE_SUBSCRIBER_ID** value, when used as the clique, will let you invade any stream, any time.

  Note: As mentioned in the overview to this class, its recommended that you set the *clique* argument to **B_SHARED_SUBSCRIBER_ID**. The clique concept will either be removed or transferred to some other class in a subsequent release.

- The *willWait* argument tells the server whether this BSubscriber will wait for the coast to clear if the immediate attempt to subscribe is denied.

A successful subscription returns **B_NO_ERROR**. If the subscription is denied (because *stream* doesn't identify a valid stream, or the *clique* value isn't acceptable) and the BSubscriber isn't waiting, **Subscribe()** returns **RESOURCE_NOT_AVAILABLE**. The **Error()** value is set to the value returned directly here.

Note: The timeout value that you can set through the **SetTimeout()** function doesn't apply to subscription (it only applies to the inter-buffer lacuna). A BSubscriber that's willing to wait for admission might be waiting a long time.

See also: **Unsubscribe()**

Timeout() *see* SetTimeout()

Unsubscribe()

virtual long **Unsubscribe(**void**)**

Revokes the BSubscriber's access to its media server and sets its subscriber ID to **B_NO_SUBSCRIBER_ID**. If the object is currently in a stream, it automatically exits the stream and the object's completion function is called.

When you delete a BSubscriber, it's automatically unsubscribed.

If the object isn't currently subscribed, the function returns **B_BAD_SUBSCRIBER**. Otherwise, it returns **B_NO_ERROR**.

See also: **Subscribe()**

Global Functions, Constants, and Defined Types

This section lists parts of the Media Kit that aren't contained in classes.

Global Functions

beep()

<media/Beep.h>

sound_handle **beep(**void**)**

beep() plays the system beep. The sound is played in a background thread and **beep()** returns immediately. If you want to re-synchronize with the sound playback, pass the **sound_handle** token (returned by this function) as the argument to **wait_for_sound()**. This will cause your thread to wait until the sound has finished playing.

beep() will mix other sounds, but it never waits if the immediate attempt to play is thwarted.

play_sound()

<media/Beep.h>

```
sound_handle play_sound(record_ref soundRef,
                        bool willMix,
                        bool willWait,
                        bool background)
```

Plays the sound file identified by *soundRef*. The *willMix* and *willWait* arguments are used to determine how the function behaves with regard to other sounds:

- If you want your sound to play all by itself, set *willMix* to **FALSE**. If you don't care if it's mixed with other sounds, set it to **TRUE**.
- If you want your sound to play immediately (whether or not you're willing to mix), set *willWait* to **FALSE**. If you're willing to wait for the sound playback resources to become available, set *willWait* to **TRUE**.

Note that setting *willMix* to **TRUE** doesn't ensure that your sound will play immediately. If the sound playback resources are claimed for exclusive access by some other process, you'll be blocked, even if you're willing to mix.

The background argument, if **TRUE**, tells the function to spawn a thread in which to play the sound. The function, in this case, returns immediately. If background is **FALSE**, the sound is played synchronously and **play_sound()** won't return until the sound has finished.

The **sound_handle** value that's returned is a token that represents the sound playback. This token is only valid if you're playing in the background; you would use it in a subsequent call to **stop_sound()** or **wait_for_sound()**. If the ref doesn't represent a file, or if the sound couldn't be played, for whatever reason, **play_sound()** returns a negative integer.

stop_sound()

<media/Beep.h>

long **stop_sound(**sound_handle *handle***)**

Stops the playback of the sound identified by *handle*, a value that was returned by a previous call to **beep()** or **play_sound()**. The return value can be ignored.

wait_for_sound()

<media/Beep.h>

long **wait_for_sound(**sound_handle *handle***)**

Causes the calling thread to block until the sound identified by *handle* has finished playing. The *handle* value should have been returned by a previous call to **beep()** or **play_sound()**. Currently, **wait_for_sound()** always returns **B_NO_ERROR**.

Constants

Byte Order Constants

<media/MediaDefs.h>

Constant	Meaning
B_BIG_ENDIAN	MSB first
B_LITTLE_ENDIAN	LSB first

These constants are used by BAudioSubscriber and BSoundFile objects to describe the order of bytes within a sound sample.

Sound File Formats

<media/SoundFile.h>

Constant	Meaning
B_UNKNOWN_FILE	The file contains “raw” data
B_AIFF_FILE	AIFF format
B_WAVE_FILE	WAVE format
B_UNIX_FILE	Sun/NeXT/SGI etc. format

These constants represent the sound file formats that are recognized by the BSoundFile class.

Media Thread Priority

<media/MediaDefs.h>

Constant	Value
B_MEDIA_LEVEL	Same as **B_REAL_TIME_PRIORITY**

All threads that are spawned by the Media Kit are given a priority of **B_MEDIA_LEVEL**; this is the same as **B_REAL_TIME_PRIORITY**, the highest priority defined by the Kernel Kit.

No-Change Constant

<media/MediaDefs.h>

Constant	Meaning
B_NO_CHANGE	Don’t change the value of this parameter

The **B_NO_CHANGE** constant is used in multiple-parameter-setting functions (such as BAudioSubscriber’s **SetSampleParameters()** to indicate that you don’t want a particular parameter to change its current setting (while changing the values of other parameters).

Sample Format Constants

<media/MediaDefs.h>

Constant	Meaning
B_LINEAR_SAMPLES	Linear quantization
B_FLOAT_SAMPLES	Floating-point samples
B_MULAW_SAMPLES	Mu-law encoding
B_UNDEFINED_SAMPLES	Anything else

These constants represent the sample formats that are recognized by the sound hardware.

Subscriber IDs

<media/MediaDefs.h>

Constant	Meaning
B_SHARED_SUBSCRIBER_ID	Share the stream with other subscribers.
B_INVISIBLE_SUBSCRIBER_ID	Subscribe to the stream regardless of the clique.
B_NO_SUBSCRIBER_ID	The BSubscriber object isn't subscribed.

The first two subscriber ID constants are most commonly used as "clique" values, passed to the **EnterStream()** function. The final ID, **B_NO_SUBSCRIBER_ID**, is the default, subscriber-isn't-subscribed subscriber ID value.

The subscriber ID constants are type as **subscriber_id** values.

Defined Types

sound_handle

<media/Beep.h>

typedef sem_id **sound_handle**

The **sound_handle** type is a token that represents sounds that are currently being played through calls to **beep()** or **play_sound()**.

subscriber_id

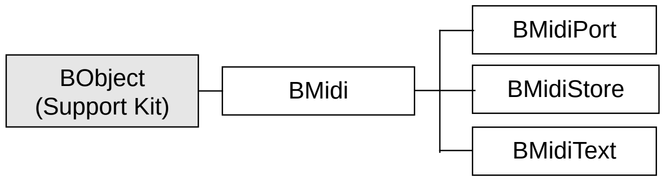
<media/MediaDefs.h>

typedef sem_id **subscriber_id**

The **subscriber_id** type is a token that uniquely identifies—system-wide—a BSubscriber object for a particular server.

6 The Midi Kit

Midi Kit Inheritance Hierarchy



6 The Midi Kit

The Musical Instrument Digital Interface (MIDI) is a standard for representing and communicating musical data. Its fundamental notion is that instantaneous musical events generated by a digital musical device can be encapsulated as "messages" of a known length and format. These messages can then be transmitted to other computer devices where they're acted on in some manner. The MIDI standard allows digital keyboards to be de-coupled from synthesizer boxes, lets computers record and playback performances on digital instruments, and so on.

The Midi Kit understands the MIDI software format (including Standard MIDI Files). With the Kit, you can create a network of objects that generate and broadcast MIDI messages. Applications built with the Midi Kit can read MIDI data that's brought into the computer through a MIDI port, process the data, write it to a file, and send it back out through the same port. The Kit contains four classes:

- The BMidi class is the centerpiece of the Kit. It defines the tenets to which all MIDI-processing objects adhere, and provides much of the machinery that realizes these ideas. BMidi is abstract—you never create direct instances of the class. Instead, you construct and connect instances of the other Kit classes, all of which derive from BMidi. You can also create your own classes that derive from BMidi.
- BMidiPort knows how to read MIDI data from and write it to a MIDI hardware port.
- BMidiStore provides a means for storing MIDI data, and for reading, writing, and performing Standard MIDI Files.
- BMidiText is a debugging aid that translates MIDI messages into text and prints them to standard output. You should only need this class while you're designing and fine-tuning your application.

To use the Midi Kit, you should have a working knowledge of the MIDI specification; no attempt is made here to describe the MIDI software format.

The BeBox comes equipped with four MIDI hardware ports. These are standard MIDI ports that accept standard MIDI cables—you don't need a MIDI interface box. The ports are aligned vertically at the back of the computer. Top-to-bottom they are MIDI-In A, MIDI-Out A, MIDI-In B, and MIDI-Out B. Currently, the Midi Kit only talks to the top set of ports (MIDI-In A and MIDI-Out A).

BMidi

Derived from:	public BObject
Declared in:	<midi/Midi.h>

Overview

BMidi is the centerpiece of the Midi Kit. It provides base class implementations of the functions that create a MIDI performance. BMidi is abstract; all other Kit classes—and any class that you want to design to take part in a performance—derive from BMidi. When you create a BMidi-derived class, you do so mainly to re-implement the hook functions that BMidi provides. The hook functions allow instances of your class to behave in a fashion that the other objects will understand.

The functions that BMidi defines fall into four categories:

- *Connection functions*. The connection functions let you connect the output of one BMidi object to the input of another BMidi object.
- *Message-generation functions*. Some BMidi objects generate (or otherwise procure) MIDI data. To do this, a derived class must implement the **Run()** hook function. **Run()** is the brains of a MIDI performance; other performance functions, such as **Start()** and **Stop()** control the performance.
- *"Spray" functions*. If a BMidi object wants to send a MIDI message to other BMidi objects, it does so by calling one of the output, or "spray," functions. There's a spray function for each type of MIDI message; for example, **SprayNoteOn()** corresponds to MIDI's Note On message. When a message is sprayed, it's sent to each of the objects that are connected to the output of the sprayer.
- *Input functions*. When a message is sprayed, the receivers of the message are notified by the automatic invocation of particular "input" functions. For example, when a BMidi object calls **SprayNoteOn()**, each of the objects that it's connected to becomes the target of the **NoteOn()** function. How the receiving object responds depends on the object's class: The input functions are virtual; the BMidi class implementations are empty.

Forming Connections

A fundamental concept of the Midi Kit is that MIDI data should "stream" through your application, passing from one BMidi-derived object to another. Each object does whatever it's designed to do: Sends the data to a MIDI port, writes it to a file, modifies it and passes it on, and so on.

You form the chain of BMidi objects that propagate MIDI data by connecting them to each other. This is done through BMidi's **Connect()** function. The function takes a single argument: The object you want the caller to connect to. By calling **Connect()**, you connect the output of the calling object to the input of the argument object.

For example, let's say you want to connect a MIDI keyboard to your computer, play it, and have the performance recorded in a file. To set this up, you connect a BMidiPort object, which reads data from the MIDI port, to a BMidiStore object, which stores the data that's sent to it and can write it to a file:

```
/* Connect the output of a BMidiPort to the input of a
 * BMidiStore.
 */
BMidiPort *m_port = new BMidiPort();
BMidiStore *m_store = new BMidiStore();

m_port->Connect(m_store);
```

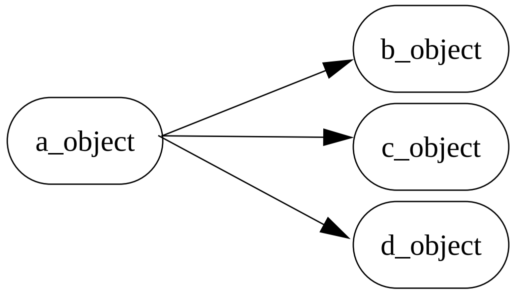
Simply connecting the objects isn't enough, however; you have to tell the BMidiPort to start listening to the MIDI port, by calling its **Start()** function. This is explained in a later section.

Once you've made the recording, you could play it back by re-connecting the objects in the opposite direction:

```
/* We'll disconnect first, although this isn't strictly
 * necessary.
 */
m_port->Disconnect(m_store);
m_store->Connect(m_port);
```

In this configuration, a **Start()** call to **m_store** would cause its MIDI data to flow into the BMidiPort (and thence to a synthesizer, for example, for realization).

You can connect any number of BMidi objects to the output of another BMidi object, as depicted below:



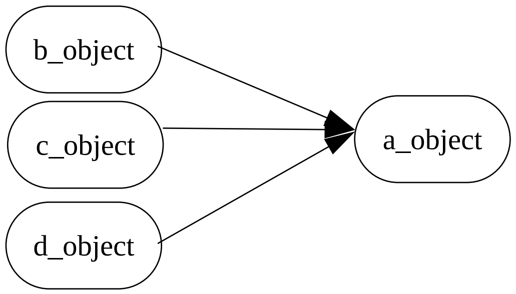
The configuration in the illustration is created thus:

```
    a_object->Connect(b_object);
    a_object->Connect(c_object);
    a_object->Connect(d_object);
```

Every BMidi object knows which objects its output is connected to; you can get a BList of these objects through the **Connections()** function. For example, **a_object**, above, would list **b_object**, **c_object**, and **d_object** as its connections.

Similarly, the same BMidi object can be the argument in any number of **Connect()** calls, as shown below and depicted in the following illustration:

```
    b_object->Connect(a_object);
    c_object->Connect(a_object);
    d_object->Connect(a_object);
```



When you use a BMidi object as the argument to a **Connect()** method, the argument object *isn't* informed. In the illustration, **a_object** *doesn't* know about the objects that are connected to its input.

Generating MIDI Messages

To generate MIDI messages, you implement the **Run()** function in a BMidi-derived class. An implementation of **Run()** should include a **while()** loop that produces (typically) a single MIDI message on each pass, and then sprays the message to the connected objects. To predicate the loop you test the value of the **KeepRunning()** boolean function.

The outline of a **Run()** implementation looks like this:

```
    void MyMidi::Run()
    {
        while (KeepRunning()) {
            /* Generate a message and spray it. */
        }
    }
```

Although your derived class can generate more than one MIDI message each time through the loop, it's recommended that you try to stick to just one.

To tell an object to perform its **Run()** function, you call the object's **Start()** function—you never call **Run()** directly. **Start()** causes the object to spawn a thread (its "run" thread) and execute **Run()** within it. When you're tired of the object's performance, you call its **Stop()** function.

The **Run()** function is needed in classes that want to introduce new MIDI data into a performance. For example, in its implementation of **Run()**, BMidiStore sprays messages that correspond to the MIDI data that it stores. In its **Run()**, a BMidiPort reads data from the MIDI port and produces messages accordingly. If you're generating MIDI data algorithmically, or reading your own file format (BMidiStore can read standard MIDI files), then you'll need to implement **Run()**. If, on the other hand, you're creating an object that "filters" data—that accepts data at its input, modifies it, then sprays it—you won't need **Run()**.

Another point to keep in mind is that the **Run()** function can run ahead of real time. It doesn't have to generate and spray data precisely at the moment that the data needs to be realized. This is further explained in the section "Time" on page 10.

Important: The BMidi-derived classes that you create *must* implement **Run()**, even if they don't generate MIDI data; "do-nothing" implementations are acceptable, in this case. For example, if you're creating a filter (as described in a later section), your **Run()** function could be, simply

```
void MidiFilter::Run()
{}
```

Spray Functions

The spray functions are used (primarily) within a **Run()** loop to send data to the running object's connections (the objects that are connected to the running object's output). There's a separate spray function for each of the MIDI message types: **SprayNoteOn()**, **SprayNoteOff()**, **SprayPitchBend()**, and so on. The arguments that these functions take are the data items that comprise the specific messages. The spray functions also take an additional argument that gives the message a time-stamp, as explained later (again, in the "Time" section).

Input Functions

The input functions take the names of the MIDI messages to which they respond: **NoteOn()** responds to a Note On message; **NoteOff()** responds to a Note Off; **KeyPressure()** to a Key Pressure change, and so on. These are all virtual functions. BMidi doesn't provide a default implementation for any of them; it's up to each BMidi-derived class to decide how to respond to MIDI messages.

Input functions are never invoked directly; they're called automatically when a running object sprays MIDI data.

Every BMidi object automatically spawns an "input" thread when it's constructed. It's in this thread that the input functions are executed. The input thread is always running—the

Start() and **Stop()** functions don't affect it. As soon as you construct an object, it's ready to receive data.

For example, let's say, once again, that you have a BMidiPort connected to a BMidiStore:

```
m_port->Connect(m_store);
```

Now you open the port (a BMidiPort detail that doesn't extend to other BMidi-derived classes) and tell the BMidiPort to start running:

```
m_port->Open("midi1");
m_port->Start();
```

As the BMidiPort is running, it sends data to its output. Since the BMidiStore is connected to the BMidiPort's output, it receives this data automatically in the form of input function invocations. In other words, when **m_port** calls its **SprayNoteOn()** function (which it does in its **Run()** loop), **m_store**'s **NoteOn()** function is automatically called. As an instance of BMidiStore, the **m_store** object caches the data that it receives through the input functions.

You can derive your own BMidi classes that implement the input functions in other ways. For example the following implementation of **NoteOn()**, in a proposed class called NoteCounter, simply keeps track of the number of times each key (in the MIDI sense) is played:

```
void NoteCounter::NoteOn(uchar channel, uchar keyNumber,
                    uchar velocity, ulong time)
{
    /* We'll assume the class has allocated an array that
     * holds the key counters.
     */
    keyCounter[keyNumber]++;
}
```

Note that the **NoteOn()** function in the example includes a *time* argument (the other arguments should be familiar if you understand the MIDI specification). This argument is explained in the "Time" section.

Creating a MIDI Filter

Some BMidi classes may want to create objects that act as filters: They receive data, modify it, and then pass it on. To do this, you call the appropriate spray functions from within the implementations of the input functions. Below is the implementation of the **NoteOn()** function for a proposed class called Transposer. It takes each Note On, transposes it up a half step, and then sprays it:

```
void Transposer::NoteOn(uchar channel, uchar keyNumber,
                    uchar velocity, ulong time)
{
    uchar new_key = max(keyNumber + 1, 127);
    SprayNoteOn(channel, new_key, velocity, time);
}
```

There's a subtle but important distinction between a filter class and a "performance" class (where the latter is a class that's designed to actually realize the MIDI data it receives). The distinction has to do with time, and is explained in the next section. An implication of the distinction that affects the current discussion is that it may not be a great idea to invest, in a single object, the ability to filter *and* perform MIDI data. By way of calibration, both BMidiStore and BMidiPort are performance classes—objects of these classes realize the data they receive, the former by caching it, the latter by sending it out the MIDI port. In neither of these classes do the input functions spray data.

Time

Every spray and input function takes a final *time* argument. This argument declares when the message that the function represents should be performed. The argument is given as an absolute measurement in *ticks*, or milliseconds. Tick 0 occurs when you boot your computer; the tick counter automatically starts running at that point. To get the current tick measurement, you call the global, Kernel Kit-defined **system_time()** function and divide by 1000.0 (**system_time()** returns microseconds).

A convention of the Midi Kit holds that time arguments are applied at an object's input. In other words, the implementation of a BMidi-derived input function would look at the time argument, wait until the designated time, and then do whatever it does that it does do. However, this only applies to BMidi-derived classes that are designed to perform MIDI data, as the term was defined in the previous section. Objects that filter data *shouldn't* apply the time argument.

To apply the *time* argument, you call the **SnoozeUntil()** function, passing the value of *time*. For example, a "performance" **NoteOn()** function would look like this:

```
void MyPerformer::NoteOn(uchar channel, uchar keyNumber,
                    uchar velocity, ulong time)
{
    SnoozeUntil(time);
    /* Perform the data here. */
}
```

If *time* designates a tick that has already tocked, **SnoozeUntil()** returns immediately; otherwise it tells the input thread to snooze until the designated tick is at hand.

An extremely important point, with regard to The SnoozeUntil() function, as used here, may cause spraying objects (objects that are spraying

Spraying Time

If you're implementing the **Run()** function, then you have to generate a time value yourself which you pass as the final argument to each spray functionthat you call. The value you generate depends on whether you class runs in real time, or ahead of time.

Running in Real Time

If your class conjures MIDI data that needs to be performed immediately, you should use the **B_NOW** macro as the value of the *time* arguments that you pass to your spray functions. **B_NOW** is simply a cover for (**system_time()**/1000.0**)** (converted to an integer). By using **B_NOW** as the *time* argument you're declaring that the data should be performed in the same tick in which it was generated. This probably won't happen; by the time the input functions are called and the data realized, a few ticks will have elapsed. In this case, the expected **SnoozeUntil()** calls (within the input function implementations) will see that the time value has passed, and so will return immediately, allowing the data to be realized as quickly as possible.

The lag between the time that you generate the data and the time it's realized depends on a number of factors, such as how loaded down your machine is and how much processing your BMidi objects perform. But the Midi Kit machinery itself shouldn't impose a latency that's beyond the tolerability of a sensible musical performance.

Running Ahead of Time

If you're generating data ahead of its performance time, you need to compute the time value so that it pinpoints the correct time in the future. For example, if you want to create a class that generates a note every 100 milliseconds, you need to do something like this:

```
void MyTicker::Run()
{
    ulong when = B_NOW;
    uchar key_num;

    while (KeepRunning()) {

        /* Make a new note. */
        SprayNoteOn(1, 60, 64, when);

        /* Turn the note off 99 ticks later. */
        when += 99;
        SprayNoteOff(1, 60, 0, when);

        /* Bump the when variable so the next Note On
         * will be 100 ticks after this one.
         */
        when += 1;
    }
}
```

When a MyTicker object is told to start running, it generates a sequence of Note On/Note Off pairs, and sprays them to its connected objects. Somewhere down the line, a performance object will apply the time value by calling **SnoozeUntil()**.

Tethering MyTicker

But what, you may wonder, keeps MyTicker from running wild and generating thousands or millions of notes—which aren't scheduled to be played for hours—as fast as possible?

The answer is in the mechansim that connects a spray function to an input function: The BMidi class creates a port (in the Kernel Kit sense) for every object. When you invoke a spray function, the data is encoded in a message and written to each of the connected objects' ports. The input functions (invoked on the connected objects) then read from their respective ports. The secret here is that these ports are declared to be 1 (one) message deep. So, as long as one of the input function calls **SnoozeUntil()**, the spraying object will never be more than one message ahead.

A useful feature of this mechanism is that if you connect a series of BMidi object that *don't* invoke **SnoozeUntil()**, you can process MIDI data faster than real-time. For example, let's say you want to spray data from one BMidiStore object, pass the data through a filter, and then store it in another BMidiStore. The BMidiStore input functions don't call **SnoozeUntil()**; thus, data will flow out of the first object, through the filter, and into its destination as quickly as possible, allowing you to process hours of real-time data in just a few seconds. Of course, if you add a performance object into this mix (so you can hear the data while it's being processed), the data flow will be tethered, as described above.

Hook Functions

Run()	Contains a loop that generates and broadcasts MIDI messages.
Start()	Starts the object's run loop. Can be overridden to provide pre-running adjustments.
Stop()	Stops the object's run loop. Can be overridden to perform post-running clean-up.

The input functions (**NoteOn()**, **NoteOff()**, and so on) are also hook functions. These are listed in the section "Input and Spray Functions" on page 17.

Constructor and Destructor

BMidi()

BMidi(void**)**

Creates and returns a new BMidi object. The object's input thread is spawned and started in this function—in other words, BMidi objects are born with the ability to accept incoming messages. The run thread, on the other hand, isn't spawned until **Start()** is called.

~BMidi()

virtual **~BMidi(**void**)**

Kills the input and run threads after they've gotten to suitable stopping points (as defined below), deletes the list that holds the connections (but doesn't delete the objects contained in the list), then destroys the BMidi object.

The input thread is stopped after all currently-waiting input messages have been read. No more messages are accepted while the input queue is being drained. The run thread is allowed to complete its current pass through the run loop and then told to stop (in the manner of the **Stop()** function).

While the destructor severs the connections that this BMidi object has formed, it doesn't sever the connections from other objects to this one. For example, consider the following (improper) sequence of calls:

```
/* DON'T DO THIS... */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
delete b_midi;
```

The **delete** call severs the connection from **b_midi** to **c_midi**, but it doesn't disconnect **a_midi** and **b_midi**. You have to disconnect the object's "back-connections" explicitly:

```
/* ...DO THIS INSTEAD */
a_midi->Connect(b_midi);
b_midi->Connect(c_midi);
...
a_midi->Disconnect(b_midi);
delete b_midi;
```

See also: **Stop()**

Member Functions

Connect()

void **Connect(**BMidi **toObject***)**

Connects the BMidi object's output to *toObject*'s input. The BMidi object can connect its output to any number of other objects. Each of these connected objects receives an input function call as the BMidi sprays messages. For example, consider the following setup:

```
my_midi->Connect(your_midi);
my_midi->Connect(his_midi);
my_midi->Connect(her_midi);
```

The output of **my_midi** is connected to the inputs of **your_midi**, **his_midi**, and **her_midi**. When **my_midi** calls a spray function—**SprayNoteOn()**, for example—each of the other objects receives an input function call—in this case, **NoteOn()**.

Any object that's been the argument in a **Connect()** call should ultimately be disconnected through a call to **Disconnect()**. In particular, care should be taken to disconnect objects when deleting a BMidi object, as described in the destructor.

See also: **~BMidi()**, **Connections(), IsConnected()**

Connections()

inline BList ***Connections(**void**)**

Returns a BList that contains the objects that this object has connected to itself. In other words, the objects that were arguments in previous calls to **Connect()**. When a BMidi object sprays, each of the objects in its connection list becomes the target of an input function invocation, as explained in the class description.

See also: **Connect()**, **Disconnect()**, **IsConnected()**

Disconnect()

void **Disconnect(**BMidi **toObject***)**

Severs the BMidi's connection to the argument. The connection must have previously been formed through a call to **Connect()** with a like disposition of receiver and argument.

See also: **Connect()**

IsConnected()

inline bool **IsConnected(**BMidi **toObject***)**

Returns **TRUE** if the argument is present in the receiver's list of connected objects.

See also: **Connect()**, **Connections()**

IsRunning()

bool **IsRunning(**void**)**

Returns **TRUE** if the object's **Run()** loop is looping; in other words, if the object has received a **Start()** function call, but hasn't been told to **Stop()** (or otherwise hasn't fallen out of the loop).

See also: **Start()**, **Stop()**

KeepRunning()

protected:

bool **KeepRunning(**void**)**

Used by the **Run()** function to predicate its **while** loop, as explained in the class description. This function should *only* be called from within **Run()**.

See also: **Run()**, **Start()**, **Stop()**

Run()

private:

void **Run(**void**)**

A BMidi-derived class places its data-generating machinery in the **Run()** function, as described in the section "Generating MIDI Messages" on page 7.

See also: **Start()**, **Stop()**, **KeepRunning()**

SnoozeUntil()

void **SnoozeUntil(**ulong *tick***)**

Puts the calling thread to sleep until *tick* milliseconds have elapsed since the computer was booted. This function is meant to be used in the implementation of the input functions, as explained in the section "Time" on page 10.

Start()

virtual void **Start(**void**)**

Tells the object to begin its run loop and execute the **Run()** function. You can override this function in a BMidi-derived class to provide your own pre-running initialization. Make sure, however, that you call the inherited version of this function within your implementation.

See also: **Stop()**, **Run()**

Stop()

virtual void **Stop(**void**)**

Tells the object to halt its run loop. Calling **Stop()** tells the **KeepRunning()** function to return **FALSE**, thus causing the run loop (in the **Run()** function) to terminate. You can override this function in a BMidi-derived class to predicate the stop, or to perform post-performance clean-up (as two examples). Make sure, however, that you invoke the inherited version of this function within your implementation.

See also: **Start()**, **Run()**

Input and Spray Functions

The protocols for the input and spray functions are given below, grouped by the MIDI message to which they correspond (the input function for each group is shown first, the spray function is second).

See the class overview for more information on these functions.

Channel Pressure

```
virtual void ChannelPressure(uchar channel,
                                        uchar pressure,
                                        ulong time = B_NOW)
```

protected:

```
void SprayChannelPressure(uchar channel,
                                        uchar pressure,
                                        ulong time)
```

Control Change

```
virtual void ControlChange(uchar channel,
                                        uchar  controlNumber,
                                        uchar  controlValue,
                                        ulong time = B_NOW)
```

protected:

```
void SprayControlChange(uchar channel,
                                        uchar  controlNumber,
                                        uchar  controlValue,
                                        ulong time)
```

Key Pressure

```
virtual void KeyPressure(uchar channel,
                                        uchar note,
                                        uchar pressure,
                                        ulong time = B_NOW)
```

protected:

```
void SprayKeyPressure(uchar channel,
                                        uchar note,
                                        uchar pressure,
                                        ulong time)
```

Note Off

virtual void **NoteOff(**uchar *channel*,
uchar *note*,
uchar *velocity*,
ulong *time* = B_NOW**)**

protected:

void **SprayNoteOff(**uchar *channel*,
uchar *note*,
uchar *velocity*,
ulong *time***)**

Note On

virtual void **NoteOn(**uchar *channel*,
uchar *note*,
uchar *velocity*,
ulong *time* = B_NOW**)**

protected:

void **SprayNoteOn(**uchar *channel*,
uchar *note*,
uchar *velocity*,
ulong *time***)**

Pitch Bend

virtual void **PitchBend(**uchar *channel*,
uchar *lsb*,
uchar *msb*,
ulong *time* = B_NOW**)**

protected:

void **SprayPitchBend(**uchar *channel*,
uchar *lsb*,
uchar *msb*,
ulong *time***)**

Program Change

virtual void **ProgramChange(**uchar *channel*,
uchar *programNumber*,
ulong *time* = B_NOW**)**

protected:

void **SprayProgramChange(**uchar *channel*,

uchar *programNumber*,
ulong *time***)**

System Common

virtual void **SystemCommon(**uchar *status*,
uchar *data1*,
uchar *data2*,
ulong *time* = B_NOW**)**

protected:

void **SpraySystemCommon(**uchar *status*,
uchar *data1*,
uchar *data2*,
ulong *time***)**

System Exclusive

virtual void **SystemExclusive(**void **data*,
long *dataLength*,
ulong *time* = B_NOW**)**

protected:

void **SpraySystemExclusive(**void **data*,
long *dataLength*,
ulong *time***)**

SystemRealTime()

virtual void **SystemRealTime(**uchar *status*, ulong *time* = B_NOW**)**

protected:

void **SpraySystemRealTime(**uchar *status*, ulong *time***)**

Tempo Change()

virtual void **TempoChange(**long *beatsPerMinute*, ulong *time* = B_NOW**)**

protected:

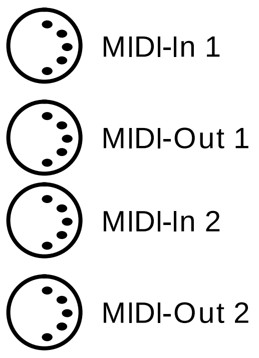
void **SprayTempoChange(**long *beatsPerMinute*, ulong *time***)**aa

BMidiPort

Derived from:	public BObject
Declared in:	<midi/MidiPort.h>

Overview

The BMidiPort class provides the mechanisms for reading MIDI data from the MIDI-In ports, and for writing MIDI data to the MIDI-Out ports. The BeBox has two pairs of MIDI-In and MIDI-Out hardware ports, stacked vertically on the back panel:



You can use a single BMidiPort object to communicate with both halves (the input side and the output side) of a single in/out pair. Thus, to talk to all four ports, you only need two BMidiPort objects. However, you can create and use any number of BMidiPort objects in your application—multiple BMidiPort objects can open and use the same hardware ports at the same time.

Opening the Ports

To obtain data from a MIDI-In port or send data to a MIDI-Out port, you must first open the ports by calling BMidiPort's **Open()** functions. The function's single argument is a string that names the identifies the in/out pair that you're opening. The two pairs of MIDI ports are named "midi1" and "midi2". For example, to open the MIDI-In 1 and MIDI-Out 1 pair, you invoke **Open()** thus:

```
BMidiPort *m_port = new BMidiPort();
m_port->Open("midi1");
```

When you're finished with the ports, you can close them through the **Close()** function. The ports are closed automatically when the BMidiPort object is destroyed.

Run() and the Input Functions

According to the BMidi rules, a BMidi-derived class implementation of **Run()** should create and spray MIDI messages. Furthermore, the implementations of the input functions should realize the messages they receive.

The BMidiPort implementation of **Run()** produces messages by reading them from the MIDI-In port and spraying them to the connected objects. The input functions send MIDI messages to the MIDI-Out port. Linguistically, this might seem backwards, but it makes sense if you think of a BMidiPort as representing not only the hardware port, but whatever is connected to the port. For example, if you're reading data that's generated by an external synthesizer, the **Run()** function can be thought of as encapsulating the synthesizer itself. From this perspective, the message-generation description of **Run()** is reasonable. Similarly, the input functions fulfill their message-realization promise when you consider them to be (for example) the synthesizer that's connected to the MIDI-Out port.

Looping through a BMidiPort Object

It's possible to use the same BMidiPort object to accept data from MIDI-In and broadcast different data to MIDI-Out. You can even connect a BMidiPort object to itself to create a "MIDI through" effect: Anything that shows up at the MIDI-In port will automatically be sent out the MIDI-Out port.

Constructor and Destructor

BMidiPort()

BMidiPort(void**)**

Connects the object to the MIDI-In and MIDI-Out ports. The MIDI-Out connection is active from the moment the object is constructed Messages that arrive through the input functions are automatically sent to the MIDI-Out port. To begin reading from the MIDI-In port, you have to invoke the object's **Start()** function.

~BMidiPort()

virtual **~BMidiPort(**void**)**

Closes the connections to the MIDI ports.

Member Functions

AllNotesOff()

bool **AllNotesOff(**bool *controlOnly*, ulong *time* = B_NOW**)**

Commands the BMidiPort object to issue an All Notes Off MIDI message to the MIDI-Out port. If *controlOnly* is **TRUE**, only the All Notes Off message is sent. If it's **FALSE**, a Note Off message is also sent for every key number on every channel.

Close()

void **Close(**void**)**

Closes the object's MIDI ports. The ports should have been previously opened through a call to **Open()**.

Open()

long **Open(**const char **name***)**

Opens a pair of MIDI ports, as identified by *name*, so the object can read and write MIDI data. The names that correspond to the two set of MIDI ports are “midi1” and “midi2”. The object isn't given exclusive access to the ports that it has opened—other BMidiPort objects, potentially from other applications, can open the same MIDI ports. When you're finished with the ports, you should close them through a (single) call to **Close().**

The function returns **B_NO_ERROR** if the ports were successfully opened.

BMidiStore

Derived from: public BMidi

Declared in: <midi/MidiStore.h>

Overview

The BMidiStore class defines a MIDI recording and playback mechanism. The MIDI messages that a BMidiStore object receives (at its input) are stored as *events* in an *event list*, allowing a captured performance to be played back later. The object can also read and write—or *import* and *export*—standard MIDI files. Typically, the performance and file techniques are combined: A BMidiStore is often used to capture a performance and then export it to a file, or to import a file and then perform it.

Recording

The ability to record a MIDI performance is vested in BMidiStore's input functions (**NoteOn()**, **NoteOff()**, and so on, as declared by the BMidi class). When a BMidiStore input function is invoked, the function fabricates a discrete event based on the data it has received in its arguments, and adds the event to its event list. The event list, in a manner of speaking, *is* the recording.

Since the ability to record is provided by the input functions, you don't need to tell a BMidiStore to start recording; it can record from the moment it's constructed.

For example, to record a performance from an external MIDI keyboard, you connect a BMidiStore to a BMidiPort object and then tell the BMidiPort to start:

```
/* Record a keyboard performance. */
BMidiStore *MyStore = new BMidiStore();
BMidiPort *MyPort = new BMidiPort();

MyPort->Connect(MyStore);
MyPort->Start();
/* Start playing... */
```

At the end of the performance, you tell the BMidiPort to stop:

```
MyPort->Stop();
```

Timestamps

Events are added to a BMidiStore's event list immediately upon arrival. Each event is given a timestamp as it arrives; the value of the timestamp is the value of the *time* argument that was passed to the input function by the "upstream" object's spray function. For example, the time argument that a BMidiPort object passes through its spray functions is always **B_NOW.** Since **B_NOW** is a shorthand for "the current tick," and since time tends to move forward at a reasonably steady rate (at least so far), the events that are recorded from a BMidiPort are guaranteed to be in chronological order (as they appear in the event list).

There's no guarantee that other spraying objects will generate *time* arguments that procede in chronological order, however. And the BMidiStore object doesn't time-sort its events as they arrive; thus, after a recording has been made, events in the event list might not be in chronological order. If you want to ensure that the events are properly ordered, you should call **Sort()** after you've added events to the event list.

Note that BMidiStore's input functions don't call **SnoozeUntil()**: A BMidiStore writes to its event list as soon as it gets a new message, it doesn't wait until the time indicated by the *time* argument.

Erasing and Editing a Recording

You can't. If you make a mistake while you're recording (for example) and want to try again, you can simulate emptying the object by disconnecting the input to the BMidiStore, destroying the object, making a new one, and re-connecting. For example:

```
MyPort->Disconnect(MyStore);
delete MyStore;
MyStore = new BMidiStore();
MyPort->Connect(MyStore);
```

Editing the events in the event list is less than impossible (were such a state possible). You can't do it, and you can't simulate it, at least not with the default implementation of BMidiStore. If you want to edit MIDI data, you have to provide your own BMidi-derived class.

Playback

To "play" a BMidiStore's list of events, you call the object's **Start()** function. For example, by reversing the roles taken by the BMidiStore and BMidiPort objects, you can send the BMidiStore's recording to an external synthesizer:

```
/* First we disconnect the objects. */
MyPort->Disconnect(MyStore);

/* Now connect in the other direction...*/
MyStore->Connect(MyPort);

/* ...and start the playback. */
MyStore->Start();
```

As described in the BMidi class specification, **Start()** invokes **Run()**. In BMidiStore's implementation of **Run()**, the function reads events in the order that they appear in the event list, and sprays the appropriate messages to the connected objects. You can interrupt a BMidiStore playback by calling **Stop()**; uninterrupted, the object will stop by itself after it has sprayed the last event in the list.

The events' timestamps are used as the *time* arguments in the spray functions that are called from within **Run()**. But with a twist: The *time* argument that's passed in the first spray call (for a given performance) is always **B_NOW**; subsequent *time* arguments are re-computed to maintain the correct timing in relation to the first event. In other words, when you tell a BMidiStore to start playing, the first event is performed immediately regardless of the actual value of its timestamp.

Setting the Current Event

A playback needn't begin with the first event in the event list. You can tell the BMidiStore to start somewhere in the middle of the list by calling **SetCurrentEvent()** before starting the playback. The function takes an integer argument that gives the index of the event that you want to begin with.

If you want to start playing from a particular time offset into the event list, you first have to figure out which event lies at that time. To do this, you ask for the event that occurs at or after the time offset (in milliseconds) through the **EventAtDelta()** function. The value that's returned by this function is suitable as the argument to **SetCurrentEvent()**. Here, we prime a playback to begin three seconds into the event list:

```
long firstEvent = MyStore->EventAtDelta(3000);
MyStore->SetCurrentEvent(firstEvent);
```

Keep in mind that **EventAtDelta()** returns the index of the first event at *or after* the desired offset. If you need to know the actual offset of the winning event, you can pass its index to **DeltaOfEvent()**:

```
long firstEvent = MyStore->EventAtDelta(3000);
long actualDelta = MyStore->DeltaOfEvent(firstEvent);
```

Reading and Writing MIDI Files

You can also add events to a BMidiStore's event list by reading, or *importing*, a Standard MIDI File. To do this, you locate the file that you want to read, create a BFile to represent it, and pass the object to the **Import()** function:

```
BFile midi_file;

/* We'll assume that a_dir is a legitimate directory.  */
if (a_dir.Contains("myfile.mid"))
{
    /* Get the file...*/
    a_dir.GetFile("myfile.mid", &midi_file);

    /* ...and import it. */
    MyStore->Import(&midi_file);
}
```

Note that the BFile object isn't open (you shouldn't call BFile's **Open()** function before you call **Import()**).

You can import any number of files into the same BMidiStore object. After you import a file, the event list is automatically sorted.

One thing you shouldn't do is import a MIDI file into a BMidiStore that contains events that were previously recorded from a BMidiPort (in an attempt to mix the file and the recording). Nor does the reverse work: You can't import a file and *then* record from a BMidiPort. The file's timestamps are incompatible with those that are generated for events that are received from the BMidiPort; the result certainly won't be satisfactory.

To write the event list as a MIDI file, you call BMidiStore's **Export()** function:

```
BFile midi_file;

/* We'll assume that a_dir is a legitimate directory. The
 * file should be empty, so we delete it first if it exists.
 */
if (a_dir.Contains("myfile.mid"))
{
    a_dir.GetFile("myfile.mid", &midi_file);
    a_dir.Remove(&midi_file);
}

/* Create the file. */
a_dir.Create(&midi_file);

/* And export the BMidiStore. */
MyStore->Export(&midi_file, 1);
```

Export()'s second argument is an integer that declares the format of the file. The MIDI specification provides three formats: 0, 1, and 2. As with **Import()**, the BFile mustn't be open.

Constructor and Destructor

BMidiStore()

BMidiStore(void**)**

Creates a new, empty BMidiStore object.

~BMidiText()

virtual **~BMidiStore(**void**)**

Frees the memory that the object allocated to store its events.

Member Functions

BeginTime()

inline ulong **BeginTime(**void**)**

Returns the time, in ticks, at which the most recent performance started. This function is only valid if the object has actually performed.

CountEvents()

inline ulong **CountEvents(**void**)**

Returns the number of events in the object's event list.

CurrentEvent()

inline ulong **CurrentEvent(**void**)**

Returns the index of the event that will be performed next.

See also: **SetCurrentEvent()**

DeltaOfEvent()

ulong **DeltaOfEvent(**ulong *index***)**

Returns the "delta time" of the *index*'th event in the object's list of events. An event's delta time is the time span, in ticks, between the first event in the event list and itself.

See also: **EventAtDelta()**

EventAtDelta()

ulong **EventAtDelta(**ulong *delta***)**

Returns the index of the event that occurs on or after *delta* ticks from the beginning of the event list.

See also: **DeltaOfEvent()**

Export()

void **Export(**BFile **aFile*, long *format***)**

Writes the object's event list as a standard MIDI file in the designated format. The BFile must be allocated, must refer to an actual file, and its data portion must not be open. The events are time-sorted before they're written.

See also: **Import()**

Import()

void **Import(**BFile **aFile***)**

Reads the standard MIDI file from the BFile given by the argument. The BFile must not be open.

See also: **Export()**

SetCurrentEvent()

void **SetCurrentEvent(**ulong *index***)**

Sets the object's "current event"—the event that it will perform next—to the event at *index* in the event list.

See also: **CurrentEvent()**

SetTempo()

void **SetTempo(**ulong *beatsPerMinute***)**

Sets the object's tempo—the speed at which it performs events—to *beatsPerMinute*. The default tempo is 60 beats-per-minute.

See also: **Tempo()**

SortEvents()

void **SortEvents(**bool *force* = FALSE**)**

Time-sorts the events in the BMidiStore. The object maintains a (conservative) notion of whether the events are already sorted; if *force* is **FALSE** (the default) and the object doesn't think the operation is necessary, the sorting isn't performed. If force is **TRUE**, the operation is always performed, regardless of its necessity.

Tempo()

ulong **Tempo(**void**)**

Returns the object's tempo in beats-per-minute.

See also: **SetTempo()**

BMidiText

Derived from:	public BMidi
Declared in:	<midi/MidiText.h>

Overview

A BMidiText object displays, to standard output, a textual description of each MIDI message it receives. You use BMidiText objects to debug and monitor your application; it has no other purpose.

To use a BMidiText object, you construct it and connect it to some other BMidi object as shown below:

```
BMidiText *midiText;

midiText = new BMidiText();
otherMidiObj->Connect(midiText);

/* Start a performance here ... */
```

BMidiText's output (the text it displays) is timed: When it receives a MIDI message that's timestamped for the future, the object waits until that time has come to display its textual representation of the message. While it's waiting, the object won't process any other incoming messages. Because of this, you shouldn't connect the same BMidiText object to more than one BMidi object. To monitor two or more MIDI-producing objects, you should connect a separate BMidiText object to each.

The text that's displayed by a BMidiText follows this general format:

timestamp: *MESSAGE TYPE*; *message data*

(Message-specific formats are given in the function descriptions, below.) Of particular note is the *timestamp* field. Its value is the number of milliseconds that have elapsed since the object received its first message. The time value is computed through the use of an internal timer; to reset this timer—a useful thing to do between performances, for example—you call the **ResetTimer()** function.

The BMidiText class doesn't generate or spray MIDI messages, so the performance and connection functions that it inherits from BMidi have no effect.

Constructor and Destructor

BMidiText()

BMidiText(void**)**

Creates a new BMidiText object. The object's timer is set to zero and doesn't start ticking until the first message is received. (To force the timer to start, call **ResetTimer(TRUE)**.)

~BMidiText()

virtual **~BMidiText(**void**)**

Does nothing.

Member Functions

ChannelPressure()

```
virtual void ChannelPressure(char channel,
                             char pressure,
                             ulong time = B_NOW)
```

Responds to a Channel Pressure message by printing the following:

timestamp: CHANNEL PRESSURE; channel = *channel*, pressure = *pressure*

The *channel* and *pressure* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ControlChange()

```
virtual void ControlChange(char channel,
                           char ctrl_num,
                           char ctrl_value,
                           ulong time = B_NOW)
```

Responds to a Control Change message by printing the following:

timestamp: CONTROL CHANGE; channel = *channel*, control = *ctrl_num*, value = *ctrl_value*

The *channel*, *ctrl_num*, and *ctrl_value* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

KeyPressure()

```
virtual void KeyPressure(char channel,
                         char note,
                         char pressure,
                         ulong time = B_NOW)
```

Responds to a Key Pressure message by printing the following:

timestamp: KEY PRESSURE; channel = *channel*, note = *note*, pressure = *pressure*

The *channel*, *note*, and *pressure* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

NoteOff()

```
virtual void NoteOff(char channel,
                     char note,
                     char velocity,
                     ulong time = B_NOW)
```

Responds to a Note Off message by printing the following:

timestamp: NOTE OFF; channel = *channel*, note = *note*, velocity = *velocity*

The *channel*, *note*, and *velocity* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

NoteOn()

```
virtual void NoteOn(char channel,
                    char note,
                    char velocity,
                    ulong time = B_NOW)
```

Responds to a Note On message by printing the following:

timestamp: NOTE ON; channel = *channel*, note = *note*, velocity = *velocity*

The *channel*, *note*, and *velocity* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

PitchBend()

virtual void **PitchBend(**char *channel*,
char *lsb*,
char *msb*,
ulong *time* = B_NOW**)**

Responds to a Pitch Bend message by printing the following:

timestamp: PITCH BEND; channel = *channel*, lsb = *lsb*, msb = *msb*

The *channel*, *lsb*, and *msb* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ProgramChange()

virtual void **ProgramChange(**char *channel*,
char *program_num*,
ulong *time* = B_NOW**)**

Responds to a Program Change message by printing the following:

timestamp: PROGRAM CHANGE; channel = *channel*, program = *program_num*

The *channel* and *program_num* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

ResetTimer()

void **ResetTimer(**bool *start* = FALSE**)**

Sets the object's internal timer to zero. Lacking a *start* argument—or with a *start* of **FALSE**—the timer doesn't start ticking until the next MIDI message is received. If *start* is **TRUE**, the timer begins immediately.

The timer value is used to compute the timestamp that's displayed at the beginning of each message description.

SystemCommon()

virtual void **SystemCommon(**char *status*,
char *data1*,
char *data2*,
ulong *time* = B_NOW**)**

Responds to a System Common message by printing the following:

timestamp: SYSTEM COMMON; status = *status*, data1 = *data1*, data2= *data2*

The *channel*, *data1*, and *data2* values are taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

SystemExclusive()

virtual void **SystemExclusive(**void **data*,
long *data_length*,
ulong *time* = B_NOW**)**

Responds to a System Exclusive message by printing the following:

timestamp: SYSTEM EXCLUSIVE;

This is followed by the data itself, starting on the next line. The data is displayed in hexadecimal, byte by byte. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

SystemRealTime()

virtual void **SystemRealTime(**char *status*, ulong *time* = B_NOW**)**

Responds to a System Real Time message by printing the following:

timestamp: SYSTEM REAL TIME; status = *status*

The *status* value is taken directly from the arguments that are passed to the function. The *timestamp* value is the number of milliseconds that have elapsed since the timer started (see **ResetTimer()** for more information on time).

7 The 3D Kit

The 3D Kit contains classes that let you create, animate, and interact with three-dimensional objects. The Kit is new in Developer Release 8; in this release, a minimum of functionality has been implemented. And an even minimumer amount of technical documentation is currently available. The minimumest, to be precise.

For now, look in the **/boot/develop/projects/Live3d** directory for some example source code that you can play with. This is source code for the **Live3D** application thta you can find in **/boot/apps**. You can also find a 3D Kit white paper on the Be Web site: Point your NetPositive at **<www.be.com/developers/3DWhitePaper.html>**.

Authentic, detailed technical documentation for the 3D Kit will be published on the Be Web site in the coming weeks.

8 The Kernel Kit

8 The Kernel Kit

The Kernel Kit is a collection of C functions that let you define and control the contexts in which your application operates. There are five main topics in the Kit:

- *Threads and Teams.* A thread is a synchronous computer process. By creating multiple threads, you can make your application perform different tasks at (virtually) the same time. A team is the collection of threads that your application creates.

- *Ports*. A port can be thought of as a mailbox for threads: A thread can write a message to a port, and some other thread (or, less usefully, the same thread) can then retrieve the message.

- *Semaphores*. A semaphore is a system-wide counting variable that can be used as a lock that protects a piece of code. Before a thread is allowed to execute the code, it must acquire the semaphore that guards it. Semaphores can also be used to synchronize the execution of two or more threads.

- *Areas*. The area functions let you allocate large chunks of virtual memory. The two primary features of areas are: They can be locked into the CPU's on-chip memory, and the data they hold can be shared between applications.

- *Images*. An image is compiled code that can be dynamically linked into a running application. By loading and unloading images you can make run-time decisions about the resources that your application has access to. Images are of particular interest to driver designers.

The rest of this chapter describes these topics in detail. The final section, "Miscellaneous Functions, Constants, and Defined Types", describes the associated API that support the Kit functions.

Threads and Teams

Declared in: <kernel/OS.h>

Overview

A thread is a synchronous computer process that executes a series of program instructions. Every application has at least one thread: When you launch an application, an initial thread—the *main thread*—is automatically created (or *spawned*) and told to run. The main thread executes the ubiquitous **main()** function, winds through the functions that are called from **main()**, and is automatically deleted (or *killed*) when **main()** exits.

The Be operating system is *multi-threaded*: From the main thread you can spawn and run additional threads; from each of these threads you can spawn and run more threads, and so on. All the threads in all applications run concurrently and asynchronously with each other. Furthermore, threads are independent of each other; most notably, a given thread doesn't own the other threads it has spawned. For example, if thread A spawns thread B, and thread A dies (for whatever reason), thread B will continue to run. (But before you get carried away with the idea of leap-frogging threads, you should take note of the caveat in the section "Death and the Main Thread" on page 11,)

Although threads are independent, they do fall into groups called *teams*. A team consists of a main thread and all other threads that "descend" from it (that are spawned by the main thread directly, or by any thread that was spawned by the main thread, and so on). Viewed from a higher level, a team is the group of threads that are created by a single application. You can't "transfer" threads from one team to another. The team is set when the thread is spawned; it remains the same throughout the thread's life.

All the threads in a particular team share the same address space: Global variables that are declared by one thread will be visible to all other threads in that team.

The following sections describe how to spawn, control, and examine threads and teams.

Spawning a Thread

You spawn a thread by calling the **spawn_thread()** function. The function assigns and returns a system-wide **thread_id** number that you use to identify the new thread in subsequent function calls. Valid **thread_id** numbers are positive integers; you can check the success of a spawn thus:

```
thread_id my_thread = spawn_thread(...);

if ((my_thread) < B_NO_ERROR)
    /* failure */
else
    /* success */
```

The arguments to **spawn_thread()**, which are examined throughout this description, supply information such as what the thread is supposed to do, the urgency of its operation, and so on.

Note: A conceptual neighbor of spawning a thread is the act of loading an executable (or loading an *app image*). This is performed by calling the **load_executable()** function. Loading an executable causes a separate program, identified as a file, to be launched by the system. For more information on the **load_executable()** function, see “Images” beginning on page 55.

Telling a Thread to Run

Spawning a thread isn’t enough to make it run. To tell a thread to start running, you must pass its **thread_id** number to either the **resume_thread()** or **wait_for_thread()** function:

- **resume_thread()** starts the new thread running and immediately returns. The new thread runs concurrently and asynchronously with the thread in which **resume_thread()** was called.
- **wait_for_thread()** starts the thread running but doesn’t return until the thread has finished. (You can also call **wait_for_thread()** on a thread that’s already running.)

Of these two functions, **resume_thread()** is the more common means for starting a thread that was created through **spawn_thread()**. **wait_for_thread()**, on the other hand, is often used to start a thread that was created through **load_executable()**.

The Entry Function

When you call **spawn_thread()**, you must identify the new thread’s *entry function*. This is a global C function (or a static C++ member function) that the new thread will execute when it’s told to run. When the entry function exits, the thread is automatically killed by the operating system.

A thread’s entry function assumes the following protocol:

```
long thread_entry(void *data);
```

The protocol signifies that the function can return a value (to whom the value is returned is a topic that will be explored later), and that it accepts a pointer to a buffer of arbitrarily-typed data. (The function’s name isn’t prescribed by the protocol; in other words, an entry function doesn’t *have* to be named “thread_entry”.)

You specify a thread's entry function by passing a pointer to the function as the first argument to **spawn_thread()**; the last argument to **spawn_thread()** is forwarded as the entry function's *data* argument. Since *data* is delivered as a **void ***, you have to cast the value to the appropriate type within your implementation of the entry function. For example, let's say you define an entry function called **lister()** that takes a pointer to a BList object as an argument:

```
long lister(void *data)
{
    /* Cast the argument. */
    BList *listObj = (BList *)data;
    ...
}
```

To create and run a thread that would execute the **lister()** function, you call **spawn_thread()** and **resume_thread()** thus (excluding error checks):

```
BList *listObj = new BList();
thread_id my_thread;

my_thread = spawn_thread(lister, ..., (void *)listObj);
resume_thread(my_thread);
```

The Entry Function's Argument

The **spawn_thread()** function *doesn't* copy the data that *data* points to. It simply passes the pointer through literally. Because of this, you should never pass a pointer that's allocated locally (on the stack).

The reason for this restriction is that there's no guarantee that the entry function will receive *any* CPU attention before the stack frame from which **spawn_thread()** was called is destroyed. Thus, the entry function won't necessarily have a chance to copy the pointed-to data before the pointer vanishes. There are ways around this restriction—for example, you could use a semaphore to ensure that the entry function has copied the data before the calling frame exits. A better solution is to use the **send_data()** function (which *does* copy its data). See "Passing Data to a Thread" on page 12.

Using a C++ Entry Function

If you're up in C++ territory, you'll probably want to define a class member function that you can use as a thread's entry function. Unfortunately, you can't pass a normal (non-static) member function directly as the entry function argument to **spawn_thread()**—the system won't know which object it's supposed to invoke the function on (it won't have a **this** pointer). To get from here to there, you have to declare two member functions:

- a static member function that is, literally, the entry function,
- and a non-static member function that the static function can invoke. This non-static function will perform the intended work of the entry function.

To "connect" the two functions, you pass an object of the appropriate class (through the *data* argument) to the static function, and then allow the static function to invoke the non-static function upon that object. An example is called for: Here we define a class that contains a static function called **entry_func()**, and a non-static function called **entryFunc()**. By convention, these two are private. In addition, the class declares a public **Go()** function, and a private **thread_id** variable:

```
class MyClass : public BObject {
public:
    long Go(void);

private:
    static long entry_func(void *arg);
    long entryFunc(void);
    thread_id my_thread;
};
```

entry_func() is the literal entry function. It doesn't really do anything—it simply casts its argument as a MyClass object, and then invokes **entryFunc()** on the object:

```
long MyClass::entry_func(void *arg)
{
    MyClass *obj = MyClass *arg;
    return (obj->entryFunc());
}
```

entryFunc() performs the actual work:

```
long MyClass::entryFunc(void)
{
    /* do something here */
    ...
    return (whatever);
}
```

The **Go()** function contains the **spawn_thread()** call that starts the whole thing going:

```
long MyClass::Go(void)
{
    my_thread = spawn_thread(entry_func, ..., this);
    return (resume_thread(my_thread));
}
```

If you aren't familiar with static member functions, you should consult a qualified C++ textbook. Briefly, the only thing you need to know for the purposes of the technique shown here, is that a static function's implementation can't call (non-static) member functions nor can it refer to member data. Maintain the form demonstrated above and you'll be rewarded in heaven.

Entry Function Return Values

The entry function's protocol declares that the function should return a **long** value when it exits. This value can be captured by sitting in a **wait_for_thread()** call until the entry function exits. **wait_for_thread()** takes two arguments: The **thread_id** of the thread that you're waiting for, and a pointer to a **long** into which the value returned by that thread's entry function will be placed. For example:

```
thread_id other_thread;
long result;

other_thread = spawn_thread(...);
resume_thread(other_thread);

...
wait_for_thread(other_thread, &result);
```

If the target thread is already dead, **wait_for_thread()** returns immediately (with an error code as described in the function's full description), and the second argument will be set to an invalid value. If you're late for the train, you'll miss the boat.

Warning: You *must* pass a valid pointer as the second argument to **wait_for_thread()**; you mustn't pass **NULL** even if you're not interested in the return value.

Thread Names

A thread can be given a name which you assign through the second argument to **spawn_thread()**. The name can be 32 characters long (as represented by the **B_OS_NAME_LENGTH** constant) and needn't be unique—more than one thread can have the same name.

You can look for a thread based on its name by passing the name to the **find_thread()** function; the function returns the **thread_id** of the so-named thread. If two or more threads bear the same name, the **find_thread()** function returns the first of these threads that it finds.

You can retrieve the **thread_id** of the calling thread by passing **NULL** to **find_thread()**:

```
thread_id this_thread = find_thread(NULL);
```

To retrieve a thread's name, you must look in the thread's **thread_info** structure. This structure is described in the **get_thread_info()** function description.

Dissatisfied with a thread's name? Use the **rename_thread()** function to change it. Fool your friends.

Thread Priority

In a multi-threaded environment, the CPUs must divide their attention between the candidate threads, executing a few instructions from this thread, then a few from that thread, and so on. But the division of attention isn't always equal: You can assign a higher or lower *priority* to a thread and so declare it to be more or less important than other threads.

You assign a thread's priority (an integer) as the third argument to **spawn_thread()**. There are two categories of priorities:

- "Time-sharing" priorities (priority values from 1 to 99).
- "Real-time" priorities (100 and greater).

A time-sharing thread (a thread with a time-sharing priority value) is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.

A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highest priority is executed first. The thread is allowed to run without being preempted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

The Kernel Kit defines seven priority constants. Although you can use other, "in-between" value as the priority argument to **spawn_thread()**, it's strongly suggested that you stick with these:

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

Synchronizing Threads

There are times when you may want a particular thread to pause at a designated point until some other (known) thread finishes some task. Here are three ways to effect this sort of synchronization:

- The most general means for synchronizing threads is to use a semaphore. The semaphore mechanism is described in great detail in the major section "Semaphores" beginning on page 31.
- Synchronization is sometimes a side-effect of sending data between threads. This is explained in "Passing Data to a Thread" on page 12, and in the major section "Ports" beginning on page 23
- Finally, you can tell a thread to wait for some other thread to die by calling **wait_for_thread()**, as described earlier.

Controlling a Thread

There are three ways to control a thread while it's running:

- You can put a thread to sleep for some number of microseconds through the **snooze()** function. After the thread has been asleep for the requested time, it automatically resumes execution with its next instruction. **snooze()** only works on the calling thread: The function doesn't let you identify an arbitrary thread as the subject of its operation. In other words, whichever thread calls **snooze()** is the thread that's put to sleep.
- You can suspend the execution of any thread through the **suspend_thread()** function. The function takes a single **thread_id** argument that identifies the thread you wish to suspend. The thread remains suspended until you "unsuspend" it through a call to **resume_thread()** or **wait_for_thread()**.
- You can kill the calling thread through **exit_thread()**. The function takes a single (long) argument that's used as the thread's exit status (to make **wait_for_thread()** happy). More generally, you can kill any thread by passing its **thread_id** to the **kill_thread()** function. **kill_thread()** *doesn't* let you set the exit status.

  Feeling all tense and irritated? Try killing an entire team of threads: The **kill_team()** function is more than a system call. It's therapy.

Death and the Main Thread

As mentioned earlier, the control that's imposed upon a particular thread isn't visited upon the "children" that have been spawned from that thread. (Recall the "thread A spawns thread B then dies" business near the beginning of this overview.) However, the death of an application's main thread can affect the other threads:

> ***When a main thread dies, it takes the team's heap, its statically allocated objects, and other team-wide resources—such as access to standard IO—with it. This may seriously cripple any threads that linger beyond the death of the main thread.***

It's certainly possible to create an application in which the main thread sets up one or more other threads, gets them running, and then dies. But such applications should be rare. In general, you should try to keep your main thread around until all other threads in the team are dead.

Passing Data to a Thread

There are three ways to pass data to a thread:

- Through the argument to the entry function, as described in "The Entry Function's Argument" on page 7.
- By using a port or, at a higher level, by sending a BMessage. Ports are described in the next major section ("Ports"); BMessages are part of the Application Kit.
- By sending data to the thread's message cache through the **send_data()** and **receive_data()** functions, as described below.

The **send_data()** function sends data from one thread to another. With each **send_data()** call, you can send two packets of information:

- a single four-byte value (this is called the *code*),
- and an arbitrarily long buffer of arbitrarily-typed data.

The function's four arguments identify, in order,

- the thread that you want to send the data to,
- the four-byte code,
- a pointer to the buffer of data (a **void ***),
- and the size of the buffer of data, in bytes.

In the following example, the main thread spawns a thread, sends it some data, and then tells the thread to run:

```
main(int argc, char *argv[])
{
    thread_id other_thread;
    long code = 63;
    char *buf = "Hello";

    other_thread = spawn_thread(entry_func, ...);
    send_data(other_thread, code, (void *)buf, strlen(buf));
    resume_thread(other_thread);
    ...
}
```

The **send_data()** call copies the code and the buffer (the second and third arguments) into the target thread's message cache and then (usually) returns immediately. In some cases, the four-byte code is all you need to send; in such cases, the buffer pointer can be **NULL** and the buffer size set to 0.

To retrieve the data that's been sent to it, the target thread (having been told to run) calls **receive_data()**. This function returns the four-byte code directly, and copies the data from the message cache into its second argument. It also returns, by reference in its first argument, the **thread_id** of the thread that sent the data:

```
long entry_func(void *data)
{
    thread_id sender;
    long code;
    char buf[512];

    code = receive_data(&sender, (void *)buf, sizeof(buf));
    ...
}
```

Keep in mind that the message data is *copied* into the second argument; you must allocate adequate storage for the data, and pass, as the final argument to **receive_data()**, the size of the buffer that you allocated. A slightly annoying aspect of this mechanism is that there isn't any way for the data-receiving thread to determine how much data is in the message cache, so it can't know, before it receives the data, what an "adequate" size for its buffer is. If the buffer isn't big enough to accommodate all the data, the left-over portion is simply thrown away. (But at least you don't get a segmentation fault.)

As shown in the example, **send_data()** is called before the target thread is running. This feature of the system is essential in situations where you want the target thread to receive some data as its first act (as demonstrated above). However, **send_data()** isn't limited to this use—you can also send data to a thread that's already running.

Blocking when Sending and Receiving

A thread's message cache isn't a queue; it can only hold one message at a time. If you call **send_data()** twice with the same target thread, the second call will block until the target reads the first transmission through a call to **receive_data().** Analogously, **receive_data()** will block if there isn't (yet) any data to receive.

If you want to make sure that you won't block when receiving data, you should call **has_data()** before calling **receive_data()**. **has_data()** takes a **thread_id** argument, and returns **TRUE** if that thread has a message waiting to be read:

```
if (has_data(find_thread(NULL)))
    code = receive_data(...);
```

You can also use **has_data()** to query the target thread before sending it data. This, you hope, will ensure that the **send_data()** call won't block:

```
if (!has_data(target_thread))
    send_data(target_thread, ...);
```

This usually works, but be aware that there's a race condition between the **has_data()** and **send_data()** calls. If yet another thread sends a message to the same target in that time interval, your **send_data()** (might) block.

Functions

exit_thread(), kill_thread(), kill_team()

void **exit_thread(**long *return_value***)**

long **kill_thread(**thread_id *thread***)**

long **kill_team(**team_id *team***)**

These functions command one or more threads to halt execution:

- **exit_thread()** tells the calling thread to exit with a return value as given by the argument. Declaring the return value is only useful if some other thread is sitting in a **wait_for_thread()** call on this thread.
- **kill_thread()** kills the thread given by the argument. The value that the thread will return to **wait_for_thread()** is undefined and can't be relied upon.
- **kill_team()** kills all the threads within the given team. Again, the threads' return values are random.

Exiting a thread is a fairly safe thing to do—since a thread can only exit itself, it's assumed that the thread knows what it's doing. Killing some other thread or an entire team is a bit more drastic since the death certificate(s) will be delivered at an indeterminate time. Nonetheless, in every case (exiting or killing) the system reclaims the resources that the thread (or team) had claimed. So executing a thread shouldn't cause a memory leak.

Keep in mind that threads die automatically (and their resources are reclaimed) if allowed to exit naturally from their entry functions. You should only need to kill a thread if something has gone screwy.

The kill functions return **B_BAD_THREAD_ID** or **B_BAD_TEAM_ID** if the argument is invalid. Otherwise, they return **B_NO_ERROR**.

find_thread()

thread_id **find_thread(**const char **name***)**

Finds and returns the thread with the given name. A *name* argument of **NULL** returns the calling thread. If *name* doesn't identify a thread, **B_NAME_NOT_FOUND** is returned.

A thread's name is assigned when the thread is spawned. The name can be changed thereafter through the **rename_thread()** function. Keep in mind that thread names needn't be unique: If two (or more) threads boast the same name, a **find_thread()** call on that name returns the first so-named thread that it finds.

get_team_info(), get_nth_team_info()

long **get_team_info(**team_id *team*, team_info **info***)**
long **get_nth_team_info(**long *n*, team_info **info***)**

These functions copy, into the *info* argument, the **team_info** structure for a particular team:

- The **get_team_info()** function retrieves information for the team identified by *team*.
- The **get_nth_team_info()** function retrieves team information for the *n*'th team (zero-based) of all teams currently running on your computer. By calling this function with a monotonically increasing *n* value, you can retrieve information for all teams. When, in this scheme, the function no longer returns **B_NO_ERROR**, all teams will have been visited.

The **team_info** structure is defined as:

```
typedef struct {
        team_id team;
        long thread_count;
        long image_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
        long argc;
        char args[64];
} team_info
```

The first field is obvious; the next three reasonably so: They give the number of threads that have been spawned, images that have been loaded, and areas that have been created or cloned within this team.

The debugger fields are used by the, uhm, the...debugger?

The **argc** field is the number of command line arguments that were used to launch the team; **args** is a copy of the first 64 characters from the command line invocation. If this team is an application that was launched through the user interface (by double-clicking, or by accepting a dropped icon), then **argc** is 1 and **args** is the name of the application's executable file.

Both functions return **B_NO_ERROR** upon success. If the designated team isn't found—because *team* in **get_team_info()** isn't valid, or *n* in **get_nth_team_info()** is out-of-bounds—the functions return **BAD_TEAM_ID**.

get_thread_info(), get_nth_thread_info()

long **get_thread_info(**thread_id *thread*, thread_info **info***)**
long **get_nth_thread_info(**team_id *team*, long *n*, thread_info **info***)**

These functions copy, into the *info* argument, the **thread_info** structure for a particular thread:

- The **get_thread_info()** function gets this information for the thread identified by *thread*.
- The **get_nth_thread_info()** function retrieves thread information for the *n*'th thread (zero-based) within the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the threads in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate threads will have been visited.

The **thread_info** structure is defined as:

```
typedef struct {
        thread_id thread;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        thread_state state;
        long priority;
        sem_id sem;
        double user_time;
        double kernel_time;
        void *stack_base;
        void *stack_end;
} thread_info
```

The fields in the structure are:

- **thread**. The **thread_id** number of the thread.
- **team**. The **team_id** of the thread's team.
- **name**. The name assigned to the thread.
- **state**. What the thread is currently doing (see the thread state constants, below).
- **priority**. The level of attention the thread gets (see the priority constants, below).
- **sem**. If the thread is waiting to acquire a semaphore, this is that semaphore.
- **user_time**. The time, in microseconds, the thread has spent executing user code.
- **kernel_time**. The amount of time the kernel has run on the thread's behalf.
- **stack_base**. A pointer to the first byte in the thread's execution stack.
- **stack_end**. A pointer to the last byte in the thread's execution stack.

The last two fields are only meaningful if you understand the execution stack format. Keep in mind that the stack grows down, from higher to lower addresses. Thus, **stack_base** will always be greater than **stack_end**.

The value of the **state** field is one of following **thread_state** constants:

Constant	Meaning
B_THREAD_RUNNING	The thread is currently receiving attention from a CPU.
B_THREAD_READY	The thread is waiting for its turn to receive attention.
B_THREAD_SUSPENDED	The thread has been suspended or is freshly-spawned and is waiting to start.
B_THREAD_WAITING	The thread is waiting to acquire a semaphore. (Note that when a thread is sitting in a **wait_for_thread()** call, or is waiting to read from or write to a port, it's actually waiting to acquire a semaphore.) When in this state, the **sem** field of the **thread_info** structure is set to the **sem_id** number of the semaphore the thread is attempting to acquire.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** function call.
B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.

The value of the priority field takes one of the following **long** constants (the difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

Thread info is provided primarily as a debugging aid. None of the values that you find in a **thread_info** structure are guaranteed to be valid—the thread's state, for example, will almost certainly have changed by the time **get_thread_info()** returns.

Both functions return **B_NO_ERROR** upon success. If the designated thread isn't found—because *thread* in **get_thread_info()** isn't valid, or *n* in **get_nth_thread_info()** is out of range—the functions return **B_BAD_THREAD_ID**. If its *team* argument is invalid, **get_nth_thread_info()** return **B_BAD_TEAM_ID**.

See also: **get_team_info()**

has_data()

bool **has_data(**thread_id *thread***)**

Returns **TRUE** if the given thread has an unread message in its message cache, otherwise returns **FALSE**. Messages are sent to a thread's message cache through the **send_data()** call. To retrieve a message, you call **receive_data()**.

See also: **send_data()**, **receive_data()**

kill_team() *see* exit_thread()

kill_thread() *see* exit_thread()

receive_data()

```
long receive_data(thread_id *sender,
                  void *buffer,
                  long buffer_size)
```

Retrieves a message from the thread's message cache. The message will have been placed there through a previous **send_data()** function call. If the cache is empty, **receive_data()** blocks until one shows up—it never returns empty-handed.

The **thread_id** of the thread that called **send_data()** is returned by reference in the *sender* argument. Note that there's no guarantee that the sender will still be alive by the time you get its ID. Also, the value of *sender* going into the function is ignored—you can't ask for a message from a particular sender.

The **send_data()** function copies two pieces of data into a thread's message cache: A single four-byte code, and a arbitrarily long data buffer. The four-byte code is delivered, here, as **receive_data()**'s return value. The contents of the buffer part of the cache is copied into **receive_data()**'s *buffer* argument (you must allocate and free *buffer* yourself). The *buffer_size* argument tells the function how many bytes of data to copy. If you don't need the data buffer—if the code value returned directly by the function is sufficient—you set *buffer* to **NULL** and *buffer_size* to 0.

Unfortunately, there's no way to tell how much data is in the cache before you call **receive_data()**. If there's more data than *buffer* can accommodate, the unaccommodated portion is discarded—a second **receive_data()** call will not read the rest of the message. Conversely, if **receive_data()** asks for more data than was sent, the function returns with the excess portion of *buffer* unmodified—**receive_data()** doesn't wait for another **send_data()** call to provide more data with which to fill up the buffer.

Each **receive_data()** corresponds to exactly one **send_data()**. Lacking a previous invocation of its mate, **receive_data()** will block until **send_data()** is called. If you don't want to block, you should call **has_data()** before calling **receive_data()** (and proceed to **receive_data()** only if **has_data()** returns **TRUE**).

See also: **send_data()**, **has_data()**

rename_thread()

long **rename_thread(**thread_id *thread***,** const char **name***)**

Changes the name of the given thread to *name*. Keep in mind that the maximum length of a thread name is **B_OS_NAME_LENGTH** (32 characters).

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. Otherwise, the function returns **B_NO_ERROR**.

resume_thread()

long **resume_thread(**thread_id *thread***)**

Tells a new or suspended thread to begin executing instructions. If the thread has just been spawned, its execution begins with the entry-point function (keep in mind that a freshly spawned thread doesn't run until told to do so through this function). If the thread was previously suspended (through **suspend_thread()**), it continues from where it was suspended.

This function only works on threads that have a status of **B_THREAD_SUSPENDED** (newly spawned threads are born with this state). You can't use this function to wake up a sleeping thread (**B_THREAD_ASLEEP**), or to unblock a thread that's waiting to acquire a semaphore (**B_THREAD_WAITING**) or waiting in a **receive_data()** call (**B_THREAD_RECEIVING**).

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. If the thread exists but isn't suspended, **B_BAD_THREAD_STATE** is returned (the target thread is unaffected in this case). Otherwise, the function returns **B_NO_ERROR**.

See also: **wait_for_thread()**

send_data()

```
long send_data(thread_id thread,
               long code,
               void *buffer,
               long buffer_size)
```

Copies data into *thread*'s message cache. The target thread can then retrieve the data from the cache by calling **receive_data()**. There are two parts to the data that you send:

- A single four-byte "code" given by the *code* argument.

- An arbitrarily long buffer of data that's pointed to by *buffer*. The length of the buffer, in bytes, is given by *buffer_size*.

If you only need to send the code, you should set *buffer* to **NULL** and *buffer_size* to 0. After **send_data()** returns you can free the *buffer* argument

Normally, **send_data()** returns immediately—it doesn't wait for the target to call **receive_data()**. However, **send_data()** will block if the target has an unread message from a previous **send_data()**—keep in mind that a thread's message cache is only one message deep. A thread that's blocked in **send_data()** assumes **B_THREAD_WAITING** status.

If the target thread couldn't allocate enough memory for its copy of *buffer*, this function fails and returns **B_NO_MEMORY**. If *thread* doesn't identify a valid thread, **BAD_THREAD_ID** is returned. Otherwise, the function succeeds and returns **B_NO_ERROR**.

See also: **receive_data()**, **has_data()**

set_thread_priority()

long **set_thread_priority(**thread_id *thread*, long *new_priority***)**

Resets the given thread's priority to *new_priority*. The priority level constants that are defined by the Kernel Kit are:

The value of the priority field takes one of the following **long** constants (the difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

The difference between "time-sharing" priorities and "real-time" priorities is explained in "Thread Priority" on page 10.

If *thread* is invalid, **B_BAD_THREAD_ID** is returned. Otherwise, the priority to which the thread was set is returned.

snooze()

long **snooze(**double *microseconds***)**

Pauses the calling thread for the given number of microseconds. The thread's state is set to **B_THREAD_ASLEEP** while it's snoozing and restored to its previous state when it awakes.

The function returns **B_ERROR** if *microseconds* is less than 0.0, otherwise it returns **B_NO_ERROR**. Note that it isn't illegal to put a thread to sleep for 0.0 microseconds, but neither is it effectual; a call of **snooze(**0.0**)** is, essentially, ignored.

spawn_thread()

```
thread_id spawn_thread(thread_entry func,
                       const char *name,
                       long priority,
                       void *data)
```

Creates a new thread and returns its **thread_id** identifier (a positive integer). The arguments are:

- *func* is a pointer to the thread's entry function. This is the function that the thread will execute when it's told to run.
- *name* is the name that you wish to give the thread. It can be, at most, **B_OS_NAME_LENGTH** (32) characters long.
- *priority* is the CPU priority level of the thread. It takes one of the following constant values (listed here from lowest to highest):

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

  For a complete explanation of these constants, see "Thread Priority" on page 10.

- *data* is forwarded as the argument to the thread's entry function.

A newly spawned thread is in a suspended state (**B_THREAD_SUSPENDED**). To tell the thread to run, you pass its **thread_id** to the **resume_thread()** function. The thread will continue to run until the entry-point function exits, or until the thread is explicitly killed (through a call to **exit_thread()**, **kill_thread()**, or **kill_team()**).

If all **thread_id** numbers are currently in use, **spawn_thread()** returns **B_NO_MORE_THREADS**; if the operating system lacks the memory needed to create the thread (which should be rare), **B_NO_MEMORY** is returned.

suspend_thread()

long **suspend_thread(**thread_id *thread***)**

Halts the execution of the given thread, but doesn't kill the thread entirely. The thread remains suspended until it is told to run through the **resume_thread()** function. Nothing prevents you from suspending your own thread, i.e.:

```
suspend_thread(find_thread(NULL));
```

Of course, this is only smart if you have some other thread that will resume you later.

This function only works on threads that have a status of **B_THREAD_RUNNING** or **B_THREAD_READY**. In other words, you can't suspend a thread that's sleeping, waiting to acquire a semaphore, waiting to receive data, or that's already suspended.

If the *thread* argument isn't a valid **thread_id** number, **B_BAD_THREAD_ID** is returned. If the thread exists, but is neither running nor ready to run, **B_BAD_THREAD_STATE** is returned. Otherwise, the function returns **B_NO_ERROR**.

wait_for_thread()

long **wait_for_thread(**thread_id *thread*, long **exit_value***)**

This function causes the calling thread to wait until *thread* (the "target thread") has died. If *thread* is suspended, the **wait_for_thread()** call will cause it to resume. Thus, you can use **wait_for_thread()** to tell a newly-spawned thread to start running.

When the target thread is dead, the value that was returned by its entry function (or that's imposed by **exit_thread()**, if such was called) is returned by reference in *exit_value*. If the target thread was killed (by **kill_thread()** or **kill_team()**), or if the entry function doesn't return a value, the value returned in *exit_value* will be unreliable.

If the target thread has already exited or is otherwise invalid, this function returns **B_BAD_THREAD_ID,** otherwise it returns **B_NO_ERROR.** Note that if the thread is killed while you're waiting for it, the function returns **B_NO_ERROR.**

See also: **resume_thread()**

Ports

Declared in: <kernel/OS.h>

Overview

A port is a system-wide message repository into which a thread can copy a buffer of data, and from which some other thread can then retrieve the buffer. This repository is implemented as a first-in/first-out message queue: A port stores its messages in the order in which they're received, and it relinquishes them in the order in which they're stored. Each port has its own message queue.

There are other ways to send data between threads. Most notably, the data-sending and -receiving mechanism provided by the **send_data()** and **receive_data()** functions can also transmit data between threads. But note these differences between using a port and using the **send_data()**/**receive_data()** functions:

- A port can hold more than one message at a time. A thread can only hold one at a time. Because of this, the function that writes data to a port (**write_port()**) rarely blocks. Sending data to a thread will block if the thread has a previous, unread message.
- The messages that are transmitted through a port aren't directed at a specific recipient—they're not addressed to a specific thread. A message that's been written to a port can be read by any thread. **send_data()**, by definition, has a specific thread as its target.

Creating a Port

A port is represented by a unique, system-wide **port_id** number (a positive integer). The **create_port()** function creates a new port and assigns it a **port_id** number. Although ports are accessible to all threads, the **port_id** numbers aren't disseminated by the operating system; if you create a port and want some other thread to be able to write to or read from it, you have to broadcast the **port_id** number to that thread. Typically, ports are used within a single team. The easiest way to broadcast a **port_id** number to the threads in a team is to declare it as a global variable.

A port is owned by the team in which it was created. When a team dies (when all its threads are killed, by whatever hand), the ports that belong to the team are deleted. A team can bestow ownership of its ports to some other team (through the **set_port_owner()** function).

If you want explicitly get rid of a port, you can call **delete_port()**. You can delete any port, not just those that are owned by the team of the calling thread.

The Message Queue: Reading and Writing Port Messages

The length of a port's message queue—the number of messages that it can hold at a time—is set when the port is created. The **B_MAX_PORT_COUNT** constant provides a reasonable queue length.

The functions **write_port()** and **read_port()** manipulate a port's message queue: **write_port()** places a message at the tail of the port's message queue; **read_port()** removes the message at the head of the queue and returns it the caller. **write_port()** blocks if the queue is full; it returns when room is made in the queue by an invocation of **read_port()**. Similarly, if the queue is empty, **read_port()** blocks until **write_port()** is called. When a thread is waiting in a **write_port()** or **read_port()** call, its state is **B_THREAD_SEM_WAIT** (it's waiting to acquire a system-defined, port-specific semaphore).

You can provide a timeout for your port-writing and port-reading operations by using the "full-blown" functions **write_port_etc()** and **read_port_etc()**. By supplying a timeout, you can ensure that your port operations won't block forever.

Although each port has its own message queue, all ports share a global "queue slot" pool—there are only so many message queue slots that can be used by all ports taken cumulatively. If too many port queues are allowed to fill up, the slot pool will drain, which will cause **write_port()** calls on less-than-full ports to block. To avoid this situation, you should make sure that your **write_port()** and **read_port()** calls are reasonably balanced.

The **write_port()** and **read_port()** functions are the only way to traverse a port's message queue. There's no notion of "peeking" at the queue's unread messages, or of erasing messages that are in the queue.

Port Messages

A port message—the data that's sent through a port—consists of a "message code" and a "message buffer." Either of these elements can be used however you like, but they're intended to fit these purposes:

- The message code (a single four-byte value) should be a mask, flag, or other predictable value that gives a general representation of the flavor or import of the message. For this to work, the sender and receiver of the message must agree on the meanings of the values that the code can take.
- The data in the message buffer can elaborate upon the code, identify the sender of the message, or otherwise supply additional information. The length of the buffer isn't restricted. To get the length of the message buffer that's at the head of a port's queue, you call the **port_buffer_size()** function.

The message that you pass to **write_port()** is copied into the port. After **write_port()** returns, you may free the message data without affecting the copy that the port holds.

When you read a port, you have to supply a buffer into which the port mechanism can copy the message. If the buffer that you supply isn't large enough to accommodate the message, the unread portion will be lost—the next call to **read_port()** won't finish reading the message.

You typically allocate the buffer that you pass to **read_port()** by first calling **port_buffer_size()**, as shown below:

```
char *buf;
long size;
long code;

/* We'll assume that my_port is valid.
 * port_buffer_size() will block until a message shows up.
 */
if ((size = port_buffer_size(my_port) < B_NO_ERROR)
    /* Handle the error */

if (size > 0)
    buf = (char *)malloc(size * sizeof(char));
else
    buf = 0;

/* Now we can read the buffer. */
if (read_port(my_port, &code, (void *)buf, size) < B_NO_ERROR)
    /* Handle the error */
```

Obviously, there's a race condition (in the example) between **port_buffer_size()** and the subsequent **read_port()** call—some other thread could read the port in the interim. If you're going to use **port_buffer_size()** as shown in the example, you shouldn't have more than one thread reading the port.

As stated in the example, **port_buffer_size()** blocks until a message shows up. If you don't want to (potentially) block forever, you should use the **port_buffer_size_etc()** version of the function. As with the other **...etc()** functions, **port_buffer_size_etc()** provides a timeout option.

Function Descriptions

create_port()

port_id **create_port(**long *queue_length*, const char **name***)**

Creates a new port and returns its **port_id** number. The port's name is set to *name* and the length of its message queue is set to *queue_length*. Neither the name nor the queue length can be changed once they're set. The name shouldn't exceed **B_OS_NAME_LENGTH** (32) characters.

In setting the length of a port's message queue, you're telling it how many messages it can hold at a time. When the queue is filled—when it's holding *queue_length* messages—subsequent invocations of **write_port()** (on that port) block until room is made in the queue (through calls to **read_port()**) for the additional messages. As a convenience, you can use the **B_MAX_PORT_COUNT** constant as the *queue_length* value; this constant represents the (ostensible) maximum port queue length. Once the queue length is set (here), it can't be changed.

This function also sets the owner of the port to be the team of the calling thread. Ownership can subsequently be transferred through the **set_port_owner()** function. When a port's owner dies (when all the threads in the team are dead), the port is automatically deleted. If you want to delete a port prior to its owner's death, use the **delete_port()** function.

The function returns **B_BAD_VALUE** if *queue_length* is out of bounds (less than one or greater than the maximum capacity). It returns **B_NO_MORE_PORTS** if all **port_id** numbers are currently being used.

See also: **delete_port()**, **set_port_owner()**

delete_port()

long **delete_port(**port_id *port***)**

Deletes the given port. The port's message queue doesn't have to be empty—you can delete a port that's holding unread messages. Threads that are blocked in **read_port()** or **write_port()** calls on the port are automatically unblocked (and return **B_BAD_SEM_ID**).

The thread that calls **delete_port()** doesn't have to be a member of the team that owns the port; any thread can delete any port.

The function returns **B_BAD_PORT_ID** if *port* isn't a valid port; otherwise it returns **B_NO_ERROR**.

See also: **create_port()**

find_port()

port_id **find_port(**const char **port_name***)**

Returns the **port_id** of the named port. If the argument doesn't name an existing port, **B_NAME_NOT_FOUND** is returned.

See also: **create_port()**

get_port_info(), get_nth_port_info()

long **get_port_info(**port_id *port*, port_info **info***)**
long **get_nth_port_info(**team_id *team*, long *n*, port_info **info***)**

These functions copy, into the *info* argument, the **port_info** structure for a particular port:

- The **get_port_info()** function gets this information for the port identified by *port*.
- The **get_nth_port_info()** function retrieves port information for the *n*'th port (zero-based) that's owned by the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the ports in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate ports will have been visited.

The **port_info** structure is defined as:

```
typedef struct port_info {
        port_id port;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long capacity;
        long queue_count;
        long total_count;
} port_info
```

The structure's fields are:

- **port**. The **port_id** number of the port.
- **team**. The **team_id** of the port's team.
- **name**. The name assigned to the port.
- **capacity**. The length of the port's message queue.
- **queue_count**. The number of messages currently in the queue.
- **total_count**. The total number of message that have been read from the port.

Note that the **total_count** number doesn't include the messages that are currently in the queue.

The information in the **port_info** structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a port as it exists just before the info-retrieving function returns.

The functions return **B_NO_ERROR** if the designated port is successfully found. Otherwise, they return **B_BAD_PORT_ID**, **B_BAD_TEAM_ID**, or **B_BAD_INDEX**.

port_buffer_size(), port_buffer_size_etc()

long **port_buffer_size(**port_id *port***)**

long **port_buffer_size_etc(**port_id *port*, long *flags*, double *timeout***)**

These functions return the length (in bytes) of the message buffer that's at the head of *port*'s message queue. You call this function in order to allocate a sufficiently large buffer in which to retrieve the message data.

The **port_buffer_size()** function blocks if the port is currently empty. It unblocks when a **write_port()** call gives this function a buffer to measure (even if the buffer is 0 bytes long), or when the port is deleted.

The **port_buffer_size_etc()** function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

If *port* doesn't identify an existing port (or if the port is deleted while the function is blocked), **B_BAD_PORT_ID** is returned. If the *timeout* limit is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (and **B_TIMEOUT** is set), and there are no messages in the queue, the function immediately returns **B_WOULD_BLOCK**.

See also: **read_port()**

port_count()

long **port_count(**port_id *port***)**

Returns the number of messages that are currently in *port*'s message queue. This is the number of messages that have been written to the port through calls to **write_port()** but that haven't yet been picked up through corresponding **read_port()** calls. This function is provided mostly as a convenience and a semi-accurate debugging tool. The value that it returns is inherently undependable (there's no guarantee that additional **read_port()** or **write_port()** calls won't change the count as this function is returning).

If *port* isn't a valid port identifier, **B_BAD_PORT_ID** is returned.

See also: **get_port_info()**

read_port(), read_port_etc()

long **read_port(**port_id *port*,
    long **msg_code*,
    void **msg_buffer*,
    long *buffer_size***)**

long **read_port_etc(**port_id *port*,
    long **msg_code*,
    void **msg_buffer*,
    long *buffer_size*,
    long *flags*,
    double *timeout***)**

These functions remove the message at the head of *port*'s message queue and copy the messages's contents into the *msg_code* and *msg_buffer* arguments. The size of the *msg_buffer* buffer, in bytes, is given by *buffer_size*. It's up to the caller to ensure that the message buffer is large enough to accommodate the message that's being read. If you want a hint about the message's size, you should call **port_buffer_size()** before calling this function.

If *port*'s message queue is empty when you call **read_port()**, the function will block. It returns when some other thread writes a message to the port through **write_port()**. A blocked read is also unblocked if the port is deleted.

The **read_port_etc()** function lets you set a limit on the amount of time the function will wait for a message to show up. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

The functions returns **B_BAD_PORT_ID** if *port* isn't valid (this includes the case where the port is deleted during a blocked **read_port()** call). If the *timeout* value is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (with **B_TIMEOUT** set), and there are no messages in the queue, the function immediately returns **B_WOULD_BLOCK**.

A successful call returns the number of bytes that were written into the *msg_buffer* argument.

See also: **write_port()**, **port_buffer_size()**

set_port_owner()

long **set_port_owner(**port_id *port*, team_id *team***)**

Transfers ownership of the designated port to *team*. A port can only be owned by one team at a time; by setting a port's owner, you remove it from its current owner.

There are no restrictions on who can own a port, or on who can transfer ownership. In other words, the thread that calls **set_port_owner()** needn't be part of the team that currently owns the port, nor must you only assign ports to the team that owns the calling thread (although these two are the most likely scenarios).

Port ownership is meaningful for one reason: When a team dies (when all its threads are dead), the ports that are owned by that team are deleted. Ownership, otherwise, has no significance—it carries no special privileges or obligations.

To discover a port's owner, use the **get_port_info()** function.

set_port_owner() fails and returns **B_BAD_PORT_ID** or **B_BAD_TEAM_ID** if one or the other argument is invalid. Otherwise it returns **B_NO_ERROR**.

See also: **get_port_info()**

write_port(), write_port_etc()

long **write_port(**port_id *port,*
long *msg_code,*
void **msg_buffer,*
long *buffer_size***)**

long **write_port_etc(**port_id *port,*
long *msg_code,*
void **msg_buffer,*
long *buffer_size,*
long *flags,*
double *timeout***)**

These functions place a message at the tail of *port*'s message queue. The message consists of *msg_code* and *msg_buffer*:

- *msg_code* holds the message code. This is a mask, flag, or other predictable value that gives a general representation of the message.
- *msg_buffer* is a pointer to a buffer that can be used to supply additional information. You pass the length of the buffer, in bytes, as the value of the *buffer_size* argument. The buffer can be arbitrarily long.

If the port's queue is full when you call **write_port()**, the function will block. It returns when a **read_port()** call frees a slot in the queue for the new message. A blocked **write_port()** will also return if the target port is deleted.

The **write_port_etc()** function lets you set a limit on the amount of time the function will wait for a free queue slot. To set the limit, you pass **B_TIMEOUT** as the flags argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait.

If *port* isn't valid **B_BAD_PORT_ID** is returned (this includes the case where the port is deleted during a blocked **read_port()** call). If the *timeout* value is exceeded, **B_TIMED_OUT** is returned. If the *timeout* limit is 0.0 (with **B_TIMEOUT** set), and the target port's queue is full, the function immediately returns **B_WOULD_BLOCK**. A successful call returns **B_NO_ERROR**.

See also: **read_port()**

Semaphores

Declared in: <kernel/OS.h>

Overview

A semaphore is a token that's used in a multi-threaded operating system to coordinate access, by competing threads, to "protected" resources or operations. This coordination usually takes one of these tacks:

- The most common use of semaphores is to limit the number of threads that can execute a piece of code at the same time. The typical limit is one—in other words, semaphores are most often used to create mutually exclusive locks.
- Semaphores can also be used to impose the order in which a series of interdependent operations are performed.

Examples of these uses are given in sections below.

How Semaphores Work

A semaphore acts as a key that a thread must acquire in order to continue execution. Any thread that can identify a particular semaphore can attempt to acquire it by passing its **sem_id** identifier—a system-wide number that's assigned when the semaphore is created—to the **acquire_sem()** function. The function doesn't return until the semaphore is actually acquired. (An alternate function, **acquire_sem_etc()** lets you specify a limit, in microseconds, on the amount of time you're willing to wait for the semaphore to be acquired. Unless otherwise noted, characteristics ascribed to **acquire_sem()** apply to **acquire_sem_etc()** as well.)

When a thread acquires a semaphore, that semaphore (typically) becomes unavailable for acquisition by other threads (in the rarer case, more than one thread is allowed to acquire the semaphore at a time; the precise determination of availability is explained in "The Thread Count" on page 32). The semaphore remains unavailable until it's passed in a call to the **release_sem()** function.

The code that a semaphore "protects" lies between the calls to **acquire_sem()** and **release_sem()**. The disposition of these functions in your code usually follows this pattern:

```
acquire_sem(my_semaphore);
/* Protected code goes here. */
release_sem(my_semaphore);
```

Keep in mind that these function calls needn't be so explicitly balanced. A semaphore can be acquired within one function and released in another. Acquisition and release of the same semaphore can even be performed by two different threads; an example of this is given in "Using Semaphores to Impose an Execution Order" on page 35.

The Thread Queue

Every semaphore has its own *thread queue*: This is a list that identifies the threads that are waiting to acquire the semaphore. A thread that attempts to acquire an unavailable semaphore is placed at the tail of the semaphore's thread queue; from the programmer's point of view, a thread that's been placed in the queue will be blocked in the **acquire_sem()** call. Each call to **release_sem()** "releases" the thread at the head of that semaphore's queue (if there are any waiting threads), thus allowing the thread to return from its call to **acquire_sem()**.

Semaphores don't discriminate between acquisitive threads—they don't prioritize or otherwise reorder the threads in their queues—the oldest waiting thread is always the next to acquire the semaphore.

The Thread Count

To assess availability, a semaphore looks at its *thread count*. This is a counting variable that's initialized when the semaphore is created. The ostensible (although, as we shall see, not entirely accurate) meaning of a thread count's initial value, which is passed as the first argument to **create_sem()**, is the number of threads that can acquire the semaphore at a time. For example, a semaphore that's used as a mutually exclusive lock takes an initial thread count of 1—in other words, only one thread can acquire the semaphore at a time.

Calls to **acquire_sem()** and **release_sem()** alter the semaphore's thread count: **acquire_sem()** decrements the count, and **release_sem()** increments it. When you call **acquire_sem()**, the function looks at the thread count (before decrementing it) to determine if the semaphore is available:

- If the count is greater than zero, the semaphore is available for acquisition, so the function returns immediately.
- If the count is zero or less, the semaphore is unavailable, and so the thread is placed in the semaphore's thread queue.

The initial thread count isn't an inviolable limit on the number of threads that can acquire a given semaphore—it's simply the initial value for the sempahore's thread count variable. For example, if you create a semaphore with an initial thread count of 1 and then immediately call **release_sem()** five times, the semaphore's thread count will increase to 6. Furthermore, although you can't initialize the thread count to less-than-zero, an initial value of zero itself is common—it's an integral part of using semaphores to impose an execution order (as demonstrated later).

Summarizing the description above, there are three significant thread count value ranges:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **acquire_sem()** calls will return without blocking.
- If the count is 0, there are no queued threads, but the next **acquire_sem()** call will block.
- A negative count (-*n*) means there are *n* threads in the semaphore's thread queue, and the next call to **acquire_sem()** will block.

Although it's possible to retrieve the value of a semaphore's thread count (by looking at a field in the semaphore's **sem_info** structure, as described later), you should only do so for amusement—while you're debugging, for example. You should never predicate your code on the basis of a semaphore's thread count.

Using a Semaphore as a Lock

As mentioned above, the most common use of semaphores is to ensure that only one thread is executing a certain piece of code at a time. The following example demonstrates this use.

Consider an application that manages a one-job-at-a-time device such as a printer. When the application wants to start a new print job (upon a request from some other application, no doubt) it spawns and runs a thread to perform the actual data transmission. Given the nature of the device, each spawned thread must be allowed to complete its transmission before the next thread takes over. However, your application wants to accept print requests (and so spawn threads) as they arrive.

To ensure that the spawned threads don't interrupt each other, you can define a semaphore that's acquired and released—that, in essence, is "locked" and "unlocked"—as a thread begins and ends its transmission, as shown below. The thread functions that are used in the example are described in "Threads and Teams" on page 5.

```
/* Include the semaphore API declarations. */
#include <OS.h>

/* The semaphore is declared globally so the spawned threads
 * will be able to get to it (there are other ways of
 * broadcasting the sem_id, but this is the easiest).
 */
sem_id print_sem;

/* print_something() is the data-transmission function.
 * The data itself would probably be passed as an argument
 * (which isn't shown in this example).
 */
long print_something(void *data);
```

```
main()
{
    /* Create the semaphore with an initial thread count of 1.
     * If the semaphore can't be created (error conditions
     * are listed later), we exit.  The second argument to
     * create_sem(), as explained in the function
     * descriptions is a handy string name for the semaphore.
     */
    if ((print_sem = create_sem(1, "print sem")) < B_NO_ERROR)
        exit -1;

    while (1)
    {
        /* Wait-for-a-request code and break conditions
         * go here.
         */
        ...

        /* Spawn a thread that calls print_something(). */
        if (resume_thread(spawn_thread(print_something ...))
            < B_NO_ERROR)
            break;
    }

    /* Acquire the semaphore and delete it (as explained
     * later)
     */
    acquire_sem(print_sem);
    delete_sem(print_sem);
    exit 0;
}

long print_something(void *data)
{
    /* Acquire the semaphore; an error means the semaphore
     * is no longer valid.  And we'll just die if it's no good.
     */
    if (acquire_sem(print_sem) < B_NO_ERROR)
        return 0;

    /* The code that sends data to the printer goes here. */

    /* Release the semaphore. */
    release_sem(print_sem);

    return 0;
}
```

The **acquire_sem()** and **release_sem()** calls embedded in the **print_something()** function "protect" the data-transmission code. Although any number of threads may concurrently execute **print_something()**, only one at a time is allowed to proceed past the **acquire_sem()** call.

Deleting a Semaphore

Notice that the example explicitly deletes the **print_sem** semaphore before it exits. This isn't wholly necessary: Every semaphore is owned by a team (the team of the thread that called **create_sem()**). When the last thread in a team dies, it takes the team's semaphores with it.

Prior to the death of a team, you can explicitly delete a semaphore through the **delete_sem()** call. Note, however, that **delete_sem()** must be called from a thread that's a member of the team that owns the semaphore—you can't delete another team's semaphores.

You're allowed to delete a semaphore even if it still has threads in its queue. However, you usually want to avoid this, so deleting a semaphore may require some thought. In the example, the main thread (the thread that executes the **main()** function) makes sure all print threads have finished by acquiring the semaphore before deleting it. When the main thread is allowed to continue (when the **acquire_sem()** call returns) the queue is sure to be empty and all print jobs will have completed.

When you delete a semaphore (or when it dies naturally), all its queued threads are immediately allowed to continue—they all return from **acquire_sem()** at once. You can distinguish between a "normal" acquisition and a "semaphore deleted" acquisition by the value that's returned by **acquire_sem()** (the specific return values are listed in the function descriptions, below).

Using Semaphores to Impose an Execution Order

Semaphores can also be used to coordinate threads that are performing separate operations, but that need to perform these operations in a particular order. In the following example, an application repeatedly spawns, in no particular order, threads that either write to or read from a global buffer. Each writing thread must complete before the next reading thread starts, and each written message must be fully read exactly once. Thus, the two operations must alternate (with a writing thread going first). Two semaphores are used to coordinate the threads that perform these operations:

```
/* Here's the global buffer. */
char buf[1024];

/* The ok_to_read and ok_to_write semaphores inform the
 * appropriate threads that they can proceed.
 */
sem_id ok_to_write, ok_to_read;

/* These are the writing and reading functions. */
long write_it(void *data);
long read_it(void *data);
```

```
main()
{
    /* These will be used when we delete the semaphores. */
    long write_count, read_count;

    /* Create the semaphores.  ok_to_write is created with a
     * thread count of 1; ok_to_read's count is set to 0.
     * This is explained below.
     */
    if ((ok_to_write = create_sem(1, "write sem"))<B_NO_ERROR)
        return (B_ERROR);

    if ((ok_to_read = create_sem(0, "read sem")) < B_NO_ERROR)
    {
        delete_sem(ok_to_write);
        return (B_ERROR);
    }

    bzero(buf,1024);

    /* Spawn some reading and writing threads. */
    while(1)
    {
        if ( ... ) /* spawn-a-writer condition */
            resume_thread(spawn_thread(write_it, ...));
        if ( ... ) /* spawn-a-reader condition */
            resume_thread(spawn_thread(read_it, ...);
        if ( ... ) /* break condition */
            break;
    }

    /* It's time to delete the semaphores.  First, get the
     * semaphores' thread counts.
     */
    if (get_sem_count(ok_to_write, &write_count) < B_NO_ERROR)
    {
        delete_sem(ok_to_read);
        return (B_ERROR);
    }

    if (get_sem_count(ok_to_read, &read_count) < B_NO_ERROR)
    {
        delete_sem(ok_to_write);
        return (B_ERROR);
    }

    /* Place this thread at the end of whichever queue is
     * shortest (or the writing queue if they're equal).
     * Remember: thread count is decremented as threads
     * are placed in the queue, so the shorter queue is
     * the one with the greater thread count.
     */
    if (write_count >= read_count)
        acquire_sem(ok_to_write);
        else
            acquire_sem(ok_to_read);

        /* Delete the semaphores and exit. */
        delete_sem(ok_to_write);
        delete_sem(ok_to_read);
        return (B_NO_ERROR);
    }

    long write_it(void *data)
    {
        /* Acquire the writing semaphore. */
        if (acquire_sem(ok_to_write) < B_NO_ERROR)
            return (B_ERROR);

        /* Write to the buffer. */
        strncpy(buf, (char *)data, 1023);

        /* Release the reading semaphore. */
        return (release_sem(ok_to_read));
    }

    long read_it(void *data)
    {
        /* Acquire the reading semaphore. */
        if (acquire_sem(ok_to_read) < B_NO_ERROR)
            return (B_ERROR);

        /* Read the message and do something with it. */
        ...

        /* Release the writing semaphore. */
        return (release_sem(ok_to_write));
    }
```

Notice the distribution of the **acquire_sem()** and **release_sem()** calls for the respective semaphores: The writing function acquires the writing semaphore (*ok_to_write*) and then releases the reading semaphore (*ok_to_read*). The reading function does the opposite. Thus, after the buffer has been written to, no other thread can write to it until it has been read (and vice versa).

By setting *ok_to_write*'s initial thread count to 1 and *ok_to_read*'s initial thread count to 0, you ensure that a writing operation will be performed first. If a reading thread is spawned first, it will block until a writing thread releases the *ok_to_read* semaphore.

When it's semaphore-deletion time in the example, the main thread acquires one of the semaphores. Specifically, it acquires the semaphore that has the fewer threads in its queue. This allows the remaining (balanced) pairs of reading and writing threads to complete before the semaphores are deleted, and throws away any unpaired reading or writing threads. (Actually, the unpaired threads aren't "thrown away" as the semaphore upon which they're waiting is deleted, but by the error check in the first line of the reading or writing function. As mentioned earlier, deleting the semaphore releases its queued threads, allowing them, in this instance, to rush to their deaths.)

Broadcasting Semaphores

The **sem_id** number that identifies a semaphore is a system-wide token—the **sem_id** values that you create in your application will identify your semaphores in all other applications as well. It's possible, therefore, to broadcast the **sem_id** numbers of the semaphores that you create and so allow other applications to acquire and release them—but it's not a very good idea. A semaphore is best controlled if it's created, acquired, released, and deleted within the same team. If you want to provide a protected service or resource to other applications, you should follow the model used by the examples: Your application should accept messages from other applications and then spawn threads that acquire and release the appropriate semaphores.

Functions

acquire_sem(), acquire_sem_etc()

long **acquire_sem(**sem_id *sem***)**
long **acquire_sem_etc(**sem_id *sem***,** long *count*, long *flags*, double *timeout***)**

These functions attempt to acquire the semaphore identified by the *sem* argument. Except in the case of an error, **acquire_sem()** doesn't return until the semaphore has actually been acquired.

acquire_sem_etc() is the full-blown acquisition version: It's essentially the same as **acquire_sem()**, but, in addition, it lets you acquire a semaphore more than once, and also provides a timeout facility:

- The *count* argument lets you specify that you want the semaphore to be acquired *count* times. This means that the semaphore's thread count is decremented by the specified amount. It's illegal to specify a count that's less than 1.

- To enable the timeout, you pass **B_TIMEOUT** as the *flags* argument, and set *timeout* to the amount of time, in microseconds, that you're willing to wait for the semaphore to be acquired. If the semaphore hasn't been acquired within *timeout* microseconds, the function gives up and returns the value **B_TIMED_OUT**. If you specify a *timeout* of 0.0 and the semaphore isn't immediately available, the function returns **B_WOULD_BLOCK**.

In addition to **B_TIMEOUT**, the Kernel Kit defines two other semaphore-acquisition flag constants (**B_CAN_INTERRUPT** and **B_CHECK_PERMISSION**). These additional flags are used by device drivers—adding these flags into a "normal" (or "user-level") acquisition has no effect. However, you should be aware that the **B_CHECK_PERMISSION** flag is always added in to user-level semaphore acquisition in order to protect system-defined semaphores.

Other than the timeout and the acquisition count, there's no difference between the two acquisition functions. Specifically, any semaphore can be acquired through either of these functions; you always release a semaphore through **release_sem()** (or **release_sem_etc()**) regardless of which function you used to acquire it.

To determine if the semaphore is available, the function looks at the semaphore's thread count (before decrementing it):

- If the thread count is positive, the semaphore is available and the current acquisition succeeds. The **acquire_sem()** or **acquire_sem_timeout()** function returns immediately upon acquisition.
- If the thread count is zero or less, the calling thread is placed in the semaphore's thread queue where it waits for a corresponding **release_sem()** call to de-queue it (or for the timeout to expire).

If the *sem* argument doesn't identify a valid semaphore, **B_BAD_SEM_ID** is returned. It's possible for a semaphore to become invalid while an acquisitive thread is waiting in the semaphore's queue. For example, if your thread calls **acquire_sem()** on a valid (but unavailable) semaphore, and then some other thread deletes the semaphore, your thread will return **B_BAD_SEM_ID** from its call to **acquire_sem()**.

If you pass an illegal *count* value (less than 1) to **acquire_sem_etc()**, the function returns **B_BAD_VALUE**. If the acquisition time surpasses the designated timeout limit (with **B_TIMEOUT** set), the **acquire_sem_etc()** function returns **B_TIMED_OUT**; if the timeout value is 0.0 and the semaphore isn't immediately available, the function returns **B_WOULD_BLOCK**.

If the semaphore is successfully acquired, the functions return **B_NO_ERROR**.

See also: **release_sem()**

create_sem()

sem_id **create_sem(**long *thread_count*, const char **name***)**

Creates a new semaphore and returns a system-wide **sem_id** number that identifies it. The arguments are:

- *thread_count* initializes the semaphore's *thread count*, the counting variable that's decremented and incremented as the semaphore is acquired and released (respectively). You can pass any non-negative number as the count, but you typically pass either 1 or 0, as demonstrated in the examples above.
- *name* is an optional string name that you can assign to the semaphore. The name is meant to be used only for debugging. A semaphore's name needn't be unique—any number of semaphores can have the same name.

Valid **sem_id** numbers are positive integers. You should always check the validity of a new semaphore through a construction such as

```
if ((my_sem = create_sem(1,"My Semaphore")) < B_NO_ERROR)
```

```
        /* If it's less than B_NO_ERROR, my_sem is invalid. */
```

create_sem() sets the new semaphore's owner to the team of the calling thread. Ownership may be re-assigned through the **set_sem_owner()** function. When the owner dies (when all the threads in the team are dead), the semaphore is automatically deleted. The owner is also signficant in a **delete_sem()** call: Only those threads that belong to a semaphore's owner are allowed to delete that semaphore.

The function returns one of the following codes if the semaphore couldn't be created:

Return Code	Meaning
B_BAD_ARG_VALUE	Invalid *thread_count* value (less than zero).
B_NO_MEMORY	Not enough memory to allocate the semaphore's name.
B_NO_MORE_SEMS	All valid **sem_id** numbers are being used.

See also: **delete_sem()**

delete_sem()

long **delete_sem(**sem_id *sem***)**

Deletes the semaphore identified by the argument. If there are any threads waiting in the semaphore's thread queue, they're immediately de-queued and allowed to continue execution.

This function may only be called from a thread that belongs to the target semaphore's owner; if the calling thread belongs to a different team, or if *sem* is invalid, the function returns **B_BAD_SEM_ID**. Otherwise, it returns**B_NO_ERROR**.

See also: **acquire_sem(**)

get_sem_count()

long **get_sem_count(**sem_id *sem,* long **thread_count***)**

Returns, by reference in *thread_count*, the value of the semaphore's thread count variable:

- A positive thread count (*n*) means that there are no threads in the semaphore's queue, and the next *n* **acquire_sem()** calls will return without blocking.

- If the count is zero, there are no queued threads, but the next **acquire_sem()** call will block.

- A negative count (-*n*) means there are *n* threads in the semaphore's thread queue and the next call to **acquire_sem()** will block.

By the time this function returns and you get a chance to look at the *thread_count* value, the semaphore's thread count may have changed. Although watching the thread count might help you while you're debugging your program, this function shouldn't be an integral part of the design of your application.

If *sem* is a valid semaphore identifier, the function returns **B_NO_ERROR**; otherwise, **B_BAD_SEM_ID** is returned (and the value of the *thread_count* argument that you pass in isn't changed).

See also: **get_sem_info()**

get_sem_info(), get_nth_sem_info()

long **get_sem_info(**sem_id *sem*, sem_info **info***)**
long **get_nth_sem_info(**team_id *team*, long *n*, sem_info **info***)**

These functions copy, into the *info* argument, the **sem_info** structure for a particular semaphore:

- The **get_sem_info()** function gets this information for the semaphore identified by *sem*.
- The **get_nth_sem_info()** function retrieves semaphore information for the *n*'th semaphore (zero-based) that's owned by the team identified by *team*. If *team* is 0 (zero), all teams are considered. You use this function to retrieve the info structures of all the semaphores in a team (or in all teams) by repeatedly calling the function with a monotonically increasing value of *n*—the actual value of *n* has no other significance. When, in this scheme, the function no longer returns **B_NO_ERROR**, all candidate semaphores will have been visited.

The **sem_info** structure is defined as:

```
typedef struct sem_info {
        sem_id sem;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long count;
        thread_id latest_holder;
} sem_info
```

The structure's fields are:

- **sem**. The **sem_id** number of the semaphore.
- **team**. The **team_id** of the semaphore's owner.
- **name**. The name assigned to the semaphore.
- **count**. The semaphore's thread count.
- **latest_holder**. The thread that most recently acquired the semaphore.

Note that the thread that's identified in the **lastest_holder** field may no longer be holding the semaphore—it may have since released the semaphore. The latest holder is simply the last thread to have called **acquire_sem()** (of whatever flavor) on this semaphore.

The information in the **sem_info** structure is guaranteed to be internally consistent, but the structure as a whole should be consider to be out-of-date as soon as you receive it. It provides a picture of a semaphore as it exists just before the info-retrieving function returns.

The functions return **B_NO_ERROR** if the designated semaphore is successfully found. Otherwise, they return **B_BAD_SEM_ID**, **B_BAD_TEAM_ID**, or **B_BAD_INDEX**.

release_sem(), release_sem_etc()

long **release_sem(**sem_id *sem***)**
long **release_sem_etc(**sem_id *sem*, long *count*, long *flags***)**

The **release_sem()** function de-queues the thread that's waiting at the head of the semaphore's thread queue (if any), and increments the semaphore's thread count. **release_sem_etc()** does the same, but for *count* threads.

Normally, releasing a semaphore automatically invokes the kernel's scheduler. In other words, when your thread calls **release_sem()** (or the sequel), you're pretty much guaranteed that some other thread will be switched in immediately afterwards, even if your thread hasn't gotten its fair share of CPU time. If you want to subvert this automatism, call **release_sem_etc()** with a *flags* value of **B_DO_NOT_RESCHEDULE**. Preventing the automatic rescheduling is particularly useful if you're releasing a number of different semaphores all in a row: By avoiding the rescheduling you can prevent some unnecessary context switching.

If *sem* is a valid semaphore identifier, these functions return **B_NO_ERROR**; if it's invalid, they return **B_BAD_SEM_ID**. Note that if a released thread deletes the semaphore (before the releasing function returns), these functions will still return **B_NO_ERROR**.

The *count* argument to **release_sem_count()** must be greater than zero; the function returns **B_BAD_VALUE** otherwise.

See also: **acquire_sem()**

set_sem_owner()

long **set_sem_owner(**sem_id *sem*, team_id *team***)**

Transfers ownership of the designated semaphore to *team*. A semaphore can only be owned by one team at a time; by setting a semaphore's owner, you remove it from its current owner.

There are no restrictions on who can own a semaphore, or on who can transfer ownership. In practice, however, the only reason you should ever transfer ownership is if you're writing a device driver and you need to bequeath a semaphore to the kernel (the team of which is known, for this purpose, as **B_SYSTEM_TEAM**).

Semaphore ownership is meaningful for two reason: When a team dies (when all its threads are dead), the semaphores that are owned by that team are deleted. Also, only a thread that belongs to a semaphore's owner is allowed to delete that semaphore.

To discover a semaphore's owner, use the **get_sem_info()** function.

set_sem_owner() fails and returns **B_BAD_SEM_ID** or **B_BAD_TEAM_ID** if one or the other argument is invalid. Otherwise it returns **B_HOKEY_POKEY**.

See also: **get_sem_info()**

Areas

Declared in: <kernel/OS.h>

Overview

An area is a chunk of virtual memory. As such, it has all the expected properties of virtual memory: It has a starting address, a size, the addresses it comprises are contiguous, and it maps to (possibly non-contiguous) physical memory. The primary differences between an area and "standard" virtual memory (memory that you allocate through **malloc()**, for example) are these:

- Different areas can refer to the same physical memory. Put another way, different virtual memory addresses can map to the same physical locations. Furthermore, the different areas needn't belong to the same application. By creating and "cloning" areas, applications can easily share the same data.
- You can specify that the area's physical memory be locked into RAM when it's created, locked on a page-by-page basis as pages are swapped in, or that it be swapped in and out as needed.
- Areas always start on a page boundary, and are allocated in integer multiples of the size of a page. (A page is 4096 bytes, as represented by the **B_PAGE_SIZE** constant.)
- You can specify the starting address of the area's virtual memory. The specification can require that the area start precisely at a certain address, anywhere above a certain address, or anywhere at all.
- An area can be read- and write-protected.

Because areas are large—4096 bytes minimum—you don't create them arbitrarily. The two most compelling reasons to create an area are the two first points listed above: To share data among different applications, and to lock memory into RAM.

Identifying an Area

An area is uniquely identified (system-wide) by its **area_id** number. The **area_id** is assigned automatically by **create_area()**, a function that does what it says. Most of the other area functions require an **area_id** argument.

When you create an area, you get to name it. Area names are not unique—any number of areas can be assigned the same name.

Sharing Areas

If you want to share an area with another application, you can broadcast the area's **area_id** number, but it's recommended that, instead, you publish the area's name. Given an area name, a "remote" application can retrieve the area's ID number by calling **find_area()**.

To use an area that was created by another application, the first thing you do, having acquired the area's **area_id** through **find_area()**, is "clone" the area. You do this by calling the **clone_area()** function. The function returns a new **area_id** number that identifies your clone of the original area. All further references to the area (in the cloning application) must be based on the ID of the clone.

The physical memory that lies beneath a cloned area is never implicitly copied—for example, the area mechanism doesn't perform a "copy-on-write." If two areas (more specifically, two **area_id** numbers) refer to the same memory because of cloning, a data modification that's affected through one area will be seen by the other area.

Note: Because names aren't unique, multiple calls to **find_area()** with the same name won't all necessarily return the same **area_id**—consider the case where more than one instantiation of the same area-creating application is running on your computer.

Locking an Area

When you're working with moderately large amounts of data, it's often the case that you would prefer that the data remain in RAM, even if the rest of your application needs to be swapped out. An argument to **create_area()** lets you declare, through the use of one of the following constants, the locking scheme that you wish to apply to your area:

- **B_FULL_LOCK** means the area's memory is locked into RAM when the area is created, and won't be swapped out.
- **B_LAZY_LOCK** allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.
- **B_NO_LOCK** means pages are never locked, they're swapped in and out as needed.

Keep in mind that locking an area essentially reduces the amount of RAM that can be used by other applications, and so increases the likelihood of swapping. So you shouldn't lock simply because you're greedy. But if the area that you're locking is going to be shared among some number of other applications, or if you're writing a real-time application that processes large chunks of data, then locking can be a benefit.

The locking scheme is set by the **create_area()** function and is thereafter immutable. You can't re-declare the lock when you clone an area.

Using an Area

Ultimately, you use an area for the virtual memory that it represents: You create an area because you want some memory to which you can write and from which you can read data. These acts are performed in the usual manner, through references to specific addresses. Setting a pointer to a location within the area, and checking that you haven't exceeded the area's memory bounds as you increment the pointer (while reading or writing) are your own responsibility. To do this properly, you need to know the area's starting address and its extent:

- An area's starting address is maintained as the **address** field in its **area_info** structure; you retrieve the **area_info** for a particular area through the **get_area_info()** function.

- The size of the area (in bytes) is given as the **size** field of its **area_info** structure.

An important point, with regard to **area_info**, is that the **address** field is only valid for the application that created or cloned the area (in other words, the application that created the **area_id** that was passed to **get_area_info()**). Although the memory that underlies an area is global, the address that you get from an **area_info** structure refers to a specific address space.

If there's any question about whether a particular **area_id** is "local" or "foreign," you can compare the **area_info.team** field to your thread's team.

Deleting an Area

When your application quits, the areas (the **area_id** numbers) that it created through **create_area()** or **clone_area()** are automatically rendered invalid. The memory underlying these areas, however, isn't necessarily freed. An area's memory is freed only when (and as soon as) there are no more areas that refer to it.

You can force the invalidation of an **area_id** by passing it to the **delete_area()** function. Again, the underlying memory is only freed if yours is the last area to refer to the memory.

Deleting an area, whether explicitly through **delete_area()**, or because your application quit, never affects the status of other areas that were cloned from it.

Functions

area_for()

area_id **area_for(**void **addr***)**

Returns the **area_id** of the area that contains the given address within your own team's address space. The argument needn't be the starting address of an area, nor must it start on a page boundary: If the address lies anywhere within one of your application's areas, the ID of that area is returned.

Since the address is taken to be in the local address space, the area that's returned will also be local—it will have been created or cloned by your application.

If the address doesn't lie within an area, **B_ERROR** is returned.

See also: **find_area()**

clone_area()

```
long clone_area(const char *clone_name,
                          void **clone_addr,
                          ulong clone_addr_spec,
                          ulong clone_protection,
                          area_id source_area)
```

Creates a new area (the *clone* area) that maps to the same physical memory as an existing area (the *source* area). The arguments are:

- *clone_name* is the name that you wish to assign to the clone area. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.
- *clone_addr* points to a value that gives the address at which you want the clone area to start; the pointed-to value must be a multiple of **B_PAGE_SIZE** (4096). The function sets the value pointed to by *clone_addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **clone_addr* depends on the value of *clone_addr_spec*, as explained next.
- *clone_addr_spec* is one of four constants that describes how *clone_addr* is to be interpreted. The first three constants, **B_EXACT_ADDRESS**, **B_BASE_ADDRESS**, and **B_ANY_ADDRESS**, have meanings as explained under **create_area()**.

  The fourth constant, **B_CLONE_ADDRESS**, specifies that the address of the cloned area should be the same as the address of the source area. Cloning the address is convenient if you have two (or more) applications that want to pass pointers to each other—by using cloned addresses, the applications won't have to offset the pointers that they receive. For both the **B_ANY_ADDRESS** and **B_CLONE_ADDRESS** specifications, the value that's pointed to by the *clone_addr* argument is ignored.

- *clone_protection* is one or both of **B_READ_AREA** and **B_WRITE_AREA**. These have the same meaning as in **create_area()**; keep in mind, as described there, that a cloned area can have a protection that's different from that of its source.

- *source_area* is the **area_id** of the area that you wish to clone. You usually supply this value by passing an area name to the **find_area()** function.

The cloned area inherits the source area's locking scheme (**B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**).

Usually, the source area and clone area are in two different applications. It's possible to clone an area from a source that's in the same application, but there's not much reason to do so unless you want the areas to have different protections.

If **area_clone()** clone is successful, the clone's **area_id** is returned. Otherwise, the function returns one of the following error constants:

Constant	Meaning
B_BAD_VALUE	Bad argument value; you passed an unrecognized constant for *addr_spec* or *lock*, the *addr* value isn't a multiple of **B_PAGE_SIZE**, you set *addr_spec* to **B_EXACT_ADDRESS** or **B_CLONE_ADDRESS** but the address request couldn't be fulfilled, or *source_area* doesn't identify an existing area.
B_NO_MEMORY	Not enough memory to allocate the system structures that support this area.
B_ERROR	Some other system error prevented the area from being created.

See also: **create_area()**, **delete_area()**

create_area()

```
area_id create_area(const char *name,
                                void **addr,
                                ulong addr_spec,
                                ulong size,
                                ulong lock,
                                ulong protection)
```

Creates a new area and returns its **area_id**. The arguments are:

- *name* is the name that you wish to assign to the area. It needn't be unique. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.

- *addr* points to the address at which you want the area to start. The value of **addr* must signify a page boundary; in other words, it must be an integer multiple of **B_PAGE_SIZE** (4096). Note that this is a pointer to a pointer: **addr*—not *addr*—

should be set to the desired address; you then pass the address of *addr* as the argument, as shown below:

```
/* Set the address to a page boundary. */
char *addr = (char *)(4096 * 100);

/* Pass the address of addr as the second argument. */
create_area( "my area", &addr, ...);
```

The function sets the value of **addr* to the area's actual starting address—it may be different from the one you requested. The constancy of **addr* depends on the value of *addr_spec*, as explained next.

- *addr_spec* is a constant that tells the function how the **addr* value should be applied. There are three address specification constants:

  B_EXACT_ADDRESS means you want the value of **addr* to be taken literally and strictly. If the area can't be allocated at that location, the function fails.

  B_BASE_ADDRESS means the area can start at a location equal to or greater than **addr.*

  B_ANY_ADDRESS means the starting address is determined by the system. In this case, the value that's pointed to by *addr* is ignored (going into the function).

  (A fourth specification, **B_CLONE_ADDRESS**, is only used by the **clone_area()** function.)

- *size* is the size, in bytes, of the area. The size must be an integer multiple of **B_PAGE_SIZE** (4096). The upper limit of *size* depends on the available swap space (or RAM, if the area is to be locked).

- *lock* describes how the physical memory should be treated with regard to swapping. There are three locking constants:

  B_FULL_LOCK means the area's memory is immediately locked into RAM and won't be swapped out.

  B_LAZY_LOCK allows individual pages of memory to be brought into RAM through the natural order of things and *then* locks them.

  B_NO_LOCK means pages are never locked, they're swapped in and out as needed.

- *protection* is a mask that describes whether the memory can be written and read. You form the mask by adding the constants **B_READ_AREA** (the area can be read) and **B_WRITE_AREA** (it can be written). The protection you describe applies only to this area. If your area is cloned, the clone can specify a different protection.

If **create_area()** is successful, the new **area_id** number is returned. If it's unsuccessful, one of the following error constants is returned:

Constant	Meaning
B_BAD_VALUE	Bad argument value. You passed an unrecognized constant for *addr_spec* or *lock*, the *addr* or *size* value isn't a multiple of **B_PAGE_SIZE**, or you set *addr_spec* to **B_EXACT_ADDRESS** but the address request couldn't be fulfilled.
B_NO_MEMORY	Not enough memory to allocate the necessary system structures that support this area. Note that this error code *doesn't* mean that you asked for too much physical memory.
B_ERROR	Some other system error prevented the area from being created. Most notably, **B_ERROR** is returned if *size* is too large.

See also: **clone_area()**, **delete_area()**

delete_area()

long **delete_area(**area_id *area***)**

Deletes the designated area. If no one other area maps to the physical memory that this area represents, the memory is freed.

Note: Currently, anybody can delete any area—the act isn't denied if, for example, the **area_id** argument was created by another application. This freedom will be rescinded in a later release. Until then, try to avoid deleting other application's areas.

If *area* doesn't designate an actual area, this function returns **B_ERROR**; otherwise it returns **B_NO_ERROR**.

See also: **create_area()**, **clone_area()**

find_area()

area_id **find_area(**const char **name***)**

Returns an area that has a name that matches the argument. Area names needn't be unique—successive calls to this function with the same argument value may not return the same **area_id**.

What you do with the area you've found depends on where it came from:

- If you're finding an area that your own application created or cloned, you can use the returned ID directly.
- If the area was created or cloned by some other application, you should immediately clone the area (unless you're doing something truly innocuous, such as simply examining the area's info structure).

If the argument doesn't identify an existing area, the **B_NAME_NOT_FOUND** error code is returned.

See also: **area_for()**

get_area_info(), get_nth_area_info()

```
long get_area_info(area_id area, area_info *info)
long get_nth_area_info(team_id team, long n, area_info *info)
```

Copies information about a particular area into the **area_info** structure designated by *info*. The first version of the function designates the area directly, by **area_id**. The second version designates the *n*'th area within the given team. If the *team* argument is 0, all teams are considered.

The **area_info** structure is defined as:

```
typedef struct area_info {
        area_id  area;
        char  name[B_OS_NAME_LENGTH];
        void  *address;
        ulong  size;
        ulong  lock;
        ulong  protection;
        team_id  team;
        ulong  ram_size;
        ulong  copy_count;
        ulong  in_count;
        ulong  out_count;
} area_info;
```

The fields are:

- **area** is the **area_id** that identifies the area. This will be the same as the function's *area* argument.
- **name** is the name that was assigned to the area when it was created or cloned.
- **address** is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the application that created (or cloned) the area.
- **size** is the size of the area, in bytes.
- **lock** is a constant that represents the area's locking scheme. This will be one of **B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**.
- **protection** specifies whether the area's memory can be read or written. It's a combination of **B_READ_AREA** and **B_WRITE_AREA.**
- **team** is the **team_id** of the thread that created or cloned this area.

The final four fields give information about the area that's useful in diagnosing system use. The fields are particularly valuable if you're hunting for memory leaks:

- **ram_size** gives the amount of the area, in bytes, that's currently swapped in.
- **copy_count** is a "copy-on-write" count that can be ignored—it doesn't apply to the areas that you create. The system can create copy-on-write areas (it does so when it loads the data section of an executable, for example), but you can't.
- **in_count** is a count of the total number of times any of the pages in the area have been swapped in.
- **out_count** is a count of the total number of times any of the pages in the area have been swapped out.

If the *area* argument doesn't identify an existing area, the function returns **B_BAD_VALUE**; otherwise it returns **B_NO_ERROR**.

resize_area()

long **resize_area(**area_id *area*, ulong *new_size***)**

Sets the size of the designated area to *new_size*, measured in bytes. The *new_size* argument must be a multiple of **B_PAGE_SIZE** (4096).

Size modifications affect the end of the area's existing memory allocation: If you're increasing the size of the area, the new memory is added to the end of area; if you're shrinking the area, end pages are released and freed. In neither case does the area's starting address change, nor is existing data modified (expect, of course, for data that's lost due to shrinkage).

If the function is successful, **B_NO_ERROR** is returned. Otherwise one of the following error codes is returned:

Constant	Meaning
B_BAD_VALUE	Either *area* doesn't signify a valid area, or *new_size* isn't a multiple of **B_PAGE_SIZE**.
B_NO_MEMORY	Not enough memory to allocate the system structures that support the new portion of the area. This should only happen if you're increasing the size of the area. Note that this error code *doesn't* mean that you asked for too much physical memory.
B_ERROR	Some other system error prevented the area from being created. Most notably, **B_ERROR** is returned if *new_size* is too large.

See also: **create_area()**

set_area_protection()

long **set_area_protection(**area_id *area*, ulong *new_protection***)**

Sets the given area's read and write protection. The *new_protection* argument is a mask that specifies one or both of the values **B_READ_AREA** and **B_WRITE_AREA**. The former means that the area can be read; the latter, that it can be written to. An area's protection only applies to access to the underlying memory through that specific area. Different area clones that refer to the same memory may have different protections.

The function fails (the old protection isn't changed) and returns **B_BAD_VALUE** if area doesn't identify a valid area; otherwise it returns **B_NO_ERROR**.

See also: **create_area()**

Images

Declared in: <kernel/image.h>

Overview

An *image* is compiled code; put another way, an image is what the compiler produces. There are three types of images:

- An *app image* is an application. Every application has a single app image.
- A *library image* is a dynamically linked library (a "shared library"). Most applications link against the system library (**libbe.so**) that Be provides.
- An *add-on image* is an image that you load into your application as it's running. Symbols from the add-on image are linked and references are resolved when the image is loaded. Thus, an add-on image provides a sort of "heightened dynamic linking" beyond that of a DLL.

The following sections explain how to load and run an app image, how to create a shared library, and how to create and load an add-on image.

Loading an App Image

Loading an app image is like running a "sub-program." The image that you load is launched in much the same way as had you double-clicked it in the Browser, or launched it from the command line. It runs in its own team—it doesn't share the address space of the application from which it was launched—and, generally, leads its own life.

Any application can be loaded as an app image; you don't need to issue special compile instructions or otherwise manipulate the binary. The one requirement of an app image is that it must have a **main()** function; hardly a restrictive request.

To load an app image, you call the **load_executable()** function, the protocol for which is:

```
thread_id load_executable(BFile *file,
                          int argc,
                          const char **argv,
                          const char **env)
```

The function takes, as its first argument, a BFile object that represents the image file. Having located the file, the function creates a new team, spawns a main thread in that team, and then returns the **thread_id** of that thread to you. The thread that's returned is the executable’s main thread. It won’t be running: To make it run you pass the **thread_id** to **resume_thread()** or **wait_for_thread()** (as explained in the major section “Threads and Teams”).

In addition to the BFile argument, **load_executable()** takes an *argc*/*argv* argument pair (which are copied and forwarded to the new thread’s **main()** function), as well as a pointer to an array of environment variables (strings):

- The *argc*/*argv* arguments must be set up properly—you can’t just pass 0 and **NULL**. To properly instantiate the arguments, the first string in the *argv* array must be the name of the image file (in other words, the name of the program that you’re going to launch). You then install any other arguments you want in the array, and terminate the array with a **NULL** entry. *argc* is set to the number of entries in the *argv* array (not counting the terminating **NULL**). It’s the caller’s responsibility to free the *argv* array after **load_executable()** returns.

- *envp* is an array of environment variables that are also passed to **main()**. Typically, you use the global **environ** pointer (which you must declare as an **extern**—see the example, below). You can, of course, create your environment variable array: As with the *argv* array, the *envp* array should be terminated with a **NULL** entry, and you must free the array when **load_executable()** returns (that is, if you allocated it yourself—don’t try to free **environ**).

The following example demonstrates a typical use of **load_executable()**. First, we include the appropriate files and declare the necessary variables:

```
#include <image.h>  /* load_executable() */
#include <OS.h>     /* wait_for_thread() */
#include <stdlib.h> /* malloc() */

/* Here's how you declare the environment variable array. */
extern char **environ;

BFile exec_file;
record_ref exec_ref;
char **arg_v; /* choose a name that doesn't collide with argv */
long arg_c; /* same here vis a vis arg_c */
thread_id exec_thread;
long return_value;
```

Next, we set our BFile’s ref so the object refers to the executable file, which we’re calling **adder**. For this example, we use **get_ref_for_path()** to set the ref’s value (see the Storage Kit chapter for more information on these manipulations):

```
get_ref_for_path("/hd/my_apps/adder", &exec_ref);
exec_file.SetRef(exec_ref);
```

Install, in the **arg_v** array, the “command line” arguments that we’re sending to **adder**. Let’s pretend the **adder** program takes two integers, adds them together, and returns the result as **main()**'s exit code. Thus, there are three arguments: The name of the program (“adder”), and the values of the two addends converted to strings. Since there are three arguments, we allocate **arg_v** to hold four pointers (to accommodate the final **NULL**). Then we allocate and copy the arguments.

```
arg_c = 3;
arg_v = (char **)malloc(sizeof(char *) * (agc + 1));

arg_v[0] = strdup("adder");
arg_v[1] = strdup("5");
arg_v[2] = strdup("3");
arg_v[3] = NULL;
```

Now that everything is properly set up, we call **load_executable()**. After the function returns, it's safe to free the allocated **arg_v** array:

```
exec_thread=load_executable(&exec_file, arg_c, arg_v, environ);
free(arg_v);
```

At this point, **exec_thread** is suspended (the natural state of a newly-spawned thread). In order to retrieve its return value, we use **wait_for_thread()** to tell the thread to run:

```
wait_for_thread(exec_thread, &return_value);
```

After **wait_for_thread()** returns, the value of **return_value** should be 8 (i.e. 5 + 3).

Creating a Shared Library

The primary documentation for creating a shared library is provided by MetroWerks in their CodeWarrior manual. Beyond the information that you find there, you should be aware of the following amendments and caveats.

- You mustn't export your library's symbols through the **-export all** compiler flag. Instead, you should either use **-export pragma** or **-@export *filename*** (which is the same as **-f *filename***). See the MetroWerks manual for details on how to use these flags.
- The libraries that you create must be placed in **/system/lib** so the loader can find them when an application (that's uses your libraries) is launched.

Creating and Using an Add-on Image

An add-on image is indistinguishable from a shared library image. Creating an add-on is, therefore, exactly like creating a shared library, a topic that we breezed through immediately above. The one exception to the rules given above is in where the add-on must live: You can keep your add-ons anywhere in the file system. When you load an add-on (through the **load_add_on()** function), you have to refer to the add-on file directly through the use of a BFile—the system doesn't search for the file for you.

Loading an Add-on Image

To load an add-on into your application, you call the **load_add_on()** function. The function takes a pointer to a BFile object that refers to the add-on file, and returns an **image_id** number that uniquely identifies the image across the entire system.

For example, let's say you've created an add-on image that's stored in the file **/hd/addons/adder** (the add-on will perform the same adding operation that was demonstrated in the **load_executable()** example). The code that loads the add-on would look like this:

```
/* For brevity, we won't check errors.  */
BFile addon_file;
record_ref addon_ref;
image_id addon_image;

/* Establish the file's ref.  */
get_ref_for_path("/hd/addons/adder", &addon_ref);
addon_file.SetRef(addon_ref);

/* Load the add-on. */
addon_image = load_add_on(&addon_file);
```

Unlike loading an executable, loading an add-on doesn't create a separate team (nor does it spawn another thread). The whole point of loading an add-on is to bring the image into your application's address space so you can call the functions and fiddle with the variables that the add-on defines.

Symbols

After you've loaded an add-on into your application, you'll want to examine the symbols (variables and functions) that it has brought with it. To get information about a symbol, you call the **get_image_symbol()** function:

long **get_image_symbol(**image_id *image*,
char **symbol_name*,
long *symbol_type*,
void ***location***)**

The function's first three arguments identify the symbol that you want to get:

- The first argument is the **image_id** of the add-on that owns the symbol.
- The second argument is the symbol's name. This assumes, of course, that you know the name. In general, using an add-on implies just this sort of cooperation.
- The third is a constant that gives the symbol's *symbol type*. The only types you should care about are **B_SYMBOL_TYPE_DATA** which you use for variables, and **B_SYMBOL_TYPE_TEXT** which you use for functions.

The function returns, by reference in its final argument, a pointer to the symbol's address. For example, let's say the **adder** add-on code looks like this:

```
long addend1 = 0;
long addend2 = 0;

long adder(void)
{
    return (addend1 + addend2);
}
```

To examine the variables (**addend1** and **addend2**), you would call **get_image_symbol()** thus:

```
long *var_a1, *var_a2;

/* addon_image is the image_id that was returned by the
 * load_add_on() call in the previous example.
 */
get_image_symbol(addon_image, "addend1", 1, &var_a1);
get_image_symbol(addon_image, "addend2", 1, &var_a2);
```

To get the symbol for the **adder()** function is a bit more complicated. The compiler renames a function's symbol to encode the data types of the function's arguments. The encoding scheme is explained in the next section; to continue with the example, we'll simply accept that the **adder()** function's symbol is

```
adder__Fv
```

And so...

```
long (*func_add)();
get_image_symbol(addon_image, "adder__Fv", 2, &func_add);
```

Now that we've retrieved all the symbols, we can set the values of the two addends and call the function:

```
*var_a1 = 5;
*var_a2 = 3;
long return_value = (*func_add)();
```

Function Symbol Encoding

The compiler encodes function symbols according to this format:

*functionName*__**F**<*arg1Type*><*arg2Type*><*arg3Type*>....

where the argument type codes are

Code	Type
i	**int**
l	**long**

f	**float**
d	**double**
c	**char**
v	**void**

In addition, if the argument is declared as unsigned, the type code character is preceded by "U". If it's a pointer, the type code (and, potentially, the "U") is preceded by "P"; a pointer to a pointer is preceded by "PP". For example, a function that's declared as

```
void Func(long, unsigned char **, float *, double);
```

would have the following symbol name:

```
Func__FlUPPcPfd
```

Note that **typedef**'s are translated to their natural types. So, for example, this:

```
void dump_thread(thread_id, bool);
```

becomes

```
dump_thread__FlUc
```

Functions

get_image_info(), get_nth_image_info()

long **get_image_info(**image_id *image*, image_info **info***)**

long **get_nth_image_info(**team_id *team*,
long *n*,
image_info **info***)**

These functions return information about a particular image. The first version identifies the image by its first argument; the second version locates the *n*'th image that's loaded into *team*. The information is returned in the *info* argument. The **image_info** structure is defined as:

typedef struct {
long **volume**;
long **directory**;
char **name**[B_FILE_NAME_LENGTH];
image_id **id**;
void ***text**;
long **text_size**;
void ***data**;
long **data_size**;
image_type **type**;
} **image_info**

The volume and directory fields are, practically speaking, private. The other fields are:

- **name**. The name of the file whence sprang the image.
- **id.** The image's **image_id** number.
- **text** and **text_size**. The address and the size (in bytes) of the image's text segment.
- **data** and **data_size**. The address and size of the image's data segment.
- **type**. A constant that tells whether this is an app, library, or add-on image.

The self-explanatory **image_type** constants are:

- **B_APP_IMAGE**
- **B_LIBRARY_IMAGE**
- **B_ADD_ON_IMAGE**

The functions return **B_BAD_IMAGE_ID** or **B_BAD_INDEX** if the designated image doesn't exist. Otherwise, they return **B_NO_ERROR**.

get_image_symbol(), get_nth_image_symbol()

```
long get_image_symbol(image_id image,
                      char *symbol_name,
                      long symbol_type,
                      void **location)

long get_nth_image_symbol(image_id image,
                          long n,
                          char *name,
                          int *name_length,
                          int *symbol_type,
                          void **location)
```

get_image_symbol() returns, in *location*, a pointer to the address of the symbol that's identified by the *image*, *symbol_name*, and *symbol_type* arguments. An example demonstrating the use of this function is given in "Symbols" on page 58.

get_nth_image_symbol() returns information about the *n*'th symbol in the given image. The information is returned in the arguments:

- *name* is the name of the symbol. You have to allocate the *name* buffer before you pass it in—the function copies the name into the buffer.

- You point *name_length* to an integer that gives the length of the *name* buffer that you're passing in. The function uses this value to truncate the string that it copies into *name*. The function then resets *name_length* to the full (untruncated) length of the symbol's name (plus one byte to accommodate a terminating **NULL**). To ensure that you've gotten the symbol's full name, you should compare the in-going value of *name_length* with the value that the function sets it to. If the in-going value is less than the full length, you can then re-invoke **get_nth_image_symbol()** with an adequately lengthened *name* buffer, and an increased *name_length* value.

Important: Keep in mind that *name_length* is reset each time you call **get_nth_image_symbol()**. If you're calling the function iteratively (to retrieve all the symbols in an image), you need to reset the *name_length* value between calls.

- The function sets *symbol_type* to **B_SYMBOL_TYPE_DATA** if the symbol is a variable, or **B_SYMBOL_TYPE_TEXT** if the symbol is a function. The argument's value going into the function is of no consequence.
- The function sets *location* to point to the symbol's address.

To retrieve **image_id** numbers on which these functions can act, use the **get_nth_image_info()** function. Such numbers are also returned directly when you load an add-on image through the **load_add_on()** function.

The functions return **B_BAD_IMAGE_ID** or **B_BAD_INDEX** if the designated image doesn't exist. Otherwise, they return **B_NO_ERROR**.

load_add_on(), unload_add_on()

image_id **load_add_on(**BFile **file***)**
long **unload_add_on(**image_id *image***)**

load_add_on() loads an add-on image, identified by *file*, into your application's address space. The function returns an **image_id** (a positive integer) that represents the loaded image. An example that demonstrates the use of **load_add_on()** is given in "Loading an Add-on Image" on page 58.

You can load the same add-on image twice; each time you load the add-on a new, unique **image_id** is created and returned. If the requested file couldn't be loaded as an add-on (for whatever reason), the function returns **B_ERROR**.

unload_add_on() removes the add-on image identified by the argument. The image's symbols are removed, and the memory that they represent is freed. If the argument doesn't identify a valid image, the function returns **B_ERROR**. Otherwise, it returns **B_NO_ERROR**.

load_executable()

```
thread_id load_executable(BFile *file,
                          int argc,
                          const char **argv,
                          const char **env)
```

Loads an app image into the system (it *doesn't* load the image into the caller's address space), creates a separate team for the new application, and spawns and returns the ID of the team's main thread. The image is identified by the *file* argument; *file* must have its ref set before this function is called. It's of no consequence whether the object is open or closed when you call this function.

The other arguments are passed to the image's **main()** function (they show up there as the function's similarly named arguments):

- *argc* gives the number of entries that are in the *argv* array.
- The first string in the *argv* array must be the name of the image file (in other words, the name of the program that you're going to launch). You then install any other arguments you want in the array, and terminate the array with a **NULL** entry. Note that the value of *argc* shouldn't count *argv*'s terminating **NULL**.
- *envp* is an array of environment variables that are also passed to **main()**. Typically, you use the global **environ** pointer:

```
extern char **environ;

load_executable(..., environ);
```

The *argv* and *envp* arrays are copied into the new thread's address space. If you allocated either of these arrays, it's safe to free them immediately after **load_executable()** returns.

The thread that's returned by **load_executable()** is in a suspended state. To start the thread running, you pass the **thread_id** to **resume_thread()** or **wait_for_thread()**.

An example that demonstrates the use of **load_executable()** is given in "Loading an App Image" on page 55.

If the function returns **B_ERROR** upon failure.

Miscellaneous Functions, Constants, and Defined Types

Miscellaneous Functions

debugger()

void **debugger(**const char **string***)**

Throws the calling thread into the debugger. The *string* argument becomes the debugger's first utterance.

get_system_info()

long **get_system_info(**system_info **info***)**

Returns information about the computer. The information is returned in *info*, a **system_info** structure.

is_computer_on()

long **is_computer_on(**void**)**

Returns 1 if the computer is on. If the computer isn't on, the value returned by this function is undefined.

system_time()

double **system_time(**void**)**

Returns the number of microseconds that have elapsed since the computer was booted.

Constants

Area Location Constants

<kernel/OS.h>

Constant	Meaning
B_ANY_ADDRESS	Put the area anywhere
B_EXACT_ADDRESS	The area must start exactly at a given address
B_BASE_ADDRESS	The area can start anywhere above a given address
B_CLONE_ADDRESS	The clone must start at the same address as the original

These constants represent the different locations at which an area can be placed when it's created. They're used as values for the address arguments in **create_area()** and **clone_area()**. **B_CLONE_ADDRESS** can be passed to **clone_area()** only; the other three can be passed to either function.

See also: "Areas" on page 45

Area Lock Constants

<kernel/OS.h>

Constant	Meaning
B_NO_LOCK	Never lock the area's pages
B_LAZY_LOCK	Lock pages as they're swapped in
B_FULL_LOCK	Lock all pages now

These constants represent an area's "locking scheme," the circumstances in which the area's underlying memory is locked into RAM. You set the locking scheme for an area by passing one of these constants to **create_area()**'s lock argument; the scheme can't be changed thereafter.

To query an area's locking scheme, retrieve its **area_info** structure (through **get_area_info()**) and look at the **lock** field.

See also: "Areas" on page 45

Area Protection Constants

<kernel/OS.h>

Constant	Meaning
B_READ_AREA	The area can be read from
B_WRITE_AREA	The area can be written into

These constants represent the read and write protection that's enforced for an area. The constants are flags that can be added together and passed as the protection argument to

create_area() and **clone_area()**. You can change an area's protection through the **set_area_protection()** function.

To query an area's protection, retrieve its **area_info** structure (through **get_area_info()**) and look at the **protection** field.

See also: "Areas" on page 45

CPU Count

<kernel/OS.h>

Constant	Value
B_MAX_CPU_NUM	8

This constant gives the maximum number of CPUs that a single system can support. The **cpu_count** field of the **system_info** structure gives the number of CPUs that are actually on a given system. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

CPU Type Constant

<kernel/OS.h>

Constant
B_CPU_PPC_601
B_CPU_PPC_603
B_CPU_PPC_603e
B_CPU_PPC_604
B_CPU_PPC_604e
B_CPU_PPC_686

These constants represent the different CPU chips that the BeBox has used/is using/might use. To discover which chip the local machine is using, looking in the **cpu_type** field of the **system_info** structure. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

File Name Length

<kernel/OS.h>

Constant	Value
B_FILE_NAME_LENGTH	64

This constant gives the maximum length of the name of a file or directory.

Image Type Constants

<kernel/image.h>

Constant	Meaning
B_APP_IMAGE	Application image
B_LIBRARY_IMAGE	Shared library image
B_ADD_ON_IMAGE	Add-on image
B_SYSTEM_IMAGE	System-defined image

The image type constants (type **image_type**) enumerate the different *images*, or loadable compiled code, that you can create or otherwise find on the system. Of the four image types, you can't create **B_SYSTEM_IMAGES**; however, it's possible to run across a system-defined image if you retrieve all the image info structures for all teams (through the **get_nth_image_info()** function).

See also: "Images" on page 55

Image Symbol Type Constants

<kernel/OS.h>

Constant	Meaning
B_SYMBOL_TYPE_DATA	The symbol is a variable
B_SYMBOL_TYPE_TEXT	The symbol is a function

The image symbol type constants describe the nature of a particular image symbol. You retrieve symbol information from an image through the **get_image_symbol()**.

See also: "Images" on page 55

Operating System Doodad Name Length

<kernel/OS.h>

Constant	Value
B_OS_NAME_LENGTH	32

This constant gives the maximum length of the name of a thread, semaphore, port, area, or other operating system bauble.

Page Size

<kernel/OS.h>

Constant	Value
B_PAGE_SIZE	4096

The **B_PAGE_SIZE** constant gives the size, in bytes, of a page of RAM.

Port Message Count

<kernel/OS.h>

Constant	Value
B_MAX_PORT_COUNT	128

This constant gives the maximum number of messages that a port can hold at a time. This value isn't applied automatically—you declare a port's message capacity when you create it.

To query a port's message capacity, retrieve its **port_info** structure (through the **get_port_info()** function) and look at the **capacity** field.

See also: "Ports" on page 23

Semaphore Control Flags

<kernel/OS.h>

Acquire Flag	Meaning
B_TIMEOUT	Honor **acquire_sem_etc()**'s *timeout* argument.
B_CAN_INTERRUPT	The semaphore can be interrupted by a signal.
B_CHECK_PERMISSION	Make sure this isn't a system semaphore.

Release Flag	Meaning
B_DO_NOT_RESCHEDULE	Don't reschedule after the semaphore is released.

These are the flag values that can be passed to the **acquire_sem_etc()** and **release_sem_etc()** functions.

The timeout flag (**B_TIMEOUT**) applies to all semaphore acquisitions: If, having set the **B_TIMEOUT** flag, your **acquire_sem_etc()** call blocks, the acquisition attempt will give up after some number of microseconds (as given in the function's *timeout* argument). If **B_TIMEOUT** isn't set, the acquisition can, potentially, block forever. The other two acquisition flags are used by device driver writers only. The meanings of these flags is given in the Device Kit chapter.

The **B_DO_NOT_RESCHEDULE** flag applies to the **release_sem_etc()** function. Normally, when a semaphore is released, the kernel immediately finds another thread to run, even if the releasing thread hasn't used up a full "schedule quantum" worth of CPU attention. By setting the **B_DO_NOT_RESCHEDULE** flag, you tell the scheduler to let the releasing thread run for its normally alloted amount of time.

Thread Priority Constants

<kernel/OS.h>

Time-Sharing Priority	Value
B_LOW_PRIORITY	5
B_NORMAL_PRIORITY	10
B_DISPLAY_PRIORITY	15
B_URGENT_DISPLAY_PRIORITY	20

Real-Time Priority	Value
B_REAL_TIME_DISPLAY_PRIORITY	100
B_URGENT_PRIORITY	110
B_REAL_TIME_PRIORITY	120

These constants represent the thread priority levels. The higher a thread's priority value, the more attention it gets from the CPUs; the constants are listed here from lowest to highest priority.

There are two priority categories:

- Time-sharing priorities (priority values from 1 to 99).
- Real-time priorities (100 and greater).

A time-sharing thread (a thread with a time-sharing priority value) is executed only if there are no real-time threads in the ready queue. In the absence of real-time threads, a time-sharing thread is elected to run once every "scheduler quantum" (currently, every three milliseconds). The higher the time-sharing thread's priority value, the greater the chance that it will be the next thread to run.

A real-time thread is executed as soon as it's ready. If more than one real-time thread is ready at the same time, the thread with the highet priority is executed first. The thread is allowed to run without being preempted (except by a real-time thread with a higher priority) until it blocks, snoozes, is suspended, or otherwise gives up its plea for attention.

You set a thread's priority when you spawn it (**spawn_thread()**); it can be changed thereafter through the **set_thread_priority()** function. Although you can set a thread priority to values other than those defined by the constants shown here, it's strongly suggested that you stick with the constants.

To query a thread's priority, look at the **priority** field of the thread's **thread_info** structure (which you can retrieve through **get_thread_info()**).

See also: "Threads and Teams" on page 5

Thread State Constants

<kernel/OS.h>

Constant	Meaning
B_THREAD_RUNNING	The thread is currently receiving attention from a CPU.
B_THREAD_READY	The thread is waiting for its turn to run.
B_THREAD_RECEIVING	The thread is sitting in a **receive_data()** call.
B_THREAD_ASLEEP	The thread is sitting in a **snooze()** call.
B_THREAD_SUSPENDED	The thread has been suspended or is freshly-spawned.
B_THREAD_WAITING	The thread is waiting to acquire a semaphore.

These constants (type **thread_state**) represent the various states that a thread can be in. You can't set a thread's state directly; the state changes as the result of the thread's sequence of operations.

You can query a thread's state by looking at the **state** field of its **thread_info** structure. To retrieve the structure, call **get_thread_info()**. Be aware, however, that a thread's state is extremely ephemeral; by the time you retrieve it, it may have changed.

See also: "Threads and Teams" on page 5

System Team ID

<kernel/OS.h>

Constant	Meaning
B_SYSTEM_TEAM	The **team_id** of the kernel's team

The **B_SYSTEM_TEAM** constant identifies the kernel's team. You should only need to use this constant if you're bequeathing ownership of a port or semaphore to the kernel, an activity that's typically the province of driver writers.

Defined Types

area_id

<kernel/OS.h>

typedef long **area_id**

The **area_id** type uniquely identifies area.

See also: "Areas" on page 45

area_info

<kernel/OS.h>

```
typedef struct {
        area_id area;
        char name[B_OS_NAME_LENGTH];
        void *address;
        ulong size;
        ulong lock;
        ulong protection;
        team_id team;
        ulong  ram_size;
        ulong  copy_count;
        ulong  in_count;
        ulong  out_count;
} area_info
```

The **area_info** structure holds information about a particular area. **area_info** structures are retrieved through the **get_area_info()** function. The structure's fields are:

- **area** is the **area_id** that identifies the area.
- **name** is the name that was assigned to the area when it was created or cloned.

- **address** is a pointer to the area's starting address. Keep in mind that this address is only meaningful to the application that created (or cloned) the area.
- **size** is the size of the area, in bytes.
- **lock** is a constant that represents the area's locking scheme. This will be one of **B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**.
- **protection** specifies whether the area's memory can be read or written. It's a combination of **B_READ_AREA** and **B_WRITE_AREA.**
- **team** is the **team_id** of the thread that created or cloned this area.

Three of the final four fields (you can ignore the **copy_count** field) give information about the area that's useful in diagnosing system use:

- **ram_size** gives the amount of the area, in bytes, that's currently swapped in.
- **in_count** is the number of times the system has swapped in a page from the area.
- **out_count** is the number of times pages have been swapped out.

See also: "Areas" on page 45

cpu_info

```
<kernel/OS.h>

typedef struct {
      double  active_time;
} cpu_info
```

The **cpu_info** structure describes facets of a particular CPU. Currently, the structure contains only one field, **active_time**, that measures the amount of time, in microseconds, that the CPU has actively been working since the machine was last booted. One structure for each CPU is created and maintained by the system. An array of all such structures can be found in the **cpu_infos** field of the **system_info** structure. To retrieve a **system_info** structure, you call the **get_system_info()** function.

See also: **system_info**

image_info

```
<kernel/image.h>

typedef struct {
      long  volume;
      long  directory;
      char  name[B_FILE_NAME_LENGTH];
      image_id id;
      void  *text;
      long  text_size;
      void  *data;
      long  data_size;
      image_type type;
} image_info
```

The **image_info** structure contains information about a specific image. The fields are:

- The **volume** and **directory** fields are, practically speaking, private.
- **name**. The name of the file whence sprang the image.
- **id.** The image's **image_id** number.
- **text** and **text_size**. The address and the size (in bytes) of the image's text segment.
- **data** and **data_size**. The address and size of the image's data segment.
- **type**. A constant that tells whether this is an app, library, or add-on image.

The self-explanatory **image_type** constants are:

- **B_APP_IMAGE**
- **B_LIBRARY_IMAGE**
- **B_ADD_ON_IMAGE**

image_type

```
<kernel/image.h>

typedef enum { ... } image_type
```

The **image_type** type defines the different image types.

See also: **Image Type Constants**

machine_id

<kernel/OS.h>

typedef long **machine_id**[2]

The **machine_id** type encodes a 64-bit number that uniquely identifies a particular BeBox. To discover the machine id of the local machine, look in the **id** field of the **system_info** structure. To retrieve this structure, call the **get_system_info()** function.

See also: **get_system_info()**

port_id

<kernel/OS.h>

typedef long **port_id**

The **port_id** type uniquely identifies ports.

See also: "Ports" on page 23

port_info

<kernel/OS.h>

```
typedef struct port_info {
        port_id port;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long capacity;
        long queue_count;
        long total_count;
} port_info
```

The **port_info** structure holds information about a particular port. It's fields are:

- **port**. The **port_id** number of the port.
- **team**. The **team_id** of the port's team.
- **name**. The name assigned to the port.
- **capacity**. The length of the port's message queue.
- **queue_count**. The number of messages currently in the queue.
- **total_count**. The total number of message that have been read from the port.

Note that the **total_count** number doesn't include the messages that are currently in the queue.

You retrieve a **port_info** structure through the **get_port_info()** function.

See also: "Ports" on page 23

sem_id

```
<kernel/OS.h>

typedef long sem_id
```

The **sem_id** type uniquely identifies semaphores.

See also: "Semaphores" on page 31

sem_info

```
<kernel/OS.h>

typedef struct sem_info {
        sem_id sem;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        long count;
        thread_id latest_holder;
} sem_info
```

The **sem_info** structure holds information about a given semaphore. The structure's fields are:

- **sem**. The **sem_id** number of the semaphore.
- **team**. The **team_id** of the semaphore's owner.
- **name**. The name assigned to the semaphore.
- **count**. The semaphore's thread count.
- **latest_holder**. The thread that most recently acquired the semaphore.

Note that the thread that's identified in the **lastest_holder** field may no longer be holding the semaphore—it may have since released the semaphore. The latest holder is simply the last thread to have called **acquire_sem()** (of whatever flavor) on this semaphore.

You retrieve a sem_info structure through the **get_sem_info()** function.

See also: "Semaphores" on page 31

system_info

```
<kernel/OS.h>

typedef struct {
        machine_id id;
        double boot_time;
        long cpu_count;
        long cpu_type;
        long cpu_revision;
        cpu_info cpu_infos[B_MAX_CPU_NUM];
        double cpu_clock_speed;
        double bus_clock_speed;
        long max_pages;
        long used_pages;
        long page_faults;
        long max_sems;
        long used_sems;
        long max_ports;
        long used_ports;
        long max_threads;
        long used_threads;
        long max_teams;
        long used_teams;
        long volume;
        long directory;
        char name [B_FILE_NAME_LENGTH];
} system_info
```

The **system_info** structure holds information about the machine and the state of the kernel. The structure's fields are:

- **id.** The 64-bit number (encoded as two **long**s) that uniquely identifies this machine.

- **boot_time**. The time at which the computer was last booted, measured in microseconds since January 1st, 1970.

- **cpu_count**. The number of CPUs.
- **cpu_type** and **cpu_revision.** The type constant and revision number of the CPUs.
- **cpu_infos**. An array of **cpu_info** structures, one for each CPU.
- **cpu_clock_speed**. The speed (in Hz) at which the CPUs operate.
- **bus_clock_speed**. The speed (in Hz) at which the bus operates.

- **max_***resource***s** and **used_***resource***s**. The five pairs of **max**/**used** fields give the total number of RAM pages, semaphores, and so on, that the system can create, and the number that are currently in use.

- **page_faults**. The number of times the system a read a page of memory into RAM due to a page fault.

- **volume**. The volume (listed by its ID) that contains the kernel.
- **directory**. The directory (listed by its ID) that contains the kernel.
- **name.** The file name of the kernel.

The **directory** field is unusable: Directory ID numbers aren't visible through the present (public) means of file system access. But you can save the directory IDs that you collect now and trade them in for a higher draft pick next season.

You retrieve a **system_info** structure through the **get_system_info()** function.

See also: **get_system_info()**

team_id

```
<kernel/OS.h>

typedef long team_id
```

The **team_id** type uniquely identifies teams.

See also: "Threads and Teams" on page 5

team_info

```
<kernel/OS.h>

typedef struct {
        team_id team;
        long thread_count;
        long image_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
        long argc;
        char args[64];
} team_info
```

The **team_info** structure holds information about a team. It's returned by the **get_team_info()** function. It's fields are:

- **team** is the team's ID number.

- **thread_count**, **image_count**, and **area_count** give the number of threads that have been spawned, images that have been loaded, and areas that have been created or cloned within this team.

- **debugger_nub_thread** and **debugger_nub_port** are used to communicate with the debugger. Unless you're designing your own debugger, you can ignore these fields.

- The **argc** field is the number of command line arguments that were used to launch the team; **args** is a copy of the first 64 characters from the command line invocation. If this team is an application that was launched through the user interface (by double-clicking, or by accepting a dropped icon), then **argc** is 1 and **args** is the name of the application's executable file.

See also: "Threads and Teams" on page 5

thread_entry

<kernel/OS.h>

typedef long (***thread_entry**)(void *)

The **thread_entry** type is a function protocol for functions that are used as the entry points for new threads. You assign an entry function to a thread when you all **spawn_thread()**; the function takes a **thread_entry** function as its first argument.

See also: "Threads and Teams" on page 5

thread_id

<kernel/OS.h>

typedef long **thread_id**

The **thread_id** type uniquely identifies threads.

See also: "Threads and Teams" on page 5

thread_info

<kernel/OS.h>

```
typedef struct {
        thread_id thread;
        team_id team;
        char name[B_OS_NAME_LENGTH];
        thread_state state;
        long priority;
        sem_id sem;
        double time;
        void *stack_base;
        void *stack_end;
} thread_info
```

This structure holds information about a thread. It's returned by functions such as **get_thread_info()**. The fields are:

- **thread**. The **thread_id** number of the thread.
- **team**. The **team_id** of the thread's team.
- **name**. The name assigned to the thread.
- **state**. A constant that describes what the thread is currently doing.
- **priority**. A constant that represents the level of attention the thread gets.
- **sem**. If the thread is waiting to acquire a semaphore, this is the **sem_id** number of that semaphore. The **sem** field is only valid if the thread's state is **B_THREAD_WAITING**.
- **time**. The amount of active attention the thread has received from the CPUs, measured in microseconds.
- **stack_base**. A pointer to the first byte of memory in the thread's execution stack.
- **stack_end**. A pointer to the last byte of memory in the thread's execution stack.

See also: "Threads and Teams" on page 5

thread_state

<kernel/OS.h>

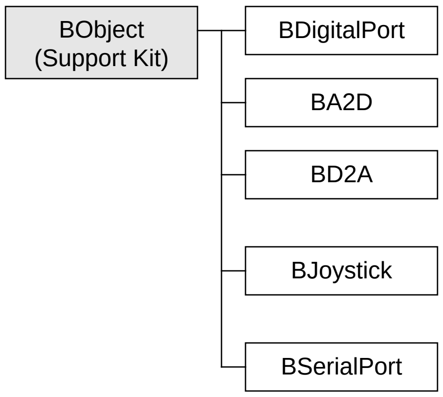
typedef enum { ... } **thread_state**

The **thread_state** type represents values that describe the various states that a thread can be in.

See also: **Thread State Constants**

9 The Device Kit

Device Kit Inheritance Hierarchy



9 The Device Kit

The Device Kit contains software for controlling various input/output devices and for writing your own device drivers. You'll find two kinds of software documented in this chapter:

- Encapsulated interfaces to some of the ports found on the back of the BeBox. Currently, this part of the kit contains five classes—BJoystick, BSerialPort, BDigitalPort, BA2D (analog to digital), and BD2A (digital to analog). A BJoystick object represents a joystick connection to the BeBox. A BSerialPort object can represent any of the four RS–232 serial ports that are visible on the back of the machine. The other three classes represent particular functions of the GeekPort™.

  These classes are all part of the shared system library, **libbe.so**. Their header files are collected in **DeviceKit.h** and are precompiled with the header files of other kits.

- The programming interfaces and protocols for developing your own drivers for input/output devices. All drivers are dynamically loaded, add-on modules that run as extensions either of the kernel or of a specific server. Most drivers run as part of the kernel, but drivers for graphics cards extend the Application Server and printer drivers connect to the Print Server.

  The programming interfaces for device drivers are *not* included in the master **DeviceKit.h** header file or the precompiled headers; this part of the Kit doesn't belong to the Be system library. A driver links only against its host module (or perhaps statically against a private library), not against the system library.

If you're interested in the interface to a joystick or serial port, you need read only about the BJoystick or BSerialPort class. If you're interested in the GeekPort interface, there's a small section that introduces the port and its three classes; look at it before turning to the particular class that interests you. If you're interesting in writing a device driver, skip the first part of the chapter and begin with "Developing a Device Driver" on page 39.

The GeekPort and its Classes

The GeekPort is a piece of hardware that communicates with external devices. Depending on how you use the GeekPort's ports, you can get up to 24 independent data paths:

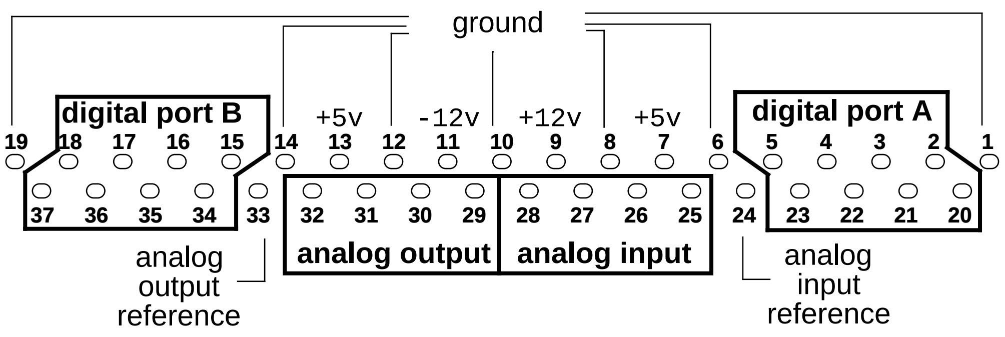
- Four 12-bit analog input channels.
- Four 8-bit analog output channels
- Two 8-bit wide digital ports (16 paths, total) that can act as inputs or outputs.

To provide high-level access to these data paths, the Device Kit defines three classes:

- The BA2D class ("analog to digital") lets you get at the analog input channels.
- The BD2A class ("digital to analog") does the same for the analog output channels.
- The BDigitalPort class lets you configure, read, and write the digital ports.

The signals and data that these classes read and write appear at the GeekPort connector, a 37-pin female connector that you'll find at the back of every BeBox. In addition to the pins that correspond to the analog and data paths, the GeekPort provides power and ground pins. Everything you need to feed your external gizmo is right there.

The GeekPort connector's pins are assigned thus:



The BA2D, BD2A, and BDigitalPort class descriptions re-visit this illustration to provide more detailed examinations of the specific GeekPort pins.

BA2D and BD2A

Derived from: public BObject

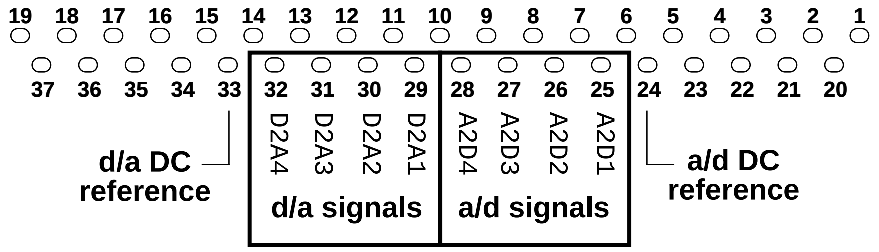
Declared in: <device/A2D.h>
<device/D2A.h>

Overview

The BA2D and BD2A classes let you talk to the GeekPort's analog-to-digital converter (ADC) and digital-to-analog converter (DAC). Before we examine the classes, let's visit the GeekPort.

The GeekPort

The GeekPort provides four channels of simultaneous analog-to-digital (a/d) and four channels of simultaneous digital-to-analog (d/a) conversion. The signals that feed the ADC arrive on pins 25-28 of the GeekPort connector; the signals that are produced by the DAC depart through pins 29-32 (as depicted below). Pins 24 and 33 are DC reference levels (ground) for the a/d and d/a signals, respectively (*don't* use pins 24 or 33 as power grounds):



In the illustration, the a/d and d/a pins are labelled ("A2D1", "A2D2", etc.) as they are known to the BA2D and BD2A classes.

If you've read the GeekPort hardware specification, you'll have discovered that the ADC can be placed in a few different modes (the DAC is less flexible). The BA2D and BD2A classes (more accurately, the ADC and DAC drivers) refine the GeekPort specification, as described in the following sections.

Note: Keep in mind that the a/d and d/a converters that the GeekPort uses are *not* part of the Crystal codec that's used by the audio software (and brought into your application through the Media Kit). The two sets of converters are completely separate and can be used independently and simultaneously. If you're doing on-board high-fidelity sound processing (or generation) in real time, you should stick with the Crystal convertors.

The ADC

The ADC accepts signals in the range [0, +4.096] Volts, performs a linear conversion, and spits out unsigned 12-bit data. The 4.096V to 12-bit conversion produces a convenient one-digital-step per milliVolt of input.

A/d conversion is performed on-demand: When you read a value from the ADC, the voltage that lies on the specified pin is immediately sampled (this is the "Single Shot" mode described in the GeekPort hardware specification). In other words, the ADC doesn't perform a sample and hold—it doesn't constantly and regularly measure the voltages at its inputs. Nonetheless, you *can't* retrieve samples at an arbitrarily high frequency simply by reading in a tight loop. This is because of the "sampling latency": When you ask for a sample, it takes the driver about ten milliseconds to process the request, not counting the (slight) overhead imposed by the C++ call (from your BA2D object). Therefore, the fastest rate at which you can get samples from the ADC is a bit less than 100 kHz.

Furthermore, the ADC driver "fakes" the four channels of a/d conversion. In reality, there's only one ADC data path; the driver multiplexes the path to create four independent signals. This means that the optimum 100 kHz sampling frequency is divided by the number of channels that you want to read. If all four channels are being read at the same time, you'll find that successive samples on a *particular* channel arrive slightly less often than once every 40 milliseconds (a rate of < 25 kHz).

Finally, the ADC hardware is shared by the GeekPort and the two joysticks. This cooperative use shouldn't affect your application—you can treat the ADC as if it were all your own—but this increases the multiplexing. In general, joysticks shouldn't need to sample very often, so while the theoretical "worst hit" on the ADC is a sample every 60 milliseconds, the reality should be much better. If we can assume that a joystick-reading application isn't oversampling, then the BA2D "sampling latency" should stay near the 10 milliseconds per channel measurement.

The DAC

The DAC accepts 8-bit unsigned data and converts it, in 16 mV steps, to an analog signal in the range [0, +4.080] Volts. Again, the quantization is linear. The DAC output isn't filtered; if you need to smooth the stair-step output, you have to build a filter into the gizmo that you're connecting to the GeekPort.

Each of the d/a pins is protected by an in-series 4.7 kOhm resistor; however, pin 33, the d/a DC reference (ground) pin, is not similarly impeded. If you want to attach an op-amp circuit to the DAC output, you should hang a 4.7 kOhm resistor on the ground pin that you're using.

When you write a digital sample to the DAC, the specified pin is immediately set to the converted voltage. The pin continues to produce that voltage until you write another sample.

Unlike the ADC, the DAC is truly a four-channel device, so there's no multiplexing imposition to slow things down. Furthermore, writing to the DAC is naturally faster than writing to the ADC. You should be able to write to the DAC as frequently as you want, without worrying about a hardware-imposed "sampling latency."

BA2D

The BA2D class lets you create objects that can read the GeekPort's a/d channels. Each BA2D object can read a single channel at a time; if you want to read all four channels simultaneously, you have to create four separate objects.

To retrieve a value from one of the a/d channels, you create a new BA2D object, open it on the channel you want (using the labels shown above), and then (repeatedly) invoke the object's **Read()** function. When you're through reading, you call **Close()** so some other object can open the channel.

Reading is a one-shot deal: For each **Read()** invocation, you get a single **ushort** that stores the 12-bit ADC value in its least significant 12 bits. To get a series of successive values, you have to put the **Read()** call in a loop. Keep in mind that there's no sampling rate or other automatic time tethering. For example, if you want to read the ADC every tenth of a second, you have to impose the waiting period yourself (by snoozing between reads, for example).

The outline of a typical a/d-reading setup is shown below:

```
#include <A2D.h>

void ReadA2D1()
{
    ushort val;
    BA2D *a2d = new BA2D();

    if (a2d->Open("A2D1") <= 0)
        return;

    while ( /* whatever */ ) {

        /* Read() returns the number of bytes that were
         * read; a successful read returns the value 2.
         */
        if (a2d->Read(&val) != 2)
            break;

        /* Apply val here. */
        ...
```

```
            /* Snooze for a bit. */
            snooze(1000);
        }
        a2d->Close();
        delete a2d;
    }
```

BD2A

Creating and using a BD2A object follows the same outline as shown above, but instead of reading a **ushort** value, you write a **uchar**. The **Write()** function returns 1 if successful:

```
    #include <D2A.h>

    void WriteD2A1()
    {
        uchar val;
        BD2A *d2a = new BD2A();

        if (d2a->Open("D2A1") <= 0)
            return;

        while ( /* whatever */ ) {
            /* Get an 8-bit value from somewhere. */
            val = ...;

            if (d2a->Write(val) != 1)
                break;

            snooze(1000);
        }
        d2a->Close();
        delete d2a;
    }
```

The DAC performs a “sample and hold”: The voltage that the DAC produces on a particular channel (and to which it sets the appropriate GeekPort pin) is maintained until another **Write()** call (on the same channel) changes the setting. Furthermore, the “hold” persists across BD2A objects: Neither closing nor deleting a BD2A object affects the voltage that’s produced by the corresponding GeekPort pin.

The BD2A class also implements a **Read()** function. This function returns the value that was most recently written to the particular DAC channel.

Constructor and Destructor

BA2D(), BD2A()

long **BA2D(**void**)**

long **BD2A(**void**)**

Creates a new object that can open an ADC or DAC channel (respectively). The particular channel is specified in a subsequent **Open()** call. Constructing a new BA2D or BD2A object doesn't affect the state of the ADC or DAC.

~BA2D(), ~BD2A()

virtual **~BA2D(**void**)**

virtual **~BD2A(**void**)**

Closes the channel that the object holds open (if any) and then destroys the object.

Important: Deleting a BD2A object *doesn't* affect the DAC channel's output voltage. If you want the voltage cleared (for example), you have to set it to 0 explicitly before deleting (or otherwise closing) the BD2A object.

Member Functions

Open(), IsOpen(), Close()

long **Open(**const char **name***)**

bool **IsOpen(**void**)**

void **Close(**void**)**

Open() opens the named ADC (BA2D) or DAC (BD2A) channel. The channel names (as you would pass them to **Open()**)are:

BA2D Channels	BD2A Channels
"A2D1"	"D2A1"
"A2D2"	"D2A2"
"A2D3"	"D2A3"
"A2D4"	"D2A4"

See the GeekPort connector illustration, above, for the correspondences between the channel names and the GeekPort connector pins.

Each channel can only be held open by one object at a time; you should close the channel as soon as you're finished with it. Furthermore, each BA2D or BD2A object can only hold one channel open at a time. When you invoke **Open()**, the channel that the object currently has open is automatically closed—even if the channel that you're attempting to open is the channel that the object already has open.

Opening an ADC or DAC channel doesn't affect the data in the channel itself. In particular, when you open a DAC channel, the channel's output voltage isn't changed.

Open() returns a positive integer if the channel is successfully opened; otherwise, it returns **B_ERROR**.

IsOpen() returns **TRUE** if the object holds its assigned channel open channel is successfully opened. Otherwise, it returns **FALSE**.

Close() does the obvious without affecting the state of the ADC or DAC channel. If you want to set a DAC channel's output voltage to 0 (for example), you must explicitly write the value before invoking **Close()**.

Read()

BA2D:

    long **Read(**ushort **adc_12_bit***)**

BD2A:

    long **Read(**uchar **dac_8_bit***)**

BA2D's **Read()** function causes the ADC to sample and convert (within a 12-bit range) the voltage level on the GeekPort pin that corresponds to the object's ADC channel. The 12-bit unsigned value is returned by reference in the *adc_12_bit* argument.

BD2A's **Read()** returns, by reference in *dac_8_bit*, the value that was most recently written to the object's particular DAC channel. The value needn't have been written by this object—it could have been written by the channel's previous opener.

Important: The BD2A **Read()** function returns a value that's cached by the DAC driver—it doesn't actually tap the GeekPort pin to see what value it's currently carrying. This should only matter to the clever few who will attempt (unsuccessfully) to use the d/a pins as input paths.

The object must open an ADC or DAC channel before calling **Read()**. The functions return **B_ERROR** if a channel isn't open, or if, for any other reason, the read failed. Otherwise they return the number of bytes that were read: 2 in the case of a BA2D, 1 for a BD2A object. Note that it's not an error to read the DAC before a value has been written to it.

Write()

BD2A only:

long **Write(**uchar *dac_8_bit***)**

Sends the *dac_8_bit* value to the object's DAC channel. The DAC converts the value to an analog voltage in the range [0, +4.080] Volts and sets the corresponding GeekPort pin. The pin continues to produce the voltage until another **Write()** call—possibly by a different BD2A object—changes the setting.

The DAC's conversion is linear: Each digital step corresponds to 16 mV at the output. The analog voltage midpoint, +2.040V, can be approximated by a digital input of 0x7F (which produces +2.032V) or 0x80 (+2.048V).

If the object isn't open, this function returns **B_ERROR**, otherwise it returns 1 (specifically, the number of bytes that were written).

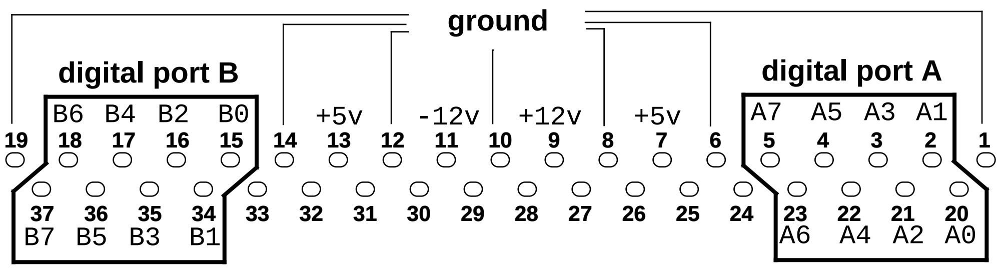
BDigitalPort

Derived from: public BObject

Declared in: <device/DigitalPort.h>

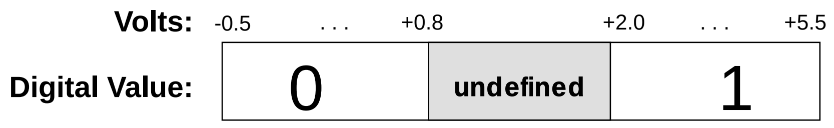
Overview

The BDigitalPort class is the programmer's interface to the GeekPort's two *digital ports*. Each digital port is an 8-bit wide device that can be set for input or output. The following illustration shows the disposition of the GeekPort connector pins as they are assigned to the digital ports:



Each pin in a digital port transmits the value of a single bit; the pins are labelled by bit position. Thus, A0 is the least significant bit of digital port A, and A7 is its most significant bit. You can use any of the seven ground pins (1, 6, 8, 10, 12, 14, and 19) in your digital port circuit. The unmarked pins (24-33) are the analog ports; see "BA2D and BD2A" on page 9 for more information on these ports.

Devices that you connect to the digital ports should send and (expect to) receive voltages that are below 0.8 Volts or above 2.0 Volts. These thresholds correspond, respectively, to the greatest value for digital 0 and the least for digital 1 (as depicted below). The correspondence to bit value for voltages between these limits is undefined.



Although there's no lower voltage limit for digital 0, nor upper limit for digital 1, the BeBox outputs voltages that are no less than 0 Volts, nor no more than +5 Volts. Your input device can exceed this range without damaging the BeBox circuitry: Excessive input emf is clipped to fall within [-0.5V, +5.5V].

Be aware that behind each digital port pin lies a 1 kOhm resistor.

BDigitalPort Objects

To access a digital port, you construct a BDigitalPort object, open it on the port you want, assign the object to work as either an input or an output, and then read or write a series of bytes from or to the object.

In the following example, we open and read from digital port A:

```
#include <DigitalPort.h>

void ReadDigitalPortA()
{
    char val;
    BDigitalPort *dPortA = new BDigitalPort();

    if (dPortA->Open("DigitalA") <= 0 ||
        dPortA->SetAsInput() != B_NO_ERROR) {
        ~dPortA;
        return;
    }

    while ( /* whatever */ ) {

        /* Read() returns the number of bytes that were
         * read; a successful read returns the value 1.
         */
        if (dPortA->Read(&val) != 1)
            break;

        /* Do something with the value. */
        ...

        /* Snooze for a bit. */
        snooze(1000);
    }
    dPortA->Close();
    delete dPortA;
}
```

As shown here, the BDigitalPort is constructed without reference to a specific port. It's not until you actually open the object (through **Open()**) that you have to identify the port that you want; identification is by name, "DigitalA" or "DigitalB". The **Read()** function returns only one value per invocation, and is untimed—if you don't provide some sort of tethering (as we do with **snooze()**, above) the read loop will spin as fast as possible.

To safeguard against an inadvertant burst of equipment-destroying output, the digital port is set to be an input when it's opened, and automatically reset to be an input when you close it.

Using Both Digital Ports at the Same Time

To access both digital ports at the same time, you have to construct two BDigitalPort objects. One of the objects can be used as an output and the other an input, both as outputs, or both as inputs.

In the following example, digital port A is used to write data to an external device, while digital port B is used for acknowledgement signalling: Before each write we set port B to 0, and after the write we wait for port B to be set to 1. We're assuming that the external device will write a 1 to port B when it's ready to receive the next 8-bits of data.

```
void WriteAndAck()
{
    char val;
    BDigitalPort *dPortA = new BDigitalPort();
    BDigitalPort *dPortB = new BDigitalPort();

    if (dPortA->Open("DigitalA") <= 0 ||
        dPortA->SetAsOutput() != B_NO_ERROR)
        goto error_tag;

    if (dPortB->Open("DigitalB") <= 0 ||
        dPortB->SetAsOutput() != B_NO_ERROR) {
        goto error_tag;

    while ( /* whatever */ ) {

        /* Clear the acknowledgement signal. */
        val = 0;
        if (dPortB->Write(&val) != 1)
            break;

        /* Reset val to the data we want to send. */
        val = ...;

        if (dPortA->Write(val) != 1)
            break;

        /* Reset digital port B to be an input. */
        if (dPortB->SetAsInput() != B_NO_ERROR)
            break;

        /* Wait for the acknowledgement. */
        while (1) {
            if (dPortB->Read(&val) != 1)
                goto error_tag;
            if (val == 1)
                break;
            snooze(1000);
        }
```

```
            /* Reset digital port B to be an output. */
            if (dPortB->SetAsOutput() != B_NO_ERROR)
                break;
        }
    error_tag:
        delete dPortA;
        delete dPortB;
    }
```

Notice that the acknowledgement signal only takes one bit of digital port B. This leaves seven bits that the external device can use to send additional data (triggers or gates, for example). The restriction in this scheme, given the structure shown above, is that this additional data would have to be synchronized with the acknowledgement signal.

By extension, if the data that you want to write to the external device is, at most, only seven-bits wide, then you could rewrite this example to use a single port: You would mask one of the bits as the acknowledgment carrier, and let the other seven bits carry the data, toggling the port between input and ouput as needed; the actual implementation is left as an exercise for the reader.

Overdriving an Output Pin

One of the features of the digital ports is that you can "overdrive" a pin from the outside. This means that you can set a port to be an output, and then force a voltage back onto the pin from an external device and read that voltage with the **Read()** function *without having to reset the port to be an input.* Keep in mind that there's a 1 KOhm resistor behind the pin (on the BeBox side), so your "overdrive" circuit has to be hot enough to balance the resistance.

When you overdrive an output pin, the voltage on the pin is altered for as long as the external force keeps it there. If you write an "opposing" value to an overdriven pin (through **Write()**), the written value won't pull the pin—the overdriven value will still be enforced. As soon as the overdrive voltage is removed, the pin will produce the voltage that was more recently written to it by the **Write()** function.

Constructor and Destructor

BDigitalPort()

long **BDigitalPort(**void**)**

Creates a new object that can open one of the digital ports. The particular port is specified in a subsequent **Open()** call.

~BDigitalPort

virtual **~BDigitalPort(**void**)**

Destroys the object, but not before closing the port that the object holds open (if any).

Deleting a BDigitalPort object sets the port (at the driver level) to be an input. The values at the port's pins are, at that point, undefined.

Member Functions

Open(), IsOpen(), Close()

long **Open(**const char **name***)**

bool **IsOpen(**void**)**

void **Close(**void**)**

Open() opens the named digital port; the *name* argument should be either "DigitalA" or "DigitalB". See the GeekPort illustration in the "Overview" section for the correspondences between the port names and the GeekPort connector pins.

A digital port can only be held open by one BDigitalPort object at a time; you should close the port as soon as you're finished with it. Furthermore, each BDigitalPort object can only hold one port open at a time. When you invoke **Open()**, the port that the object currently has open is automatically closed—even if the port that you're attempting to open is the port that the object already has open.

When you open a digital port, the device is automatically set to be an input. If you want the port to be an output, you must follow this call with a call to **SetAsOutput()**. Just to be safe, it couldn't hurt to explicitly set the port to be an input (through **SetAsInput()**) if that's what you want.

Open() returns a positive integer if the named port is successfully opened. Otherwise, it returns **B_ERROR**.

IsOpen() returns **TRUE** if the object currently has a port open, and **FALSE** if not.

Close() does the obvious. When a digital port is closed, it's set to be an input at the driver level.

Read()

long **Read(**char **buf***)**

Reads the data that currently lies on the digital ports pins, and returns this data as a single word in *buf*. Although you usually read a digital port that's been set to be an input, it's also possible to read an output port. In any case, the port must be open.

If the port was successfully read, the function returns 1 (the number of bytes read). Otherwise, it returns **B_ERROR**.

SetAsInput(), SetAsOutput(), IsInput(), IsOutput()

long **SetAsInput(**void**)**

long **SetAsOutput(**void**)**

bool **IsInput(**void**)**

bool **IsOutput(**void**)**

SetAsInput() and **SetAsOutput()** set the object's port to act as an input or output. They return **B_ERROR** if the object isn't open, and **B_NO_ERROR** otherwise.

IsInput() and **IsOutput()** return **TRUE** and **FALSE** much as you would expect them to.

Write()

long **Write(**char *value***)**

Sends *value* to the object's port. The port continues to produce the written data until another **Write()** call changes the setting.

The object must be open as an output for this function to succeed. Success is indicated by a return value of 1 (the number of bytes that were written). Failure returns **B_ERROR**.

BJoystick

Derived from:	public BObject
Declared in:	<device/Joystick.h>

Overview

A BJoystick object provides an interface to a joystick connected to the BeBox. There are two joystick ports on the back of the machine, one above the other. With the aid of a simple Y connector, each of them can support two joysticks for a total of four ports.

Unlike the event and message-driven interface to the mouse and keyboard, the interface to a joystick is strictly demand-driven. An application must repeatedly poll the state of the joystick by calling the BJoystick object's **Update()** function. **Update()** queries the port and updates the object's data members to reflect the current state of the joystick.

Data Members

double **timestamp**	The time of the most recent update, as measured in microseconds from the time the machine was last booted.
short **horizontal**	The horizontal position of the joystick at the time of the last update.
short **vertical**	The vertical position of the joystick at the time of the last update.
bool **button1**	**TRUE** if the first button was pressed at the time of the last update, and **FALSE** if not.
bool **button2**	**TRUE** if the second button was pressed at the time of the last update, and **FALSE** if not.

horizontal and **vertical** values can range from 0 through 4,095, but joysticks typically don't use the full range and some don't register all values within the range that is used. The scale is not linear—identical increments in different parts of the range can reflect differing amounts of horizontal and vertical movement. The exact variance from linearity and the extent of the usable range are partly characteristics of the individual joystick and partly functions of the BeBox hardware < which will be more fully documented in a later release >.

Constructor and Destructor

BJoystick()

BJoystick(void**)**

Initializes the BJoystick object so that all values are set to 0. Before using the object, you must call **Open()** to open a particular joystick port. For the object to register any meaningful values, you must call **Update()** to query the open port.

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

~BJoystick()

virtual **~BJoystick(**void**)**

Closes the port, if it was not closed already.

Member Functions

Open(), Close()

long **Open(**const char **name***)**

void **Close(**void**)**

These functions open the *name* joystick port and close it again. There are two ports on the back panel of the BeBox, and they have names that correspond to their labels on the machine (and in *The Be User's Guide* diagram):

“joystick1” (on the top)
“joystick2” (on the bottom)

By attaching a Y cable to a machine port, you can make it support two joysticks. Cables, therefore, add two additional ports:

“joystick3” (on the top)
“joystick4” (on the bottom)

The cable maps the bottom row of pins on a machine port to the top row on a cable port. Therefore, the first two names listed above correspond to the top row of pins on a machine port; the last two names correspond to the bottom row of pins.

If it's able to open the port, **Open()** returns a positive integer. If unable or if the *name* isn't valid, it returns **B_ERROR**. If the *name* port is already open, **Open()** tries to close it first, then open it again.

To be able to obtain joystick data, a BJoystick object must have a port open.

Update()

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

Updates the data members of the object so that they reflect the current state of the joystick. An application would typically call **Update()** periodically to poll the condition of the device, then read the values of the data members.

This function returns **B_ERROR** if the BJoystick object doesn't have a port open, and **B_NO_ERROR** if it does.

BSerialPort

Derived from: public BObject

Declared in: <device/SerialPort.h>

Overview

A BSerialPort object represents an RS-232 serial port connection to the BeBox. There are four such ports on the back of the machine.

Through BSerialPort functions, you can read data received at a serial port and write data over the connection. You can also configure the connection—for example, set the number of data and stop bits, determine the rate at which data is sent and received, and select the type of flow control (hardware or software) that should be used.

To read and write data, a BSerialPort object must first open one of the serial ports by name. For example:

```
BSerialPort *connection = new BSerialPort;
if ( connection->Open("serial2") > 0 ) {
    . . .
}
```

The BSerialPort object communicates with the driver for the port it has open. The driver maintains an input buffer of 1K bytes to collect incoming data and an output buffer half that size to hold outgoing data. When the object reads and writes data, it reads from and writes to these buffers.

Constructor and Destructor

BSerialPort()

BSerialPort(void**)**

Initializes the BSerialPort object to the following default values:

- Hardware flow control (see **SetFlowControl()**)
- A data rate of 19,200 bits per second (see **SetDataRate()**)
- A serial unit with 8 bits of data, 1 stop bit, and no parity (see **SetDataBits()**)
- Blocking, but with a timeout of 0.0 microseconds, for reading data (see **Read()**)

The new object doesn't represent any particular serial port. After construction, it's necessary to open one of the ports by name.

The type of flow control must be decided before a port is opened. But the other default settings listed above can be changed before or after opening a port.

See also: **Open()**

~BSerialPort()

virtual **~BSerialPort(**void**)**

Makes sure the port is closed before the object is destroyed.

Member Functions

ClearInput(), ClearOutput()

void **ClearInput(**void**)**

void **ClearOutput(**void**)**

These functions empty the serial port driver's input and output buffers, so that the contents of the input buffer won't be read (by the **Read()** function) and the contents of the output buffer (after having been written by **Write()**) won't be transmitted over the connection.

The buffers are cleared automatically when a port is opened.

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

Close() *see* Open()

DataBits() *see* SetDataBits()

DataRate() *see* SetDataRate()

FlowControl() *see* SetFlowControl()

IsCTS()

bool **IsCTS(**void**)**

Returns **TRUE** if the Clear to Send (CTS) pin is asserted, and **FALSE** if not.

IsDCD()

bool **IsDCD(**void**)**

Returns **TRUE** if the Data Carrier Detect (DCD) pin is asserted, and **FALSE** if not.

IsDSR()

bool **IsDSR(**void**)**

Returns **TRUE** if the Data Set Ready (DSR) pin is asserted, and **FALSE** if not.

IsRI()

bool **IsRI(**void**)**

Returns **TRUE** if the Ring Indicator (RI) pin is asserted, and **FALSE** if not.

Open(), Close()

long **Open(**const char **name***)**

void **Close(**void**)**

These functions open the *name* serial port and close it again. Ports are identified by names that correspond to their labels on the back panel of the BeBox:

“serial1”
“serial2”
“serial3”
“serial4”

To be able to read and write data, the BSerialPort object must have a port open. It can open first one port and then another, but it can have no more than one open at a time. If it already has a port open when **Open()** is called, that port is closed before an attempt is made to open the *name* port. (Thus, both **Open()** and **Close()** close the currently open port.)

Open() can’t open the *name* port if some other entity already has it open. (If the BSerialPort itself has *name* open, **Open()** first closes it, then opens it again.)

If it’s able to open the port, **Open()** returns a positive integer. If unable, it returns **B_ERROR**.

When a serial port is opened, its input and output buffers are emptied and the Data Terminal Ready (DTR) pin is asserted.

See also: **Read()**

ParityMode() ***see*** **SetDataBits()**

Read(), SetBlocking(), SetTimeout()

long **Read(**void **buffer*, long *maxBytes***)**

void **SetBlocking(**bool *shouldBlock***)**

void **SetTimeout(**double *timeout***)**

Read() takes incoming data from the serial port driver and places it in the data *buffer* specified. In no case will it read more than *maxBytes*—a value that should reflect the capacity of the *buffer*; it returns the actual number of bytes read. **Read()** fails if the BSerialPort object doesn't have a port open.

The number of bytes that **Read()** reads before returning depends not only on *maxBytes*, but also on the *shouldBlock* flag and the *timeout* set by the other two functions.

SetBlocking() determines whether **Read()** should block and wait for *maxBytes* of data to arrive at the serial port if that number isn't already available to be read. If the *shouldBlock* flag is **TRUE**, **Read()** will block. However, if *shouldBlock* is **FALSE**, **Read()** will take however many bytes are waiting to be read, up to the maximum asked for, then return immediately. If no data is waiting at the serial port, it returns without reading anything.

SetTimeout() sets a time limit on how long **Read()** will block while waiting for data to arrive at the input buffer. The *timeout* is relevant to **Read()** only if the *shouldBlock* flag is **TRUE**. (However, the time limit also applies to the **WaitForInput()** function, which always blocks if the limit is greater than 0.0, regardless of the *shouldBlock* flag.)

The *timeout* is expressed in microseconds and is limited to 25,500,000.0 (25.5 seconds); it's set to the maximum value if a greater amount of time is specified. Differences less than 100,000.0 microseconds (0.1 second) are not recognized; they're rounded to the nearest tenth of a second. If the *timeout* is set to 0.0 microseconds, **Read()** (and **WaitForInput()**) will not block.

The default *shouldBlock* setting is **TRUE**, but the default *timeout* is 0.0, which prevents blocking in any case. < In future releases, the default timeout will be an infinite amount of time; it won't impose a time limit on blocking. >

Like the standard **read()** system function, **Read()** returns the number of bytes it succeeded in placing in the *buffer*, which may be 0. It returns **B_ERROR** (–1) if there's an error of any kind—for example, if the BSerialPort object doesn't have a port open. It's not considered an error if a timeout expires.

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

SetBlocking() ***see*** **Read()**

SetDataBits(), SetStopBits(), SetParityMode(), DataBits(), StopBits(), ParityMode()

void **SetDataBits(**data_bits *count***)**

void **SetStopBits(**stop_bits *count***)**

void **SetParityMode(**parity_mode *mode***)**

data_bits **DataBits(**void**)**

stop_bits **StopBits(**void**)**

parity_mode **ParityMode(**void**)**

These functions set and return characteristics of the serial unit used to send and receive data. **SetDataBits()** sets the number of bits of data in each unit. The *count* can be:

B_DATA_BITS_7 or
B_DATA_BITS_8

The default is **B_DATA_BITS_8**.

SetStopBits() sets the number of stop bits in each unit. It can be:

B_STOP_BITS_1 or
B_STOP_BITS_2

The default is **B_STOP_BITS_1**.

SetParityMode() sets whether the serial unit contains a parity bit and, if so, the type of parity used. The mode can be:

B_EVEN_PARITY,
B_ODD_PARITY, or
B_NO_PARITY

The default is **B_NO_PARITY**.

SetDataRate(), DataRate()

void **SetDataRate(**data_rate *bitsPerSecond***)**

data_rate **DataRate(**void**)**

These functions set and return the rate (in bits per second) at which data is both transmitted and received. Permitted values are:

B_0_BPS	**B_200_BPS**	**B_4800_BPS**
B_50_BPS	**B_300_BPS**	**B_9600_BPS**
B_75_BPS	**B_600_BPS**	**B_19200_BPS**
B_110_BPS	**B_1200_BPS**	**B_38400_BPS**
B_134_BPS	**B_1800_BPS**	**B_57600_BPS**
B_150_BPS	**B_2400_BPS**	**B_115200_BPS**

The default data rate is **B_19200_BPS**. If the rate is set to 0 (**B_0_BPS**), data will be sent and received at an indeterminate number of bits per second.

SetDTR()

long **SetDTR(**bool *pinAsserted***)**

Asserts the Data Terminal Ready (DTR) pin if the *pinAsserted* flag is **TRUE**, and de-asserts it if the flag is **FALSE**.

See also: **SetRTS()**

SetFlowControl(), FlowControl()

void **SetFlowControl(**ulong *mask***)**

ulong **FlowControl(**void**)**

These functions set and return the type of flow control the driver should use. There are two possibilities:

B_SOFTWARE_CONTROL	Control is maintained through XON and XOFF characters inserted into the data stream.
B_HARDWARE_CONTROL	Control is maintained through the Clear to Send (CTS) and Request to Send (RTS) pins.

The *mask* passed to **SetFlowControl()** and returned by **FlowControl()** can be just one of these constants—or it can be a combination of the two, in which case the driver will use both types of flow control together. It can also be 0, in which case the driver won't use any flow control. **B_HARDWARE_CONTROL** is the default.

SetFlowControl() should be called before a specific serial port is opened. You can't change the type of flow control the driver uses in midstream.

SetParityMode() *see* SetDataBits()

SetRTS()

long **SetRTS(**bool *pinAsserted***)**

Asserts the Request to Send (RTS) pin if the *pinAsserted* flag is **TRUE**, and de-asserts it if the flag is **FALSE**.

See also: **SetDTR()**

SetStopBits() *see* **SetDataBits()**

SetTimeout() *see* **Read()**

StopBits() *see* **SetDataBits()**

WaitForInput()

long **WaitForInput(**void**)**

Waits for input data to arrive at the serial port and returns the number of bytes available to be read.

If data is ready to be read when this function is called, it immediately returns without blocking and reports how many bytes there are. If data hasn't arrived, it blocks and waits for the first bytes to be transmitted. When they're detected, it immediately reports how many have arrived.

This function doesn't respect the flag set by **SetBlocking()**; it blocks even if blocking is turned off for the **Read()** function. However, it does respect the timeout set by **SetTimeout()**. If the timeout expires before input data arrives at the serial port, it returns 0. A timeout of 0.0 microseconds doesn't give **WaitForInput()** enough time to block; it returns immediately.

See also: **Read()**

Write()

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

Writes up to *numBytes* of *data* to the serial port's output buffer. This function will be successful in writing the data only if the BSerialPort object has a port open. The output buffer holds a maximum of 512 bytes.

Like the **write()** system function, **Write()** returns the actual number of bytes written, which will never be more than *numBytes*, and may be 0. If it fails (for example, if the BSerialPort object doesn't have a serial port open) or if it's interrupted before it can write anything, it returns **B_ERROR** (–1).

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

Constants and Defined Types

This section lists all the constants and types defined for the BJoystick, BSerialPort, BDigitalPort, BA2D, and BD2A classes—though, in fact, only the BSerialPort class relies on any defined constants or types. Everything listed here is explained more fully in the descriptions of the member functions of that class.

Constants

data_bits Constants

<device/SerialPort.h>

Enumerated constant
B_DATA_BITS_7
B_DATA_BITS_8

These constants name the possible number of data bits in a serial unit.

See also: **BSerialPort::SetDataBits()**

data_rate Constants

<device/SerialPort.h>

Enumerated constant	Enumerated constant
B_0_BPS	**B_1200_BPS**
B_50_BPS	**B_1800_BPS**
B_75_BPS	**B_2400_BPS**
B_110_BPS	**B_4800_BPS**
B_134_BPS	**B_9600_BPS**
B_150_BPS	**B_19200_BPS**
B_200_BPS	**B_38400_BPS**
B_300_BPS	**B_57600_BPS**
B_600_BPS	**B_115200_BPS**

These constants give the possible rates—in bits per second (bps)—at which data can be transmitted and received over a serial connection.

See also: **BSerialPort::SetDataRate()**

Flow Control Constants

<device/SerialPort.h>

Enumerated constant

B_SOFTWARE_CONTROL
B_HARDWARE_CONTROL

These constants form a mask that records the method(s) of flow control the serial port driver should use.

See also: **BSerialPort::SetFlowControl()**

parity_mode Constants

<device/SerialPort.h>

Enumerated constant

B_NO_PARITY
B_ODD_PARITY
B_EVEN_PARITY

These constants list the possibilities for parity when transmitting data over a serial connection.

See also: **BSerialPort::SetDataBits()**

stop_bits Constants

<device/SerialPort.h>

Enumerated constant

B_STOP_BITS_1
B_STOP_BITS_2

These constants name the possible number of stop bits in a serial unit.

See also: **BSerialPort::SetDataBits()**

Defined Types

data_bits

<device/SerialPort.h>

typedef enum { . . . } **data_bits**

This type is used to set and return the number of data bits in a serial unit.

See also: “**data_bits** Constants” above, **BSerialPort::SetDataBits()**

data_rate

<device/SerialPort.h>

typedef enum { . . . } **data_rate**

This type is used to set and return the rate at which data is sent and received through a serial connection.

See also: “**data_rate** Constants” above, **BSerialPort::SetDataRate()**

parity_mode

<device/SerialPort.h>

typedef enum { . . . } **parity_mode**

This type is used to set and return the type of parity that should be used when sending and receiving data.

See also: “**parity_mode** Constants” above, **BSerialPort::SetDataBits()**

stop_bits

<device/SerialPort.h>

typedef enum { . . . } **stop_bits**

This type is used to set and return the number of stop bits in a serial unit.

See also: “**stop_bits** Constants” above, **BSerialPort::SetDataBits()**

Developing a Device Driver

A device driver ties an input/output hardware device to the computer's operating system. To develop a driver, you have to know about both ends of that link:

- On the one hand, you need to be thoroughly familiar with the hardware device and its particular interface.
- On the other hand, you must understand the operating system and the demands it places on the driver.

Hardware specifications and manuals can provide you with the first kind of information; this documentation can help only with the second—that is, with information specific to the Be operating system. On the next page, you'll see a list of recommended documentation for the DMA controller, the PCI bus, and other hardware found inside the BeBox. This book is concerned solely with how a driver must be structured to work with Be system software.

Overview

On the BeBox, device drivers run as dynamically loaded add-on modules—as extensions of a host component of the operating system. The Application Server, the part of the operating system that's responsible for all graphics operations, is the host for graphics card drivers. The Print Server hosts drivers for printers. The kernel acts as the host for all other drivers. The kernel and the two servers load drivers on demand, and can unload them when they're no longer needed.

Because drivers are linked to their hosts and run in the host's address space, they must play by the host's rules. The kernel and servers impose three different kinds of restrictions on loadable drivers:

- A driver must be constructed so that it can respond to its host. It has to be able to inform the host of the device or devices it drives, and it has to provide functions that the host can call to operate the driver. As an add-on module, a driver lacks an independent main thread of execution (and a **main()** function). Instead, it provides the host with entry points to driver functionality and responds only to the host's instructions.
- A driver can call only those functions that it implements itself or that the host makes available to it. It cannot, for example, link against the shared system library and call any function it wants from the Kernel Kit or Storage Kit. It might statically link against a private library, but it typically links only to the host and is limited to calling functions that the host exports.

- The driver must be compiled as an add-on module and it must be installed in a directory where the host expects to find it.

Although the kernel, the Application Server, and the Print Server all impose these three kinds of restrictions on their loadable drivers, they each impose a different set of restrictions. The kernel's rules are not the Application Server's rules, and the Application Server's are not the same as the Print Server's.

To learn the rules that apply to the type of driver you intend to develop, begin with the section listed in the following chart:

To develop:	Go to:
A driver for a graphics card	"Developing a Driver for a Graphics Card" on page 77
A driver for a printer	< Wait until the next release, or contact Be developer support. The interface for printer drivers is under development and not yet documented. >
All other drivers	"Developing a Kernel-Loadable Driver" on page 41

Recommended Reading

For information on the PCI bus:

PCI Local Bus Specification, revision 2.1, June 1, 1995, PCI Special Interest Group, PO Box 14070, Portland OR 97214, (800) 433-5177 or (503) 797-4207

For information on the ISA bus, ISA 8259 interrupt controller, and ISA-standard 8237 DMA controller:

82378 System I/O (SIO), August 1994, order number 290473-004, Intel Corporation, 2200 Mission College Boulevard, PO Box 58119, Santa Clara, CA 95052

For information on the SCSI common access method (CAM):

Draft Proposed American National Standard *SCSI-2 Common Access Method Transport and SCSI Interface Module,* ASC X3T–10, revision 12, December 14, 1994, American National Standards Institute, 11 West 42nd Street, New York, NY 10036

Developing a Kernel-Loadable Driver

At the most basic level, devices (other than graphics cards) are controlled by system calls that the kernel traps and translates for the driver. Five different kinds of functions control input/output devices on the BeBox:

open()	Opens a device for reading or writing.
close()	Closes a device that was previously opened.
read()	Reads data from the device.
write()	Writes data to the device.
ioctl()	Formats, initializes, queries, and otherwise controls the device.

All these functions, except **ioctl()**, are Posix-compliant. Instead of **ioctl()**, Posix defines a set of functions like **tcsetattr()** and **tcflush()** to control data terminals. These functions are supported, but they can be treated as special cases of **ioctl()**. (Posix also defines a **fcntl()** function for file control that has the same syntax as **ioctl()**.)

When **open()**, **close()**, **read()**, **write()**, or **ioctl()** is called for a device, the kernel expects a driver to do the work that's required. Each driver must implement a set of functions that correspond directly to the five system calls. Everything the driver does to operate the device is initiated through one of these functions.

Because drivers run in the kernel's address space as extensions of the kernel, they must conform to the kernel's expectations. Separate sections discuss the three types of restrictions that the kernel imposes on drivers:

- "Entry Points" on page 42 describes how drivers must be constructed to work with the kernel. The driver must provide the kernel with entry points to its functionality and follow the kernel's instructions.
- "Exported Functions" on page 49 discusses the kinds of functions that the kernel exports for drivers. These include some from the Kernel Kit, Support Kit, and standard C library, and some that are defined especially for drivers. The driver is limited to calling functions that it itself implements or that the kernel exports.
- "Installation" on page 54 discusses how to compile a driver and install it on the BeBox. The driver must be installed where the kernel can find it.

The exported functions that the kernel defines specifically for drivers are documented following these three sections.

Entry Points

The kernel loads a device driver when it's needed—typically when someone first attempts to open the device for reading or writing. Opening a device is a prerequisite to using it.

To the **open()** function, drivers are identified by a fictitious pathname beginning with **/dev/**. For example, this code opens the parallel port driver for writing:

```
int fd = open("/dev/parallel", O_WRONLY);
```

The first thing the kernel must do is match the device name—"/dev/parallel" in this case—to a driver; it must find a driver for the device. The driver might be one that has already been loaded, or it might be one that the kernel must search for and load. Loadable drivers reside in the **/system/drivers** directory; this is where the kernel looks for drivers and where they all must be installed.

Once a driver has been located and loaded, the kernel begins communicating with it—first to get information from it and test whether it's the right driver, then to initialize it and have it open the device.

One key piece of information that the kernel needs from the driver is the names of all its devices. Another is a list of the functions it can call to exercise those devices. For each device, the driver implements a set of hook functions that open the device, control it, read data from it, write data to it, and perhaps eventually close it. These hooks correspond to the system functions discussed above.

To give the kernel initial access to this information, drivers make declarations—of functions and data structures—using names the kernel will look for. Five such names will be discussed in the following sections:

init_driver()	Initializes the driver after it's loaded.
uninit_driver()	Cleans up after the driver before it's unloaded.
devices	Declares the devices and their hook functions.
publish_device_names()	Lists the devices the driver handles.
find_device_entry()	Associates a device with its hook functions.

These are the main entry points for driver control.

Driver Initialization

Immediately after loading a driver, the kernel gives it a chance to initialize itself. If the driver implements a function called **init_driver()**, the kernel will call it before proceeding with anything else—before asking the driver to open a device. The function should expect no arguments and return either **B_ERROR** or **B_NO_ERROR**:

long **init_driver(**void**)**

A return of **B_ERROR** means that the driver can't continue; the kernel will consequently unload it. A return of **B_NO_ERROR** means that all is well; the kernel will continue by asking the driver to open a device. The absence of an **init_driver()** function is equivalent to a return of **B_NO_ERROR**.

init_driver() might go a long way toward initializing the data structures the driver uses to do its work—for example, setting up needed semaphores. However, details specific to a particular device should be left to the hook function that opens that device.

When the kernel is about to get rid of a driver, it gives the driver a chance to undo what **init_driver()** did. If the driver implements a function called **uninit_driver()**, the kernel will call it immediately before unloading the driver. This function has the same syntax as the initialization function:

long **uninit_driver(**void**)**

This function can do nothing to prevent the driver from being unloaded. It should simply clean up after the driver—for example, delete semaphores—and return **B_NO_ERROR**.

Driver initialization and its opposite happen just once—when the driver is loaded and unloaded. In contrast, devices might be opened and closed many times while the driver continues to reside in the kernel.

Device Declarations

For the kernel to find the driver for a given device, all drivers must declare the names of the devices they control. For the kernel to be able to communicate with the driver to operate the device, every driver must declare a set of device-specific hook functions the kernel can call.

These declarations are made in a **device_entry** structure that maps the device name to the set of hook functions. This structure is declared in **device/Drivers.h** and contains the following fields:

const char ***name**	The name of the device—for example, "/dev/serial". This is the same name that's passed to **open()**. Driver code can assign any name it wants to the device, but it must begin with the "/dev/" prefix, which distinguishes devices from ordinary files.
device_open_hook **open**	The function that the kernel should call to open the device. The kernel will invoke this function to respond to **open()** system calls.
device_close_hook **close**	The function that should be called to close the device. It corresponds to the **close()** system call.
device_control_hook **control**	The function that the kernel should call to control the device, including querying the driver for information about it. The kernel will invoke this function to respond to **ioctl()** calls.

device_io_hook **read**	The function that should be called to read data from the device. The kernel will invoke this function to respond to the **read()** system call.
device_io_hook **write**	The function that should be called to write data to the device. It corresponds to the **write()** system call.

(The five functions are described in more detail under "Hook Functions" below.)

A driver declares one **device_entry** structure for each device it can drive. If it can handle more than one device, it must provide a **device_entry** structure for each one. If it permits a device to be referred to by more than one name, it must provide a structure for each name it recognizes.

There are two ways for a driver to provide the kernel with the information in a **device_entry** structure: If the list of devices is known at compile time, the driver can declare them statically. If the list might change at run time, it can return them dynamically.

Static Drivers

Most drivers are designed to handle a fixed set of known devices—perhaps a single device or perhaps many. Such drivers should declare a null-terminated array of **device_entry** structures under the global name **devices**:

device_entry **devices**[]

For example, the serial port driver might declare a **devices** array that looks like this:

```
device_entry devices[7] = {
    {"/dev/serial1", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial2", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial3", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/serial4", open_func, close_func, control_func,
            read_func, write_func},
    {"/dev/com3", open_com_func, close_func, control_com_func,
            read_func, write_func},
    {"/dev/com4", open_com_func, close_func, control_com_func,
            read_func, write_func},
    0
};
```

In this case, the driver handles the four serial ports seen on the back of the BeBox, each of which it identifies by a different name. It can also control "com3" and "com4" ports on an add-on board.

As this example illustrates, the hook functions declared in a **device_entry** structure are specific to the device. For the most part, the serial port driver above uses the same set of functions to operate all the devices, but declares special functions for opening and controlling “com3” and “com4”.

Note also that the array is null terminated.

Dynamic Drivers

A driver can also provide **device_entry** information dynamically. Instead of a **devices** array, it implements two functions, **publish_device_names()** and **find_device_entry()**.

The first of these functions should be declared as follows:

> char ****publish_device_names(**const char **deviceName***)**

If passed a proposed *deviceName* that matches the name of a device the driver handles, or if passed a **NULL** device name, this function should return a null-terminated array of all the names of all the devices that it handles. For example, the serial port driver described above would return the following array:

```
"/dev/serial1",
"/dev/serial2",
"/dev/serial3",
"/dev/serial4",
"/dev/com3",
"/dev/com4",
0
```

However, if the proposed device name doesn’t match any that the driver handles, **publish_device_names()** should return **NULL**.

While **publish_device_names()** informs the kernel of the devices that the driver handles, **find_device_entry()** returns entry information about a particular device. It has the following form:

> device_entry ***find_device_entry(**const char **deviceName***)**

If passed the name of a device that the driver knows about, this function should return the **device_entry** for that name. If the *deviceName* doesn’t match one of the driver’s devices, it should return **NULL**.

The kernel first calls **publish_device_names()** during the boot sequence to find what devices the driver handles. It may call the function again to update the list when it tries to match a driver to a specific device. If a match is made, it calls **find_device_entry()** to get the list of hook functions for the device.

Hook Functions

The five hook functions that are declared in a **device_entry** structure can have any names that you want to give them, provided that they don't clash with names that the kernel exports (see "Exported Functions" on page 49). However, their syntax is strictly prescribed by the kernel (through type definitions found in **device/Drivers.h**).

The five functions have two points in common: First, they each return an error code, which should be 0 (**B_NO_ERROR**) if there is no error. The error value is passed through as the return value for the **open()**, **close()**, **read()**, **write()**, or **ioctl()** system call that caused the kernel to invoke the driver function. The driver should return error values that are compatible with ones that are expected from those functions.

Second, all five functions are passed information identifying the device. As their first argument, they receive a pointer to a **device_info** structure (also defined in **device/Drivers.h**), which contains just two fields:

device_entry ***entry**	The **device_entry** structure for the device that's being operated on. This is a copy of information that the driver declared in its **devices** array or that its **find_device_entry()** function returned.
void ***private_data**	Arbitrary data that describes the device. This data is a way for the driver to record information about the device and have it persist between function calls. Although the kernel stores this data and passes it to the driver, the driver initializes it and maintains it; the kernel doesn't query or modify it.

Thus, all of the device-specific hook functions have the same return types and initial arguments:

long **function(**device_info **info, . . .***)**

Differences among the functions are discussed below.

Opening and Closing a Device

The function that opens a device is of type **device_open_hook** and the one that closes it is of type **device_close_hook**. They're defined as follows:

typedef long (***device_open_hook**)**(**device_info **info*, ulong *flags***)**

typedef long (***device_close_hook**)**(**device_info **info***)**

The *flags* mask that's passed to the **open()** system call is passed through to the device function. It typically will contain a flag like **O_RDONLY**, **O_RDWR**, or **O_WRONLY**.

Since the hook function that opens the device is the first one that's called, it might set up the **device_info** description of the device (to the extent that **init_driver()** hasn't already done so). It might also use that description, or some static data, to record whether or not the device is currently open. Typically, only one process can have a device open at a time. If the hook function sees that the device is already open, it can refuse to open it again.

Whatever values these functions return will also be returned by the **open()** and **close()** system calls.

Reading and Writing Data

The driver functions that read data from and write data to a device must be of type **device_io_hook**, which is defined as follows:

> typedef long (***device_io_hook**)**(**device_info **info*,
> void **buffer*, ulong *numBytes*, ulong *position***)**

The function that reads data from the device should place up to *numBytes* of data into the specified *buffer*. The hook that writes to the device should take *numBytes* of data from the *buffer*. These functions should read and write the data beginning at the *position* offset on the device. The offset is meaningful for some drivers (mostly drivers for storage devices), but can be ignored by others (such as a serial port driver).

Whatever values these functions return will also be returned by the **read()** and **write()** system calls.

Controlling the Device

The hook function that initializes, formats, queries, and otherwise controls a device is of type **device_control_hook**, defined as follows:

> typedef long (***device_control_hook**)**(**device_info **info*, ulong *op*, void **data***)**

The second argument, *op*, is a constant that specifies the particular control operation that the function should perform. The third argument, *data*, points either to some information that the control function needs to carry out the *op* operation or to a data structure that it should fill in with information that the operation requests. The interpretation of the data pointer depends entirely on the nature of the operation and will differ from operation to operation; the *op* and *data* arguments go hand-in-hand.

For example, if the *op* code is **SET_CONFIG**, *data* might point to a structure with values that the control function should use to re-configure the device. If the operation is **GET_ENABLED_STATE**, *data* might point to an integer that the function would be expected to set to either 1 or 0. If it's **RESTART**, *data* might simply be **NULL**.

The kernel defines a number of control operations (which are explained in the next section). These are operations that the kernel might call upon any driver to perform.

If you define your own control operations (for an **ioctl()** call on your driver), you should be sure that they aren't confused with any that the kernel currently defines—or any that it will define in the future. We pledge that all system-defined control constants will have values below **B_DEVICE_OP_CODES_END**. The constants you define should be increments above this value. For example:

```
enum {
    REPORT_STATUS = B_DEVICE_OP_CODES_END + 1,
    SET_TIMER,
    . . .
}
```

If a control function doesn't recognize the *op* code it's passed or can't perform the requested operation, it should return **B_ERROR** (–1).

Control Operations

Several control operations are defined by the kernel. The kernel can request any driver to perform these operations, even in the absence of an **ioctl()** call. A control function should respond to as many of these requests as it can. It should respond to inappropriate or unrecognized requests by returning **B_ERROR**.

The set of system-defined control operations is described below.

B_GET_SIZE and B_SET_SIZE

These control operations request the driver to get and set the memory capacity of the physical device. The capacity is measured in bytes and is recorded as a **ulong** integer. For a **B_GET_SIZE** request, the control function should write this number to the location referred to by the *data* pointer. For **B_SET_SIZE**, *data* will be the requested number of bytes (not a pointer to it).

B_SET_BLOCKING_IO and B_SET_NONBLOCKING_IO

These operations determine whether or not the driver should block when reading and writing data. **B_SET_BLOCKING_IO** requests the driver to put itself in blocking mode. Its read function should wait for data to arrive if none is readily available and its write function should wait for the device to be ready to accept data if it's not immediately free to take it. If **B_SET_NONBLOCKING_IO** is requested, the read function should return immediately if there is no data available to read and the write function should return immediately if the device isn't ready to accept written data.

For these operations, the *data* argument doesn't contain a meaningful value.

B_GET_READ_STATUS and B_GET_WRITE_STATUS

These control operations request the driver to report whether or not it's ready to read and write without blocking. The control function should respond by placing **TRUE** or **FALSE** as a **ulong** integer in the location that *data* points to.

For **B_GET_READ_STATUS**, it should respond **TRUE** if there's data waiting to be read, and **FALSE** if not. For **B_GET_WRITE_STATUS**, it should respond **TRUE** if the device is free to accept data, and **FALSE** if not.

B_GET_GEOMETRY

This *op* code requests the driver to supply information about the physical configuration of the device; it's generally appropriate only for mass storage devices. The control function should write the requested information into the **device_geometry** structure that the *data* pointer refers to. A **device_geometry** structure contains the following fields:

ulong **bytes_per_sector**	The number of bytes in each sector of storage.
ulong **sectors_per_track**	The number of sectors in each track.
ulong **cylinder_count**	The number of cylinders.
ulong **head_count**	The number of heads.
bool **removable**	Whether or not the storage medium can be removed (**TRUE** if it can be, **FALSE** if not).
bool **read_only**	Whether or not the medium can be read but not written (**TRUE** if it cannot be written, **FALSE** if it can).
bool **write_once**	Whether or not the medium can be written once, after which it becomes read-only (**TRUE** if it can be written only once, **FALSE** if it cannot be written or can be written more than once).

B_FORMAT

This operation requests the control function to format the device. The *data* argument doesn't contain any valid information.

Exported Functions

After a driver has been loaded, it runs as part of the kernel in the kernel's address space. It therefore is restricted to calling functions (a) that it implements or (b) that the kernel makes available to it. The driver links against the kernel alone; it cannot also independently link to something else, even the standard C library.

The kernel exports five kinds of functions so that they're available to a driver:

- Support Kit functions, such as **read_32_swap()** and **atomic_or()**. Like the error constants and data types that are defined in the Support Kit, these functions are available to drivers.
- Standard kernel functions, such as **area_for()** and **write_port()**, that were documented in the chapter on the Kernel Kit. Because the driver runs in the kernel's address space, it accesses these functions directly, not through the Kit. For this reason, not all of the functions are available to drivers; there are some services the kernel can provide to others, but not to itself.
- Library functions that the kernel incorporates. These are functions from the standard C library that have been adopted by the kernel and that the kernel, in turn, exports to the driver. They're a small, selected subset of library functions.
- Standard system calls, such as **read()** and **ioctl()**.
- Special functions that are implemented specifically for device drivers.

Functions from all five groups are listed in the sections below. Special driver functions are documented in detail in the section entitled "Functions for Drivers" on page 55.

Support Kit Functions

The kernel exports the following Support Kit functions:

read_16_swap()	**atomic_and()**
read_32_swap()	**atomic_or()**
write_16_swap()	**atomic_add()**
write_32_swap()	**real_time_clock()**

See the chapter on the Support Kit for descriptions of these functions.

Kernel Kit Functions

Most functions from the Kernel Kit are available to drivers. However, a few are not, sometimes because it would make no sense for a driver to call the function, and sometimes because it's difficult for the kernel to provide its very basic services to its own modules. In some cases, a special function is defined for drivers that takes the place of the missing Kit function. For example, **spawn_thread()** can't spawn a thread in the kernel. Since drivers run in the kernel, they need to use the special **spawn_kernel_thread()** instead. Similarly, **debugger()** can't be used to debug the kernel. Drivers should call **kernel_debugger()** instead.

The following Kernel Kit functions are exported for drivers:

Semaphores

create_sem()	**get_sem_info()**
acquire_sem()	**get_nth_sem_info()**
acquire_sem_etc()	**get_sem_count()**
release_sem()	**set_sem_owner()**
release_sem_etc()	**delete_sem()**

Threads

find_thread()	**suspend_thread()**
rename_thread()	**resume_thread()**
set_thread_priority()	**wait_for_thread()**
get_thread_info()	**exit_thread()**
get_nth_thread_info()	**kill_thread()**

Teams

kill_team()
get_team_info()
get_nth_team_info()

Ports

create_port()	**find_port()**
read_port()	**port_count()**
read_port_etc()	**port_buffer_size()**
write_port()	**port_buffer_size_etc()**
write_port_etc()	**set_port_owner()**
get_port_info()	**delete_port()**
get_nth_port_info()	

Time

snooze()
system_time()

Other

area_for()
get_system_info()

C Library Functions

The kernel exports a small number of functions from the standard C library. They include:

Functions declared in stdlib.h

atof()
atoi()
atol()
strtod()
strtol()
strtoul()
bsearch()
qsort()

malloc()
calloc()
free()
abs()
div()
labs()
ldiv()

Functions and macros declared in ctype.h

isalnum()
isalpha()
iscntrl()
isdigit()
isxdigit()
islower()
isupper()

ispunct()
isspace()
isprint()
isgraph()

tolower()
toupper()

Functions declared in string.h

strlen()
strcat()
strncat()
strcpy()
strncpy()
strcmp()
strncmp()
strchr()
strrchr()

strspn()
strcspn()
strstr()
strpbrk()
memset()
memchr()
memcmp()
memcpy()
memmove()

Functions declared in stdio.h

sprintf()
vsprintf()

The driver accesses these functions from the kernel, not from the library.

System Calls

The kernel also exports the five system calls that control devices:

open()
close()
read()
write()
ioctl()

Kernel Functions for Drivers

The kernel defines the following functions especially for drivers. For full documentation of these functions, see "Functions for Drivers" on page 55.

Spinlocks:

acquire_spinlock()
release_spinlock()

Disabling interrupts:

disable_interrupts()
restore_interrupts()

Interrupt handling:

set_io_interrupt_handler()	**set_isa_interrupt_handler()**
disable_io_interrupt()	**disable_isa_interrupt()**
enable_io_interrupt()	**enable_isa_interrupt()**

Memory management:

lock_memory()	**isa_address()**
unlock_memory()	**ram_address()**
get_memory_map()	

ISA DMA:

start_isa_dma()	**lock_isa_dma_channel()**
start_scattered_isa_dma()	**unlock_isa_dma_channel()**
make_isa_dma_table()	

PCI:

read_pci_config()
write_pci_config()
get_nth_pci_info()

Debugging:

dprintf()
set_dprintf_enabled()
kernel_debugger()

Hardware versions:

motherboard_version()
io_card_version()

SCSI common access method:

xpt_init()	**xpt_action()**
xpt_ccb_alloc()	**xpt_bus_register()**
xpt_ccb_free()	**xpt_bus_deregister()**

Other

spin()
spawn_kernel_thread()

Installation

The driver must be compiled as an add-on image, which in practical terms is much the same as compiling a shared library. *The Kernel Kit* chapter explains add-on images, and the Metrowerks *CodeWarrior* manual gives compilation instructions. In summary, you'll need to specify the following options for the linker (as **LDFLAGS** in the **makefile**):

- Instruct the linker to produce an add-on image by listing the **–G** (or **–sharedlibrary**) option.
- Disable the default behavior of linking against the shared system library by including the **–nodefaults** option.
- Export the driver's entry points so that the kernel can access them. The simplest way to do this is to export everything with the **–export all** option.
- Link the driver against the kernel by specifying the **/system/kernel** file. This is the only file that the driver should be linked against.

For the kernel to be able to find the compiled driver, it must be installed in the **/system/drivers** directory. This is the only place that the kernel looks for drivers to load.

When an attempt is made to open a device, the kernel first looks for its driver among those that are already loaded. Failing that, it looks on a floppy disk (in **/fd/system/drivers**). Failing to find one there, it looks next on the boot disk (in **/boot/system/drivers**).

If the **/system/drivers** directory contains more than one driver for the same device, it's indeterminate which one will be loaded.

You can give your driver any name you wish, as long as it doesn't match the name of another file in **/system/drivers**.

Functions for Drivers

The kernel exports a number of functions for the benefit of device drivers. These are functions that drivers can call to do their work; they're not functions that are available to applications. Although implemented by the kernel, they're not part of the Kernel Kit. The device driver accesses these functions directly from the kernel, not through a library.

acquire_spinlock(), release_spinlock()

<device/KernelExport.h>

void **acquire_spinlock(**spinlock **lock***)**

void **release_spinlock(**spinlock **lock***)**

These functions acquire and release the *lock* spinlock. Spinlocks, like semaphores, are used to protect critical sections of code that must remain on the same processor for a single path of execution—for example, code that atomically accesses a hardware register or a shared data structure. A common use for spinlocks is to protect data structures that both an interrupt handler and normal driver code must access.

However, spinlocks work quite differently from semaphores. No count is kept of how many times a thread has acquired the lock, for example, so calls to **acquire_spinlock()** and **release_spinlock()** should not be nested. More importantly, **acquire_spinlock()** spins while attempting to acquire the lock; it doesn't block or release its hold on the CPU.

These functions assume that interrupts have been disabled. They should be nested within calls to **disable_interrupts()** and **restore_interrupts()** as follows:

```
spinlock lock;
cpu_status former = disable_interrupts();
acquire_spinlock(&lock);
/* critical code goes here */
release_spinlock(&lock);
restore_interrupts(former);
```

These two pairs of functions enable the thread to get into the critical code without rescheduling. Disabling interrupts ensures that the thread won't be preemptively rescheduled. Because **acquire_spinlock()** doesn't block, it provides the additional assurance that the thread won't be voluntarily rescheduled.

Executing the critical code under the protection of the spinlock guarantees that no other thread will execute the same code at the same time on another processor. Spinlocks should be held only as long as necessary and released as quickly as possible.

See also: **create_spinlock()**

create_spinlock(), delete_spinlock()

<device/KernelExport.h>

spinlock ***create_spinlock(**void**)**

void **delete_spinlock(**spinlock **lock***)**

< These functions will, when implemented and exported, produce and destroy spinlocks. Currently, they're declared but not exported. To create a spinlock at present, simply declare a **spinlock** variable and pass a pointer to it to **acquire_spinlock()**. >

See also: **acquire_spinlock()**

disable_interrupts(), restore_interrupts()

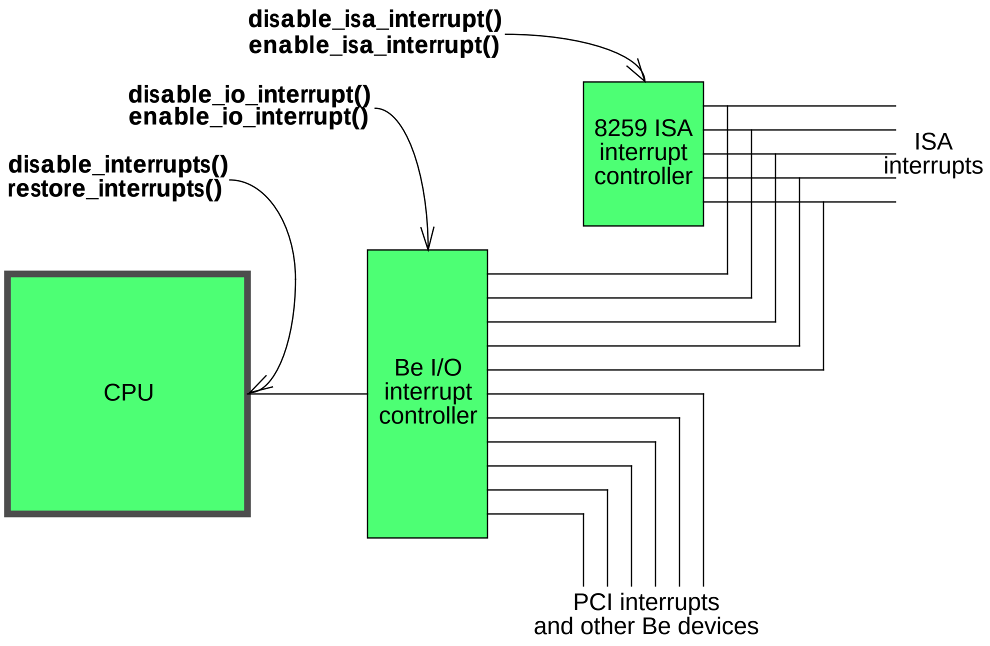
<device/KernelExport.h>

cpu_status **disable_interrupts(**void**)**

void **restore_interrupts(**cpu_status *status***)**

These functions disable interrupts at the CPU (the one the caller is currently running on) and restore them again. **disable_interrupts()** prevents the CPU from being interrupted and returns its previous status—whether or not interrupts were already disabled before the **disable_interrupts()** call. **restore_interrupts()** restores the previous *status* of the CPU, which should be the value that **disable_interrupts()** returned. Passing the status returned by **disable_interrupts()** to **restore_interrupts()** allows these functions to be paired and nested.

As diagrammed below, individual interrupts can be enabled and disabled at two other hardware locations. **disable_isa_interrupt()** and **enable_isa_interrupt()** work at the ISA standard 8259 interrupt controller, and **disable_io_interrupt()** and **enable_io_interrupt()** act at the Be-defined I/O interrupt controller that combines ISA and PCI interrupts.



Interrupts that have been disabled by **disable_interrupts()** must be reenabled by **restore_interrupts()**.

See also: **acquire_spinlock()**, **set_io_interrupt_handler()**, **set_isa_interrupt_handler()**

disable_io_interrupt() *see* set_io_interrupt_handler()

disable_isa_interrupt() *see* set_isa_interrupt_handler()

dprintf(), set_dprintf_enabled(), kernel_debugger()

<device/KernelExport.h>

void **dprintf(**const char **format*, ...**)**

bool **set_dprintf_enabled(**bool *enabled***)**

void **kernel_debugger(**const char **string***)**

dprintf() is a debugging function that has the same syntax and behavior as standard C **printf()**, except that it writes its output to the fourth serial port ("/dev/serial4") at a data rate of 19,200 bits per second. By default, **dprintf()** is disabled.

set_dprintf_enabled() enables **dprintf()** if the *enabled* flag is **TRUE**, and disables it if the flag is **FALSE**. It returns the previous enabled state. Calls to this function can be nested by caching the return value of a call that disables printing and passing it to the paired call that restores the previous state.

kernel_debugger() drops the calling thread into a debugger that writes its output to the fourth serial port at 19,200 bits per second, just as **dprintf()** does. This debugger produces *string* as its first message; it's not affected by **set_dprintf_enabled()**.

kernel_debugger() is identical to the **debugger()** function documented in the Kernel Kit, except that it works in the kernel and engages a different debugger. Drivers should use it instead of **debugger()**.

See also: **debugger()** in the Kernel Kit

enable_io_interrupt() *see* set_io_interrupt_handler()

enable_isa_interrupt() *see* set_isa_interrupt_handler()

get_memory_map()

<device/KernelExport.h>

long **get_memory_map(**void **address*, ulong *numBytes*,
physical_entry **table*, long *numEntries***)**

Locates the separate pieces of physical memory that correspond to the contiguous buffer of virtual memory beginning at *address* and extending for *numBytes*. Each piece of physical memory is described by a **physical_entry** structure. It has just two fields:

void ***address**	The address of a block of physical memory.
ulong **size**	The number of bytes in the block.

This function is passed a pointer to a *table* of **physical_entry** structures. It fills in the table, stopping when the entire buffer of virtual memory has been described or when *numEntry* entries in the table have been written, whichever comes first.

If the *table* provided isn't big enough, you'll need to call **get_memory_map()** again and ask it to describe the rest of the buffer. If the table is too big, this function sets the **size** field of the entry following the last one it needed to 0. This indicates that it has finished mapping the entire *address* buffer.

Memory should be locked while it is being mapped. Before calling **get_memory_map()**, call **lock_memory()** to make sure that it all stays in place:

```
physical_entry table[count];
lock_memory(someAddress, someNumberOfBytes, FALSE);
get_memory_map(someAddress, someNumberOfBytes, table, count);
. . .
unlock_memory(someAddress, someNumberOfBytes);
```

< This function consistently returns **B_NO_ERROR**. >

See also: **lock_memory()**, **start_isa_dma()**

get_nth_pci_info()

<device/PCI.h>

long **get_nth_pci_info(**long *index*, pci_info **info***)**

This function looks up the PCI device at *index* and provides a description of it in the **pci_info** structure that *info* refers to. Indices begin at 0 and there are no gaps in the list.

The **pci_info** structure contains a number of fields that report values found in the configuration register space for the device and it also describes how the device has been mapped into the system. The following fields are common to all devices:

ushort **vendor_id**	An identifier for the manufacturer of the device.
ushort **device_id**	An identifier for the particular device of the vendor, assigned by the vendor.
uchar **bus**	The bus number.
uchar **device**	The number that identifies the location of the device on the bus.
uchar **function**	The function number in the device.
uchar **revision**	A device-specific version number, assigned by the vendor.
uchar **class_api**	The type of specific register-level programming interface for the device (the lower byte of the class code field).
uchar **class_sub**	The specific type of function the device performs (the middle byte of the class code field).
uchar **class_base**	The broadly-defined device type (the upper byte of the class code field).
uchar **line_size**	The size of the system cache line, in units of 32-bit words.
uchar **latency**	The latency timer for the PCI bus master.
uchar **header_type**	The header type.
uchar **bist**	The contents of the register for the built-in self test.
uchar **u**	A union of structures, one for each header type.

Currently, there's only one header (type 0x00), but in the future there may be others. Consequently, header-specific information is recorded in a union of structures, one for each header type. The union (named simply **u**) at present has just one member, a structure for the current header (named **h0**):

```
typedef struct {
        . . .
        union {
                struct {
                        . . .
                } h0;
        } u;
} pci_info
```

The fields of the **h0** structure are:

ulong **cardbus_cis**	The CardBus CIS pointer.
ushort **subsystem_id**	The vendor-assigned identifier for the add-in card containing the device.
ushort **subsystem_vendor_id**	The identifier for manufacturer of the add-in card that contains the device.
ulong **rom_base**	The base address for the expansion ROM, as viewed from the host processor.
ulong **rom_base_pci**	The base address for the expansion ROM, as viewed from the PCI bus. This is the address a bus master would use.
ulong **rom_size**	The amount of memory in the expansion ROM, in bytes.
ulong **base_registers**[6]	The base addresses of requested memory spaces and I/O spaces, as viewed from the host processor.
ulong **base_registers_pci**[6]	The base addresses of requested memory spaces and I/O spaces, as viewed from the PCI bus. This is the address a bus master would use.
ulong **base_register_sizes**[6]	The sizes of requested memory spaces and I/O spaces.
uchar **base_register_flags**[6]	The flags from the base-address registers.

uchar **interrupt_line**	The interrupt line. This number identifies the interrupt associated with the device. See **set_io_interrupt_handler()**.
uchar **interrupt_pin**	The interrupt pin that the device uses.
uchar **min_grant**	The minimum burst period the device needs, assuming a clock rate of 33 MHz.
uchar **max_latency**	The maximum frequency at which the device needs access to the PCI bus.

In **device/PCI.h**, you'll find a number of constants that you can use to test various fields of a **pci_info** structure. See the *PCI Local Bus Specification*, published by the PCI Special Interest Group (Portland, OR) for more information on the configuration of a PCI device.

get_nth_pci_info() returns **B_NO_ERROR** if it successfully describes a PCI device, and **B_ERROR** if it can't find the device (for example, if *index* is out-of-range).

See also: **read_pci_config()**

io_card_version() *see* motherboard_version()

isa_address()

<device/KernelExport.h>

void ***isa_address(**long *offset***)**

Returns the virtual address corresponding to the specified *offset* in the ISA I/O address space. By passing an *offset* of 0, you can find the base address that's mapped to the ISA address space.

kernel_debugger() *see* dprintf()

lock_isa_dma_channel(), unlock_isa_dma_channel()

<device/KernelExport.h>

long **lock_isa_dma_channel(**long *channel***)**

long **unlock_isa_dma_channel(**long *channel***)**

These functions reserve an ISA DMA *channel* and release a channel previously reserved. They return **B_NO_ERROR** if successful, and **B_ERROR** if not. Like semaphores, these functions work only if all participating parties adhere to the protocol.

There are 7 ISA DMA channels. In general, they're used as follows:

Channel	Use
0	Unreserved, available
1	Unreserved, available
2	Reserved for the floppy disk controller
3	Reserved for the parallel port driver
5	Reserved for IDE
6	Reserved for sound
7	Reserved for sound

Channel 4 is taken by the system; it cannot be used.

lock_memory(), unlock_memory()

<device/KernelExport.h>

long **lock_memory(**void **address*, ulong *numBytes*, bool *willChange***)**

long **unlock_memory(**void **address*, ulong *numBytes***)**

lock_memory() makes sure that all the memory beginning at the specified virtual *address* and extending for *numBytes* is resident in RAM, and locks it so that it won't be paged out until **unlock_memory()** is called. It pages in any of the memory that isn't resident at the time it's called.

The *willChange* flag should be **TRUE** if any part of the memory range will be altered while it's locked—especially if the hardware device will do anything to modify the memory, since that won't otherwise be noticed by the system and the modified pages may not be written. The *willChange* flag should be **FALSE** if the memory won't change while it's locked.

Each of these functions returns **B_NO_ERROR** if successful and **B_ERROR** if not. The main reason that **lock_memory()** would fail is that you're attempting to lock more memory than can be paged in.

make_isa_dma_table() *see* start_isa_dma()

motherboard_version(), io_card_version()

<device/KernelExport.h>

long **motherboard_version(**void**)**

long **io_card_version(**void**)**

These functions return the current versions of the motherboard and of the I/O card.

ram_address()

<device/KernelExport.h>

void ***ram_address(**void **physicalAddress***)**

Returns the address of a physical block of system memory (RAM) as viewed from the PCI bus. If passed **NULL** as the *physicalAddress*, this function returns a pointer to the first byte of RAM; otherwise it returns a pointer to the *physicalAddress*.

This information is needed by bus masters—components, such as the ethernet and some SCSI controllers, that can perform DMA reads and writes (directly read from and write to system memory without CPU intervention).

Memory must be locked when calling this function. For example:

```
physical_entry table[count];
void *where;

lock_memory(someAddress, someNumberOfBytes, FALSE);
get_memory_map(someAddress, someNumberOfBytes, table, count);
where = ram_address(table[i].address)
. . .
unlock_memory(someAddress, someNumberOfBytes);
```

See also: **get_memory_map()**, **lock_memory()**

read_pci_config(), write_pci_config()

<device/PCI.h>

long **read_pci_config(**uchar *bus*, uchar *device*, uchar *function*,
long *offset*, long *size***)**

void **write_pci_config(**uchar *bus*, uchar *device*, uchar *function*,
long *offset*, long *size*, long *value***)**

These functions read from and write to the PCI configuration register space. The *bus*, *device*, and *function* arguments can be read from the **bus**, **device**, and **function** fields of the **pci_info** structure provided by **get_nth_pci_info()**. They identify the configuration space that belongs to the device.

The *offset* is an offset to the location in the 256-byte configuration space that is to be read or written and *size* is the number of bytes to be read from that location or written to it. Permitted sizes are 1, 2, and 4 bytes. **read_pci_config()** returns the bytes that are read, **write_pci_config()** writes *size* bytes of *value* to the *offset* location.

See also: **get_nth_pci_info()**

release_spinlock() *see* acquire_spinlock()

restore_interrupts() *see* disable_interrupts()

set_dprintf_enabled() *see* dprintf()

set_io_interrupt_handler(), disable_io_interrupt(), enable_io_interrupt()

<device/KernelExport.h>

long **set_io_interrupt_handler(**long *interrupt*,
interrupt_handler *function*, void **data***)**

long **disable_io_interrupt(**long *interrupt***)**

long **enable_io_interrupt(**long *interrupt***)**

These functions manage interrupts at the Be-designed I/O interrupt controller that combines ISA and PCI interrupts. The *interrupt* can be an ISA IRQ value or the **interrupt_line** field read from the **pci_info** structure provided by **get_nth_pci_info()**.

set_io_interrupt_handler() installs the handler *function* that will be called each time the specified *interrupt* occurs. This function should have the following syntax:

bool **handler(**void **data***)**

The *data* that's passed to **set_io_interrupt_handler()** will be passed to the handler function each time it's called. It can be anything that might be of use to the handler, or **NULL**. This function should always return **TRUE**.

set_io_interrupt_handler() itself returns **B_NO_ERROR** if successful in installing the handler, and **B_ERROR** if not.

disable_io_interrupt() disables the named *interrupt*, and **enable_io_interrupt()** reenables it. Both functions return **B_ERROR** for an invalid *interrupt* number, and **B_NO_ERROR** otherwise. Neither function takes into account the disabled or enabled state of the interrupt as it might be affected by other functions, such as **disable_isa_interrupt()** or **restore_interrupts()**. An interrupt that has been disabled by **disable_io_interrupt()** must be reenabled by **enable_io_interrupt()**.

See also: **get_nth_pci_info()**, **disable_interrupts()**

set_isa_interrupt_handler(), disable_isa_interrupt(), enable_isa_interrupt()

<device/KernelExport.h>

long **set_isa_interrupt_handler(**long *interrupt*,
interrupt_handler *function*, void **data***)**

long **disable_isa_interrupt(**long *interrupt***)**

long **enable_isa_interrupt(**long *interrupt***)**

These functions manage interrupts at the 8259 ISA-compatible interrupt controller. The *interrupt* is identified by its standard IRQ value.

set_isa_interrupt_handler() installs the handler *function* for the specified *interrupt*. This function should take one argument and return a **bool**:

bool **handler(**void **data***)**

The argument is the same *data* that's passed to **set_isa_interrupt_handler()**; it can be any kind of data the *function* might need, or **NULL**. The return value indicates whether the interrupt was handled—**TRUE** if it was and **FALSE** if not. By returning **FALSE**, the handler function can indicate that the device didn't generate the interrupt. The system can then try a different handler installed for a different device at the same interrupt number.
< However, this architecture is not currently supported, so the handler function should always return **TRUE**. >

set_isa_interrupt_handler() returns **B_NO_ERROR** if it can install the handler for the interrupt, and **B_ERROR** if not.

disable_isa_interrupt() disables the specified ISA *interrupt*, and **enable_isa_interrupt()** reenables it. Both functions return **B_ERROR** if the interrupt passed is not a valid IRQ value. Neither function considers whether the interrupt might be disabled or enabled by some other function, such as **disable_io_interrupt()**. An interrupt that has been disabled by **disable_isa_interrupt()** must be reenabled by **enable_isa_interrupt()**.

ISA interrupts can also be managed at the Be-designed interrupt dispatcher that controls PCI interrupts. The Be interrupt controller is somewhat faster than the edge-sensitive ISA controller. If your device can generate a level-sensitive interrupt, it should use the counterpart **set_io_interrupt_handler()** function instead of **set_isa_interrupt_handler()**. However, if it depends on the edge-sensitive ISA interrupt controller widely found in the PC world, it needs to use these ISA functions.

See also: **set_io_interrupt_handler()**, **disable_interrupts()**

spawn_kernel_thread()

<device/KernelExport.h>

thread_id **spawn_kernel_thread(**thread_entry *func*, const char **name*,
long *priority*, void **data***)**

This function is a counterpart to **spawn_thread()** in the Kernel Kit, which is not exported for drivers. It has the same syntax as the Kernel Kit function, but is able to spawn threads in the kernel itself.

See also: **spawn_thread()** in the Kernel Kit

spin()

<device/KernelExport.h>

void **spin(**double *microseconds***)**

Executes a delay loop lasting at least the specified number of *microseconds*. It could last longer, due to rounding errors, interrupts, and context switches.

start_isa_dma(), start_scattered_isa_dma(), make_isa_dma_table()

<device/KernelExport.h>

long **start_isa_dma(**long *channel*, void **address*, long *transferCount*,
uchar *mode*, uchar *eMode***)**

long **start_scattered_isa_dma(**long *channel*, isa_dma_entry **table*,
uchar *mode*, uchar *eMode***)**

long **make_isa_dma_table(**void **address*, long *numBytes*,
ulong *numTransferBits*,
isa_dma_entry **table*, long *numEntries***)**

These functions initiate ISA DMA memory transfers for the specified *channel*. They engage the ISA 8237 DMA controller.

start_isa_dma() starts the transfer of a contiguous block of physical memory beginning at the specified *address*. It requests *transferCount* number of transfers, which cannot be greater than **B_MAX_ISA_DMA_COUNT**. Each transfer will move 8 or 16 bits of memory, depending on the *mode* and *eMode* flags. These arguments correspond to the mode and extended mode flags recognized by the DMA controller.

The physical memory *address* that's passed to **start_isa_dma()** can be obtained by calling **get_memory_map()**.

start_scattered_isa_dma() starts the transfer of a memory buffer that's physically scattered in various pieces. The separate pieces of memory are described by the *table* passed as a second argument and provided by **make_isa_dma_table()**.

make_isa_dma_table() provides a description of the separate chunks of physical memory that make up the contiguous virtual buffer that begins at *address* and extends for *numBytes*. This function anticipates a subsequent call to **start_scattered_isa_dma()**, which initiates a DMA transfer. It ensures that the information it provides is in the format expected by the 8237 DMA controller. This depends in part on how many bits will be transferred at a time. The third argument, *numTransferBits*, provides this information. It can be **B_8_BIT_TRANSFER** or **B_16_BIT_TRANSFER**.

Each chunk of physical memory is described by a **isa_dma_entry** structure, which contains the following fields (not that its arcane details matter, since you don't have to do anything with the information except pass it to **start_scattered_isa_dma()**):

ulong **address**	A physical memory address (in little endian format).
ushort **transfer_count**	The number of transfers it will take to move all the physical memory at that address, minus 1 (in little endian format). This value won't be greater than **B_MAX_ISA_DMA_COUNT**.
int **flags**.**end_of_list**:1	A flag that's set to mark the last chunk of physical memory corresponding to the virtual buffer.

make_isa_dma_table() is passed a pointer to a *table* of **isa_dma_entry** structures. It fills in the table, stopping when the entire buffer of virtual memory has been described or when *numEntry* entries in the table have been written, whichever comes first. It returns the number of bytes from the virtual *address* buffer that it was able to account for in the *table*.

start_isa_dma() and **start_scattered_isa_dma()** both return **B_NO_ERROR** if successful in initiating the transfer, and **B_ERROR** if the channel isn't free.

unlock_isa_dma_channel() *see* lock_isa_dma_channel()

unlock_memory() *see* lock_memory()

write_pci_config() *see* read_pci_config()

xpt_init(), xpt_ccb_alloc(), xpt_ccb_free(), xpt_action(), xpt_bus_register(), xpt_bus_deregister()

<device/CAM.h>

long **xpt_init (**void**)**

CCB_HEADER ***xpt_ccb_alloc(**void**)**

void **xpt_ccb_free(**void **ccb***)**

long **xpt_action(**CCB_HEADER **ccbHeader***)**

long **xpt_bus_register(**CAM_SIM_ENTRY **entryPoints***)**

long **xpt_bus_deregister(**long *pathID***)**

These functions conform to the SCSI common access method (CAM) specification. See the draft ANSI standard *SCSI-2 Common Access Method Transport and SCSI Interface Modules* for information.

< The current implementation doesn't support asynchronous callback functions. All CAM requests are executed synchronously in their entirety. >

Constants and Defined Types for Kernel-Loadable Drivers

This section lists the constants and types that are defined for drivers the kernel loads. Everything listed here was explained in the previous sections on “Developing a Kernel-Loadable Driver” and “Functions for Drivers”.

Constants

Control Operations

<device/Drivers.h>

Enumerated constant

B_GET_SIZE
B_SET_SIZE
B_SET_NONBLOCKING_IO
B_SET_BLOCKING_IO
B_GET_READ_STATUS
B_GET_WRITE_STATUS
B_GET_GEOMETRY
B_FORMAT

B_DEVICE_OP_CODES_END = 9999

These constants name the control operations that the kernel defines. You should expect the control hook function for any driver you develop to be called with these constants as the operation code (*op*).

All system-defined control constants are guaranteed to have values less than **B_DEVICE_OP_CODES_END**. Since additional constants might be defined for future releases, any that you define should be greater than **B_DEVICE_OP_CODES_END**.

See also: “Control Operations” on page 48

ISA DMA Transfer Maximum

<device/KernelExport.h>

Defined constant	Value
B_MAX_ISA_DMA_COUNT	0x10000

This constant indicates the maximum number of transfers for a single DMA request.

See also: **start_isa_dma()** on page 66

ISA DMA Transfer Sizes

<device/KernelExport.h>

Enumerated constant

B_8_BIT_TRANSFER
B_16_BIT_TRANSFER

These constants are passed to **make_isa_dma_table()** to indicate the size of a single DMA transfer.

See also: **start_isa_dma()** on page 66

Defined Types

cpu_status

<device/KernelExport.h>

typedef ulong **cpu_status**

This defined type is returned by **disable_interrupts()** to record whether interrupts were already disabled or not. It can be passed to **restore_interrupts()** to restore the previous state.

See also: **disable_interrupts()** on page 56

device_close_hook

<device/Drivers.h>

typedef long (***device_close_hook**)**(**device_info **info***)**

The hook function that the kernel calls to close a device must conform to this type.

See also: “Opening and Closing a Device” on page 46

device_control_hook

<device/Drivers.h>

typedef long (***device_control_hook**)**(**device_info **info*, ulong *op*, void **data***)**

The hook function for controlling a device must conform to this type.

See also: “Controlling the Device” on page 47

device_entry

<device/Drivers.h>

```
typedef struct {
        const char *name;
        device_open_hook open;
        device_close_hook close;
        device_control_hook control;
        device_io_hook read;
        device_io_hook write;
} device_entry
```

This structure declares the name of a device and the hook functions that the kernel can call to operate that device. The driver must provide one **device_entry** declaration for each of its devices.

See also: “Device Declarations” on page 43

device_geometry

<device/Drivers.h>

```
typedef struct {
        ulong bytes_per_sector;
        ulong sectors_per_track;
        ulong cylinder_count;
        ulong head_count;
        bool removable;
        bool read_only;
        bool write_once;
} device_geometry
```

Drivers use this structure to report the physical configuration of a mass-storage device.

See also: “**B_GET_GEOMETRY**” on page 49

device_info

<device/Drivers.h>

```
typedef struct {
        device_entry *entry;
        void *private_data;
} device_info
```

This structure contains publicly declared and private information about a device. It's passed as the first argument to each of the device-specific hook functions.

See also: "Hook Functions" on page 46

device_io_hook

<device/Drivers.h>

```
typedef long (*device_io_hook)(device_info *info, void *data, ulong numBytes,
                               ulong position)
```

The hook functions that the kernel calls to read data from or write it to a device must conform to this type.

See also: "Reading and Writing Data" on page 47

device_open_hook

<device/Drivers.h>

```
typedef long (*device_open_hook)(device_info *info, ulong flags)
```

The hook function that opens a device must conform to this type.

See also: "Opening and Closing a Device" on page 46

interrupt_handler

```
<device/KernelExport.h>

typedef bool (*interrupt_handler)(void *data)
```

The functions that are installed to handle interrupts must conform to this type.

See also: **set_io_interrupt_handler()** on page 64, **set_isa_interrupt_handler()** on page 65

isa_dma_entry

```
<device/KernelExport.h>

typedef struct {
        ulong address;
        ushort transfer_count;
        uchar reserved;
        struct {
                int end_of_list:1;
                int reserved:7;
        } flags;
} isa_dma_entry
```

This structure is filled in by **make_isa_dma_table()** and is passed unchanged to **start_scattered_isa_dma()**.

See also: **start_isa_dma()** on page 66

pci_info

```
<device/PCI.h>

typedef struct {
    ushort vendor_id;
    ushort device_id;
    uchar bus;
    uchar device;
    uchar function;
    uchar revision;
    uchar class_api;
    uchar class_sub;
    uchar class_base;
    uchar line_size;
    uchar latency;
    uchar header_type;
    uchar bist;
    uchar reserved;
    union {
        struct {
            ulong cardbus_cis;
            ushort subsystem_id;
            ushort subsystem_vendor_id;
            ulong rom_base;
            ulong rom_base_pci;
            ulong rom_size;
            ulong base_registers[6];
            ulong base_registers_pci[6];
            ulong base_register_sizes[6];
            uchar base_register_flags[6];
            uchar interrupt_line;
            uchar interrupt_pin;
            uchar min_grant;
            uchar max_latency;
        } h0;
    } u;
} pci_info
```

This structure reports values from the PCI configuration register space and describes how the device has been mapped into the system.

See also: **get_nth_pci_info()** on page 59

physical_entry

```
<device/KernelExport.h>

typedef struct {
        void *address;
        ulong size;
} physical_entry
```

This structure is used to describe a chunk of physical memory corresponding to some part of a contiguous virtual buffer.

See also: **get_memory_map()** on page 58

spinlock

```
<device/KernelExport.h>

typedef vlong spinlock
```

This data type serves the **acquire_spinlock()**/**release_spinlock()** protocol.

See also: **acquire_spinlock()** on page 55

Developing a Driver for a Graphics Card

Like other drivers, drivers for graphics cards are dynamically loaded modules—but they're loaded by the Application Server, the software component that's responsible for graphics operations, not by the kernel. Because they're add-on modules, these drivers share some similarities with other drivers:

- They lack a **main()** function, but must provide entry points where the host software (the Application Server in this case) can access driver functionality.
- They're mostly limited to calling functions that they implement themselves. They don't link against the system library or the host Application Server, < but they can link against a private library that gives them the ability to make some system calls >.
- They must be compiled as add-on modules and installed in a place well-known to the host.

Because the host module is the Application Server rather than the kernel, graphics card drivers must follow protocols that the Server defines, not those that the kernel imposes on other drivers. The entry points, exported functions, and installation directory are all specific to graphics card drivers. Therefore, if you're developing a driver for a graphics card, disregard the preceding sections of this chapter dealing with kernel drivers and follow the rules outlined in this section instead.

Control of a graphics card driver resides only with the host module; there are no functions (like **open()** or **ioctl()**) through which a program can control the driver.

However, applications can get direct access to the graphics card through the BWindowScreen class in the Game Kit. Access is provided by making a clone of the graphics card driver and attaching it to the application. The original driver remains running as part of the Application Server, but its connection to the screen is suspended while the clone is active.

Entry Point

Every graphics card driver must implement a function called **control_graphics_card()**. This is the Application Server's main entry point into the driver; it's the function the

Server calls to set up the driver, query it for information, pass it configuration instructions, and generally control what it does. It has the following syntax:

long **control_graphics_card(**ulong *op*, void **data***)**

The first argument, *op*, names the operation the driver is requested to perform. The second argument, *data*, points either to some information that will help the driver carry out the request or to a location where it should write some information as a result of the operation. The exact type of data in either case depends on the nature of the operation. The *op* and *data* arguments are inextricably linked.

The return value is an error code. In general, the control function should return **B_NO_ERROR** if it can successfully respond to a particular *op* request, and **B_ERROR** if it cannot. It should also respond **B_ERROR** to any undefined *op* code requests it doesn't understand.

Main Control Operations

There are seventeen control operations that a driver's **control_graphics_card()** function can be requested to perform (seventeen *op* codes defined in **device/GraphicsCard.h**). Nine of these operations give the Application Server general control over the driver. The other eight concern the cloning of the driver and the direct control of the frame buffer through the Game Kit. Those operations are discussed under "Control Operations for Cloning the Driver" and "Control Operations for Manipulating the Frame Buffer" below. This sections lists and discusses the nine main control operations.

B_OPEN_GRAPHICS_CARD

This *op* code requests the driver to open and initialize the graphics card specified by the *data* argument. If the driver can open the card, it should do so and return **B_NO_ERROR**. If it can't, it should return **B_ERROR**. The *data* pointer refers to a **graphics_card_spec** structure with the following fields:

void ***screen_base**	The beginning of memory on the graphics card. The driver can locate the frame buffer somewhere in this memory, but not necessarily at the base address.
void ***io_base**	The base address for the I/O registers that control the graphics card. Registers are addressed by 16-bit offsets from this base address.
ulong **vendor_id**	The number that identifies the manufacturer of the graphics chip on the card.
ulong **device_id**	A number that identifies the particular graphics chip of that manufacturer.

If the driver can open the graphics card, it should take the opportunity to initialize any data structures it might need. However, it should wait for further instructions—particularly a

B_CONFIG_GRAPHICS_CARD request—before initializing the frame buffer or turning on the video display.

If the driver returns **B_ERROR**, indicating that it's not the driver for the specified graphics card, it will immediately get a request to close the card as a prelude to being unloaded.

B_CLOSE_GRAPHICS_CARD

This operation notifies the graphics card driver that it's about to be unloaded. The *data* argument is meaningless (it doesn't point to any valid information). The Application Server ignores the return value and unloads the driver no matter what.

B_SET_INDEXED_COLOR

This operation is used to set up the palette of colors that can be displayed when the frame buffer is 8 bits deep. It requests the driver to place a particular color at a particular position in the list of 256 colors that's kept on the card. The *data* argument points to a **indexed_color** structure with two pieces of information:

long **index**	The index of the color in the list. This value is used as the color value in the **B_COLOR_8_BIT** color space. Indices begin at 0.
rgb_color **color**	The full 32-bit color that should be associated with the index.

A driver can expect a series of **B_SET_INDEXED_COLOR** requests soon after it is opened. It might get subsequent requests when an application (through the Game Kit) modifies the color list, and when the game returns control to the Application Server.

B_GET_GRAPHICS_CARD_HOOKS

This *op* code requests **control_graphics_card()** to supply the Application Server with an array of function pointers. Each pointer is to a hook function that the Server can call to carry out a specific graphics task. A total of **B_HOOK_COUNT** (48 at present) pointers must be written, although only a quarter of that number are currently used. The full array should be written to the location the *data* argument points to, with **NULL** values inserted for undefined functions.

A later section, "Hook Functions" on page 87, describes the hook functions, the tasks they should perform, their arguments and return types, and their positions in the array.

A driver can expect a **B_GET_GRAPHICS_CARD_HOOKS** request soon after it is opened, and again any time the screen configuration changes. The hook functions can be tailored to a specific screen dimension and depth.

B_GET_GRAPHICS_CARD_INFO

This *op* code requests the **control_graphics_card()** function to supply information about the driver and the current configuration of the screen. The *data* argument points to a **graphics_card_info** structure where it should write this information. This structure contains the following fields:

short **version**	The version of the Be architecture for graphics cards that the driver was designed to work with. The current version is 2.
short **id**	An identifier for the driver, understood in relation to the version number. The Application Server doesn't check this number; it can be set to any value you desire.
void ***frame_buffer**	A pointer to the first byte of the frame buffer.
char **rgba_order**[4]	The order of color components as the bytes for those components are stored in video memory (in the frame buffer). This array should arrange the characters 'r' (red), 'g' (green), 'b' (blue), and 'a' (alpha) in the order in which those components are intermeshed for each pixel in the frame buffer; a typical order is "bgra". This field is valid only for screen depths of 32 bits per pixel.
short **flags**	A mask containing flags that describe the ability of the graphics card driver to perform particular tasks.
short **bits_per_pixel**	The depth of the screen in bits per pixel. Only 32-bit (**B_RGB_32_BIT**) and 8-bit (**B_COLOR_8_BIT**) depths are currently supported.
long **bytes_per_row**	The offset, in bytes, between two adjacent rows of pixel data in the frame buffer (the number of bytes assigned to each row).
short **width**	The width of the frame buffer in pixels (the number of pixel columns it defines).
short **height**	The height of the frame buffer in pixels (the number of pixel rows it defines).

Three constants are currently defined for the **flags** mask:

B_CRT_CONTROL	Indicates that the driver is able to control, to any extent, the position or the size of the CRT display on the monitor—that there's a provision for controlling the CRT through software, not just hardware.
B_GAMMA_CONTROL	Indicates that the driver is able to make gamma corrections that compensate for the particular characteristics of the display device.

B_FRAME_BUFFER_CONTROL	Indicates that the driver allows clients to set arbitrary dimensions for the frame buffer and to control which portion of the frame buffer (the display area) is mapped to the screen.

The driver will receive frequent **B_GET_GRAPHICS_CARD_INFO** requests. The **graphics_card_info** structure it supplies should always reflect the values currently in force.

B_GET_REFRESH_RATES

This *op* code asks **control_graphics_card()** to place the current refresh rate, as well as the maximum and minimum rates, in the **refresh_rate_info** structure referred to by the *data* pointer. This structure contains the following fields:

float **min**	The minimum refresh rate that the graphics card is capable of, given the current configuration.
float **max**	The maximum refresh rate that the graphics card is capable of, given the current configuration.
float **current**	The current refresh rate.

All values should be provided in hertz.

B_GET_SCREEN_SPACES

This *op* code requests the driver to supply a mask containing all possible configurations of the screen space—all supported combinations of pixel depth and dimensions of the pixel grid. The mask is formed from the following constants—which are defined in **interface/InterfaceDefs.h**—and is written to the location indicated by the *data* pointer:

B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

For example, if the mask includes **B_8_BIT_1600x1200**, the driver can configure a frame buffer that's simultaneously 8 bits deep (the **B_COLOR_8_BIT** color space), 1,600 pixel columns wide, and 1,200 pixel rows high. The mask should include all configurations that the graphics card is capable of supporting.

< The Application Server currently doesn't permit depths of 16 bits. >

(**InterfaceDefs.h** defines one other screen space, **B_8_BIT_640x400**, but this is reserved for the default "supervga" driver provided by Be.)

B_CONFIG_GRAPHICS_CARD

This *op* code asks the control function to configure the display according to the values set in the **graphics_card_config** structure that the *data* argument points to. This structure contains the following fields:

ulong **space**	The size of the pixel grid on-screen and the depth of the frame buffer in bits per pixel. This field will be one of the constants listed above for the **B_GET_SCREEN_SPACES** control operation.
float **refresh_rate**	The refresh rate of the screen in hertz.
uchar **h_position**	The horizontal position of the CRT display on the monitor.
uchar **v_position**	The vertical position of the CRT display on the monitor.
uchar **h_size**	The horizontal size of the CRT display on the monitor.
uchar **v_size**	The vertical size of the CRT display on the monitor.

The most important configuration parameter is the **space** field. The driver should reconfigure the screen to the depth and size requested and return **B_NO_ERROR**. If it can't carry out the request, it should return **B_ERROR**.

Failure to comply with the other fields of the **graphics_card_config** structure should not result in a **B_ERROR** return value. The driver should come as close as it can to the requested refresh rate. The last four fields are appropriate only for drivers that reported that they could control the positioning of the CRT display (by setting the **B_CRT_CONTROL** flag in response to a **B_GET_GRAPHICS_CARD_INFO** request).

The values for the four CRT configuration fields range from 0 through 100, with 50 as the default. Values of less than 50 for **h_position** and **v_position** should move the display toward the left and top; those greater than 50 should move it to the right and bottom. Values of less than 50 for **h_size** and **v_size** should make the display narrower and shorter, squeezing it into a smaller area; values greater than 50 should make it wider and taller.

B_SET_SCREEN_GAMMA

This operation asks the driver to set up a table for adjusting color values to correct for the peculiarities of the display device. The *data* argument points to a **screen_gamma** structure with gamma corrections for each color component. It contains the following three fields:

uchar **red**[256]	Mappings for the red component.
uchar **green**[256]	Mappings for the green component.
uchar **blue**[256]	Mappings for the blue component.

Each field is a component-specific array. The stated color value is used as an index into the array; the value found at that index substitutes for the stated value. For example, if the value at **blue**[152] is 154, all blue component values of 152 should be replaced by 154, essentially adding to the blueness of the color as displayed.

Only drivers that indicated they could make gamma corrections (by setting the **B_GAMMA_CONTROL** flag in response to a **B_GET_GRAPHICS_CARD_INFO** request) need to respond to **B_SET_SCREEN_GAMMA** requests.

< The control function is currently not requested to perform this operation. >

Control Operations for Cloning the Driver

Normally, an application's access to the screen is mediated by the Application Server. The application can draw in windows the Server provides through BView objects with graphics environments kept by the Server. The Application Server doesn't let applications communicate directly with the graphics card driver.

To give an application direct access to the screen, as the Game Kit does, the Application Server must get out of the way and the driver must be attached directly to the application. This is accomplished, not by detaching the driver from the Server, but by making a copy of it—a clone—for the application. While the clone is active, the Server suspends its graphic operations.

Graphics card drivers must therefore be prepared to clone themselves—to respond to the four control operations described below. Two of the requests are made of a driver the Application Server has loaded, and two are made of the clone.

B_GET_INFO_FOR_CLONE

This *op* code requests **control_graphics_card()** to write information about the current state of the driver to the location referred to by the *data* pointer. This request is made of a driver loaded by the Application Server; the information it provides is passed to the clone (in a **B_SET_CLONED_GRAPHICS_CARD** request) so that the clone can duplicate the state of the driver.

The driver should package the requested information in a data structure it defines; it can be any structure you desire. The package should include all the driver's variable settings—everything from the current configuration of the screen to the location of the frame buffer in card memory. For example, if the structure is called **info_for_clone**, driver code might look something like this:

```
case B_GET_INFO_FOR_CLONE:
    ((info_for_clone *)data)->depth = info.bits_per_pixel;
    ((info_for_clone *)data)->height = info.height;
    ((info_for_clone *)data)->width = info.width;
    ((info_for_clone *)data)->row_byte = info.bytes_per_row;
    ((info_for_clone *)data)->frame_base = info.frame_buffer;
    ((info_for_clone *)data)->io_base = spec.io_base;
    ((info_for_clone *)data)->available_mem = unused_memory;
    ((info_for_clone *)data)->refresh_rate = rate.current;
    . . .
    break;
```

Of course, information that's kept on the card itself, such as the current color map, does not have to be duplicated for the clone.

Since an attempt is made to keep the driver and its clone in the same state, you can expect numerous **B_GET_INFO_FOR_CLONE** requests while the clone is active.

B_GET_INFO_FOR_CLONE_SIZE

This operation requests the driver to inform the Application Server how many bytes of information it will provide in response to a **B_GET_INFO_FOR_CLONE** request. The control function should write the size of the data structure as a **long** integer in the location that the *data* pointer refers to. For example:

```
*((long *)data) = sizeof(info_for_clone);
```

This information enables the Application Server to allocate enough memory to hold the data it will receive.

B_SET_CLONED_GRAPHICS_CARD

This operation sets up the clone. In the *data* pointer, it passes the clone's **control_graphics_card()** function all the information that the driver provided in response to a **B_GET_INFO_FOR_CLONE** request. The clone should read the information from the *data* pointer and set all the parameters that are provided.

The clone receives a **B_SET_CLONED_GRAPHICS_CARD** request instead of a **B_OPEN_GRAPHICS_CARD** notification when it first is created and loaded by the Game Kit. It subsequently will receive the request many more times—whenever it must be synchronized with the driver loaded by the Application Server.

B_CLOSE_CLONED_GRAPHICS_CARD

This *op* code is passed to the clone's **control_graphics_card()** function to signal that the clone is about to be unloaded. The clone receives this notification instead of **B_CLOSE_GRAPHICS_CARD**. The *data* pointer should be ignored.

Control Operations for Manipulating the Frame Buffer

The BWindowScreen class of the Game Kit defines a set of four functions that give applications more or less arbitrary control over the frame buffer:

ProposeFrameBuffer()
SetFrameBuffer()
SetDisplayArea()
MoveDisplayArea()

Each of these functions translates to an identically named operation that the driver's **control_graphics_card()** function can be requested to perform. Graphics card drivers announce their ability to respond to these requests by including a constant in the **flags** field of the **graphics_card_info** structure they report in response to a **B_GET_GRAPHICS_CARD_INFO** request. The constant is **B_FRAME_BUFFER_CONTROL**.

All four of the control operations use the same structure to pass data to the driver, though they don't all make use the same set of fields within the structure. The structure is called **frame_buffer_info** and it contains the following fields:

short **bits_per_pixel**	The depth of the frame buffer; the number of bits assigned to a pixel.
short **bytes_per_row**	The number of bytes that are used to store one row of pixel data in the frame buffer.
short **width**	The width of the frame buffer in pixels (the total number of pixel columns).
short **height**	The height of the frame buffer in pixels (the total number of pixel rows.
short **display_width**	The width of the screen display in pixels (the number of pixel columns displayed on-screen).
short **display_height**	The height of the screen display in pixels (the number of pixel rows displayed on-screen).
short **display_x**	The pixel column in the frame buffer that's mapped to the leftmost column of pixels on the screen, where columns are indicated by a left-to-right index beginning with 0.
short **display_y**	The pixel row in the frame buffer that's mapped to the topmost row of pixels on the screen, where rows are indicated by a top-to-bottom index beginning with 0.

The first four fields of this structure are identical to the last four of the **graphics_card_info** structure. However, **graphics_card_info** is used only to return information to the host, whereas **frame_buffer_info** can pass requests to the driver. It's possible for those four fields to be set to arbitrary values, so the frame buffer isn't limited to the standard configurations of depth, width, and height described under "**B_GET_SCREEN_SPACES**" above. (Of course, the driver can reject proposed configurations that it can't accommodate.)

The last four fields of the **frame_buffer_info** structure distinguish between the frame buffer itself and the part of the frame buffer that's displayed on-screen—the *display area*. This distinction permits the display area to be moved and resized on a (possibly) much larger area defined by the frame buffer. For buffered drawing, the frame buffer can be partitioned into discrete sections and the display area moved from one to another. For hardware scrolling, the display area can be moved repeatedly by small increments. For simulated zooming, it's size can be incrementally reduced or expanded.

Both areas are defined by a width (the number of pixel columns the area includes) and a height (the number of pixel rows). The display area is located in the frame buffer by the index to the column (**display_x**) and row (**display_y**) of its left top pixel. See the **SetDisplayArea()** function on page 14 in *The Game Kit* chapter for an illustration

The four operations that exercise control over the frame buffer are described below.

B_PROPOSE_FRAME_BUFFER

This *op* code proposes a particular width and depth for the frame buffer to the driver. The only valid fields of the **frame_buffer_info** structure passed through the *data* pointer are **bits_per_pixel** and **width**. If the driver can configure a frame buffer with those dimensions, it should fill in the rest of frame buffer description and return **B_NO_ERROR**. In the **bytes_per_row** field, it should write the minimum number of bytes required to store each row of pixel data given the proposed depth and width. In the **height** field, it should report the maximum number of pixel rows it can provide given the other dimensions. The fields of the **frame_buffer_info** structure that describe the display area can be ignored.

The driver should not actually configure the frame buffer in response to the proposal; it should wait for a **B_SET_FRAME_BUFFER** instruction. **B_PROPOSE_FRAME_BUFFER** merely tests the driver's capabilities.

If the driver can't accommodate a frame buffer with the proposed dimensions, it should place –1 in the **bytes_per_row** and **height** fields and return **B_ERROR**.

B_SET_FRAME_BUFFER

This operation requests the driver's **control_graphics_card()** function to configure the frame buffer according to the description in the **frame_buffer_info** structure passed through the *data* pointer. All fields in the structure contain meaningful values and should be read.

The specified configuration ought to have been previously tested through a **B_PROPOSE_FRAME_BUFFER** operation, and therefore should be one the driver can accommodate. If it's not, **control_graphics_card()** should do nothing and return **B_ERROR**. If it can configure the frame buffer according to the request, it should return **B_NO_ERROR**.

B_SET_DISPLAY_AREA

This *op* code requests the control function to set the display area, as specified by the last four fields of the **frame_buffer_info** structure passed through the *data* pointer. The other fields should be ignored.

If the driver can map the display area as requested, **control_graphics_card()** should return **B_NO_ERROR**. Otherwise, it should return **B_ERROR**.

B_MOVE_DISPLAY_AREA

This *op* code requests the control function to move the display area without resizing it, as specified by the **display_x** and **display_y** fields of the **frame_buffer_info** structure that the *data* pointer refers to. The other fields of the structure should be ignored.

The driver should move the display area so that the left top pixel displayed on-screen is the one located at (**display_x**, **display_y**) in the frame buffer and return **B_NO_ERROR**. If it can't move the display area to that location, it should return **B_ERROR**.

Hook Functions

A graphics card driver can implement hook functions to manage the cursor and perform particular, well-defined drawing tasks on behalf of the Application Server. Drivers should implement as many of these functions as they can to speed on-screen graphics performance.

The driver informs the Application Server about these functions soon after it's loaded when its **control_graphics_card()** function receives a **B_GET_GRAPHICS_CARD_HOOKS** request (see page 79 above). In response to this request, the driver needs to place an array of **B_HOOK_COUNT** (48) function pointers at the location the *data* argument points to. The request is repeated whenever the configuration of the frame buffer (its dimensions and depth) changes. The driver can provide hook functions specific to a particular configuration.

Currently, only the first 12 slots in the array are defined. These functions fall into four groups:

- Indices 0–2: The first three functions define and manage the cursor. Drivers must implement all three of these functions, or none of them. The Application Server defers to driver-defined cursors because of the significant performance improvements they offer.

- Indices 3–9: The next seven hook functions take on specific drawing tasks, such as stroking a minimum-width line or filling a rectangle. You can choose which of these functions to implement.
- Index 10: The function at this index is used to synchronize the Application Server with the driver. Drivers should implement it only if the Server might sometimes need to wait for the driver to finish the drawing undertaken by any of the other hook functions.
- Index 11: The final function inverts the colors in a rectangle.

Each undefined slot in the array of hook functions should be filled with a **NULL** pointer. Similarly, the driver should place a **NULL** value in any defined slot if it can't usefully implement the function.

Although all pointers in the array are declared to be of type **graphics_card_hook**,

typedef void (***graphics_card_hook**)**(**void**)**

each function has its own set of arguments and returns a meaningful error value, declared as a **long**. The functions should be implemented to return **B_NO_ERROR** if all goes well and they're successful in performing the task at hand, and **B_ERROR** if unsuccessful. It's better by far to place a **NULL** pointer in the array than to define a function that always returns **B_ERROR**.

The coordinate system that the Application Server assumes for all hook functions equates one coordinate unit to one screen pixel. The origin is at the pixel in the left top corner of the screen. In other words, an *x* coordinate value is a left-to-right index to a pixel column and a *y* coordinate value is a top-to-bottom index to a pixel row.

The following sections discuss each of the hook functions in turn.

Index 0: Defining the Cursor

The function at index 0 is called to set the cursor image. It has the following syntax:

long **define_cursor(**uchar **xorMask*, uchar **andMask*, long *width*, long *height*, long *hotX*, long *hotY***)**

The first two arguments, *xorMask* and *andMask*, together define the shape of the cursor. Each mask has a depth of 1 bit per pixel, yielding a total of four possible values for each cursor pixel. They should be interpreted as follows:

xorMask	andMask	meaning
0	0	Transparency; let the color of the screen pixel under the cursor pixel show through.
1	0	Inversion; invert the color of the screen pixel.

0	1	White; replace the screen pixel with a white cursor pixel.
1	1	Black; replace the screen pixel with a black cursor pixel.

Inversion in its simplest form is accomplished by taking the complement of the color index or of each color component. For example:

```
color = 255 - color;
```

< However, the results of inversion may not be very pleasing given the current color map. Therefore, none of the Be-defined cursors will use inversion until a future release. It would be better for your drivers to avoid it as well. The color map will be corrected in a future release. >

The second two arguments, *width* and *height*, determine the size of the cursor image in pixels. Currently, the Application Server supports only one cursor size; they must be 16 pixels wide and 16 pixels high.

The (*hotX*, *hotY*) arguments define the hot pixel in the image—the pixel that's used to report the location of the cursor. They assume a coordinate system where the pixel at the left top corner of the image is (0, 0) and the one at the right bottom corner is (15, 15).

This function should change the cursor image on-screen, if the cursor is currently displayed on-screen. But if the cursor is hidden, it should not show it. Wait for explicit calls to the next two functions to move the cursor or change its on-screen status.

Index 1: Moving the Cursor

The function at index 1 changes the location of the cursor image. It should expect two arguments:

long **move_cursor(**long *screenX*, long *screenY***)**

In response, this function should move the cursor so that its hot pixel corresponds to (*screenX*, *screenY*).

Index 2: Showing and Hiding the Cursor

The function at index 2 shows and hides the cursor:

long **show_cursor(**bool *flag***)**

If the *flag* argument is **TRUE**, this function should show the cursor image on-screen; if it's **FALSE**, it should remove the cursor from the screen.

< If this function is asked to show the cursor before the function at index 1 is called, it should show it at (0, 0). >

Index 3: Drawing a Line with an 8-Bit Color

The function at index 3 draws a straight line in the **B_COLOR_8_BIT** color space. It takes 10 arguments:

> long **draw_line_with_8_bit_depth(**long *startX*, long *startY*, long *endX*, long *endY*,
> uchar *colorIndex*, bool *clipToRect*, short *clipLeft*,
> short *clipTop*, short *clipRight*, short *clipBottom***)**

The first four arguments define the starting and ending points of the line; it begins at (*startX*, *startY*) and ends at (*endX*, *endY*). Both points are included within the line. The fifth argument, *colorIndex*, is the color of the line; it's an index into the map of 256 colors.

< In the current release, the second and third arguments are inverted; the first four arguments are ordered: *startX*, *endX*, *startY*, *endY*. >

If the sixth argument, *clipToRect*, is **TRUE**, the function should draw only the portion of the line that lies within the clipping rectangle defined by the last four arguments. The sides of the rectangle are included within the drawing area—they're inside the visible region; everything outside the rectangle is clipped.

If *clipToRect* is **FALSE**, the final four arguments should be ignored.

This function should draw a line of minimal thickness, which means a line no thicker than one pixel at any given point. If the line is more vertical than horizontal, only one pixel per row between the start and end points should be colored; if it's more horizontal than vertical, only one pixel per column should be colored.

Index 4: Drawing a Line with a 32-Bit Color

The function at index 4 is like the one at index 3, except that it draws a line in the **B_RGB_32_BIT** color space:

> long **draw_line_with_32_bit_depth(**long *startX*, long *startY*, long *endX*, long *endY*,
> ulong *color*, bool *clipToRect*, short *clipLeft*,
> short *clipTop*, short *clipRight*, short *clipBottom***)**

The only difference between this and the previous function is the *color* argument. Here the color is specified as a full 32-bit quantity with 8-bit red, green, blue, and alpha components. The *color* argument arranges the components in the order that the driver asked for them (in the **rgba_order** field of the **graphics_card_info** structure that it provided in response to a **B_GET_GRAPHICS_CARD_INFO** request).

Otherwise, this function should work just like the one at index 3. < And like the function at index 3, the second and third arguments are inverted in the current release; the first four arguments are ordered: *startX*, *endX*, *startY*, *endY*. >

Index 5: Drawing a Rectangle with an 8-Bit Color

The function at index 5 should be implemented to fill a rectangle with a color specified by its index:

long **draw_rect_with_8_bit_depth(**long *left*, long *top*, long *right*, long *bottom*, uchar *colorIndex***)**

The *left*, *top*, *right*, and *bottom* sides of the rectangle should be included in the area being filled.

Index 6: Drawing a Rectangle with a 32-Bit Color

The function at index 6, like the one at index 5, fills a rectangle:

long **draw_rect_with_32_bit_depth(**long *left*, long *top*, long *right*, long *bottom*, ulong *color***)**

The *color* value contains the four color components—red, green, blue, and alpha—arranged in the natural order for the device (the same order that the driver recorded in the **rgba_order** field of the **graphics_card_info** structure it provided to the Application Server).

The sides of the rectangle should be included in the area being filled.

Index 7: Copying Pixel Data

The function at index 7 should copy pixel values from a source rectangle on-screen to a destination rectangle:

long **blit(**long *sourceX*, long *sourceY,* long *destinationX,* long *destinationY,* long *width*, long *height***)**

The left top corner of the source rectangle is the pixel at (*sourceX*, *sourceY*). The left top pixel of the destination rectangle is at (*destinationX*, *destinationY*). Both rectangles are *width* pixels wide and *height* pixels high, and both are guaranteed to always lie entirely on-screen. The *width* and *height* arguments will always contain positive values.

Index 8: Drawing a Line Array with an 8-Bit Color

The function at index 8 should draw an array of lines in the **B_COLOR_8_BIT** color space. It takes the following set of arguments:

> long **draw_array_with_8_bit_depth(**indexed_color_line **array*, long *numItems*,
> bool *clipToRect*, short *clipLeft*, short *clipTop*,
> short *clipRight*, short *clipBottom***)**

The line *array* holds a total of *numItems*. Each item is specified as an **indexed_color_line** structure, which contains the following fields:

short **x1**	The *x* coordinate of one end of the line.
short **y1**	The *y* coordinate of one end of the line.
short **x2**	The *x* coordinate of the other end of the line.
short **y2**	The *y* coordinate of the other end of the line.
uchar **color**	The color of the line, expressed as an index into the color map.

The function should draw each line from (**x1**, **y1**) to (**x2**, **y2**) using the **color** specified for that line.

If the *clipToRect* flag is **TRUE**, nothing should be drawn that falls outside the clipping rectangle defined by the final four arguments. The sides of the rectangle are included in the visible region. If *clipToRect* is **FALSE**, the final four arguments should be ignored.

Each line in the array should be drawn with the minimal possible thickness, as described under "Index 3: Drawing a Line with an 8-Bit Color" on page 90 above.

Index 9: Drawing a Line Array with a 32-Bit Color

The function at index 9 has the same syntax as the one at index 8, except for the first argument:

> long **draw_array_with_32_bit_depth(**rgb_color_line **array*, long *numItems*,
> bool *clipToRect*, short *clipLeft*, short *clipTop*,
> short *clipRight*, short *clipBottom***)**

Here, each line in the array is specified as an **rgb_color_line** structure, rather than as an **indexed_color_line**. The two structures differ only in how the color is specified:

short **x1**	The *x* coordinate of one end of the line.
short **y1**	The *y* coordinate of one end of the line.
short **x2**	The *x* coordinate of the other end of the line.
short **y2**	The *y* coordinate of the other end of the line.
rgb_color **color**	The color of the line, expressed as a full 32-bit value.

In all other respects, this function should work like the one at index 8.

Index 10: Synchronizing Drawing Operations

The Application Server calls the function at index 10 to synchronize its activities with the driver. It takes no arguments:

long **sync(**void**)**

This function simply returns when the driver is finished modifying the frame buffer—when it's finished touching video RAM.

If any of the other hook functions works asynchronously—if it returns before the drawing it's asked to do is complete—the synchronizing function should wait until all drawing operations have been completed before it returns. The return value is not important; the Application Server ignores it.

However, if all the other hook functions are synchronous—if they don't return until the drawing is finished—this function would simply return; it would be empty. It's better not to implement such a function. It's preferable to put a **NULL** pointer at index 10 and save the Application Server a function call.

Therefore, your driver should implement this function only if at least one of the other hook functions draws asynchronously.

Index 11: Inverting Colors

The function at index 11 should invert the colors in a rectangle. It has the following syntax:

long **invert_rect(**long *left*, long *top*, long *right*, long *bottom***)**

Inversion is typically defined as taking the complement of each color component. For example:

```
color.red = 255 - color.red;
color.green = 255 - color.green;
color.blue = 255 - color.blue;
```

The inversion rectangle includes the pixel columns and rows that four arguments designate.

Exported Functions

Graphics card drivers are not linked against the Application Server, so the Server cannot export functions to them. They are also not linked against any library. Consequently, they're generally limited to calling functions that they implement themselves.

However, in the current release, a graphics card driver can be statically linked against **scalls.o**, located with the libraries in **/develop/libraries**. < A future release will replace this file with a private library. >

Linking against **scalls.o** makes it possible for the driver to call any system function. The future library won't be as liberal, however, so system calls should be limited to the following functions:

Functions defined in the Kernel Kit

create_sem()
acquire_sem()
release_sem()
delete_sem()

system_time()
snooze()

spawn_thread()
resume_thread()

Functions defined in the Support Kit

atomic_add()

Other functions

dprintf() < accessed through the name **_kdprintf_()** >
set_dprintf_enabled() < accessed through the name **_kset_dprintf_enabled_()** >

Functions from the first two groups are documented in the respective chapters on the Kernel Kit and the Support Kit. Functions in the last group are documented in the section on "Functions for Drivers" in this chapter on page 55.

Installation

The graphics card driver should be compiled as an "add-on image," as described in *The Kernel Kit* chapter. This means following the directions for compiling a shared library presented in the Metrowerks *CodeWarrior* manual. In summary, you'll need to specify the following options for the linker (as **LDFLAGS** in the **makefile**):

- Tell the linker to produce an add-on image by including the **–G** (or **–sharedlibrary**) option.
- Turn off the default behavior—which is to link against the Be system library, **libbe.so**—by specifying the **–nodefaults** flag.
- Export the driver's entry point, **control_graphics_card()**, so that the Application Server can call it. The most direct way to do this is to surround its definition with explicit directives that turn exporting on and off,

  ```
  #pragma export on
  long control_graphics_card(ulong op, void *data)
  {
      . . .
  }
  #pragma export off
  ```

  and inform the linker with an **–export pragma** flag.
- Specify the **scalls.o** file if you want your driver to call any of the system functions listed under "Exported Functions" on page 94 above. Don't link the driver against any other file.

After the driver has been compiled, it should be installed in:

/system/add-ons/app_server

This is the only place where the Application Server will look for graphics card drivers to load.

The Server first looks for a driver for the graphics card in an **app_server** directory on a floppy disk (**/fd/system/add-ons/app_server**). Failing to find one, it looks next on the boot disk (**/boot/system/add-ons/app_server**).

When it searches each disk for a driver, the Application Server begins by looking for one developed specifically for the installed graphics card. If there are more than one, it's indeterminate which one it will choose to load. If there aren't any, the Server looks for the generic driver called **supervga**. This driver should be able to do a minimal job of putting a display on-screen, but probably won't be able to exploit the full potential of the graphics card.

You can give your driver any name you wish. However, the name "supervga" is reserved for the generic driver provided by Be.

Constants and Defined Types for Graphics Card Drivers

This section lists the various constants and types that are defined for graphics card drivers. Explanations for all of them can be found in the preceding section, "Developing a Driver for a Graphics Card".

Constants

Control Operations

<device/GraphicsCard.h>

Enumerated constant	Enumerated constant
B_OPEN_GRAPHICS_CARD	**B_GET_INFO_FOR_CLONE**
B_CLOSE_GRAPHICS_CARD	**B_GET_INFO_FOR_CLONE_SIZE**
B_GET_GRAPHICS_CARD_INFO	**B_SET_CLONED_GRAPHICS_CARD**
B_GET_GRAPHICS_CARD_HOOKS	**B_CLOSE_CLONED_GRAPHICS_CARD**
B_SET_INDEXED_COLOR	
B_GET_SCREEN_SPACES	**B_PROPOSE_FRAME_BUFFER**
B_CONFIG_GRAPHICS_CARD	**B_SET_FRAME_BUFFER**
B_GET_REFRESH_RATES	**B_SET_DISPLAY_AREA**
B_SET_SCREEN_GAMMA	**B_MOVE_DISPLAY_AREA**

These constants define the various control operations that the Application Server or the Game Kit can request a driver to perform.

See also: "Main Control Operations" on page 78

Hook Count

<device/GraphicsCard.h>

Defined constant	Value
B_HOOK_COUNT	48

This constant is the number of hook function pointers that a driver must provide. Most will be **NULL** pointers.

See also: "Hook Functions" on page 87

Info Flags

<device/GraphicsCard.h>

Defined constant

B_CRT_CONTROL
B_GAMMA_CONTROL
B_FRAME_BUFFER_CONTROL

These flags report the driver's ability to control the CRT display, make gamma corrections, and permit nonstandard configurations of the frame buffer.

See also: "**B_GET_GRAPHICS_CARD_INFO**" on page 80

Defined Types

frame_buffer_info

<device/GraphicsCard.h>

```
typedef struct {
        short bits_per_pixel;
        short bytes_per_row;
        short width;
        short height;
        short display_width;
        short display_height;
        short display_x;
        short display_y;
} frame_buffer_info
```

This structure is used to pass information to the driver on how the frame buffer should be configured.

See also: the BWindowScreen class in the Game Kit, "Control Operations for Manipulating the Frame Buffer" on page 85 above

graphics_card_config

<device/GraphicsCard.h>

```
typedef struct {
        ulong space;
        float refresh_rate;
        uchar h_position;
        uchar v_position;
        uchar h_size;
        uchar v_size;
} graphics_card_config
```

This structure is used to pass the driver a set of parameters describing how the graphics card should be configured.

See also: "**B_CONFIG_GRAPHICS_CARD**" on page 82

graphics_card_hook

<device/GraphicsCard.h>

```
typedef void (*graphics_card_hook)(void)
```

This is the general type declaration for a hook function. Specific hook functions will in fact declare various sets of arguments and all return a **long** error code rather than **void**.

See also: "Hook Functions" on page 87

graphics_card_info

<device/GraphicsCard.h>

```
typedef struct {
        short version;
        short id;
        void *frame_buffer;
        char rgba_order[4];
        short flags;
        short bits_per_pixel;
        short bytes_per_row;
        short width;
        short height;
} graphics_card_info
```

Drivers use this structure to supply information about themselves and the current configuration of the frame buffer to the Application Server and to the BWindowScreen class in the Game Kit.

See also: "**B_GET_GRAPHICS_CARD_INFO**" on page 80

graphics_card_spec

<device/GraphicsCard.h>

```
typedef struct {
        void *screen_base;
        void *io_base;
        ulong vendor_id;
        ulong device_id;
        ulong _reserved1_;
        ulong _reserved2_;
} graphics_card_spec
```

This structure informs the driver about the graphics card and how it's mapped into the system.

See also: "**B_OPEN_GRAPHICS_CARD**" on page 78

indexed_color

<device/GraphicsCard.h>

```
typedef struct {
        long index;
        rgb_color color;
} indexed_color
```

This structure is used to set up the list of 256 colors in the **B_COLOR_8_BIT** color space. It locates a particular color at a particular index in the list.

See also: "**B_SET_INDEXED_COLOR**" on page 79

indexed_color_line

<device/GraphicsCard.h>

```
typedef struct {
        short x1;
        short y1;
        short x2;
        short y2;
        uchar color;
} indexed_color_line
```

This structure defines a colored line in the **B_COLOR_8_BIT** color space.

See also: "Index 8: Drawing a Line Array with an 8-Bit Color" on page 92

refresh_rate_info

```
<device/GraphicsCard.h>

typedef struct {
        float min;
        float max;
        float current;
} refresh_rate_info
```

Drivers use this structure to report the current refresh rate, and the maximum and minimum possible rates.

See also: “**B_GET_REFRESH_RATES**” on page 81

rgb_color_line

```
<device/GraphicsCard.h>

typedef struct {
        short x1;
        short y1;
        short x2;
        short y2;
        rgb_color color;
} rgb_color_line
```

This structure defines a colored line in the **B_RGB_32_BIT** color space.

See also: “Index 9: Drawing a Line Array with a 32-Bit Color” on page 92

screen_gamma

```
<device/GraphicsCard.h>

typedef struct {
        uchar red[256];
        uchar green[256];
        uchar blue[256];
} screen_gamma
```

This structure defines the table used to make gamma corrections for the screen display.

See also: “**B_SET_SCREEN_GAMMA**” on page 82

10 The Game Kit

10 The Game Kit

The Game Kit is a collection of software that's especially useful for developing games. Currently, the collection consists of just one class, BWindowScreen, but it will grow in future releases. A BWindowScreen object gives an application direct access to the screen—that is, direct access to the driver for the graphics card so it can bypass the Application Server, customize the card for the game, call graphics functions the driver implements, and draw directly into the frame buffer.

Although designed with games in mind, nothing in the Game Kit is restricted to game applications. Other kinds of applications can profitably take advantage of this Kit.

BWindowScreen

Derived from:	public BWindow
Declared in:	<game/WindowScreen.h>

Overview

A BWindowScreen object has the dual nature its name implies: It's both a window and an object that provides direct access to the screen, bypassing the window system. When a BWindowScreen object becomes the active window—which it does when constructed—it establishes a direct connection to the graphics card driver for the screen, independent of the Application Server. This permits the application to set up a game-specific graphics environment on the card, call driver-implemented drawing functions, and directly manipulate the frame buffer.

The Application Server's graphic operations are suspended until the BWindowScreen object gives up active window status. While it's active, normal drawing operations have no effect; application code can move windows and call upon BView objects to draw, but nothing is rendered on-screen. Only the BWindowScreen object can provide access to the frame buffer.

By constructing a BWindowScreen object, an application takes over the whole screen. The object's frame rectangle is as large as the screen, so that the Application Server will automatically erase every pixel when the window becomes active and refresh everything when it ceases to be the active window. While the BWindowScreen is active, nothing except what the application draws will be visible to the user—no dock and no other windows. The entire screen is the application's canvas.

A BWindowScreen object remains a window while it has control of the screen; it stays attached to the Application Server and its message loop continues to function. It gets messages reporting the user's actions on the keyboard and mouse, just like any other active window. Because it covers the whole screen, it's notified of all mouse and keyboard events. < Messages that report mouse events are currently unreliable; the cursor is reported at a static location, inhibiting mouse-moved messages and making mouse-down and mouse-up messages inaccurate. >

This class respects workspaces. A BWindowScreen object releases its grip on the screen when the user turns to another workspace and reestablishes its control when the user returns to the workspace and it again becomes the active window. Short of quitting the application, changing workspaces is the only way that the user can move in and out of the game. Because other windows and applications aren't visible while the BWindowScreen object is connected to the screen, the usual methods of selecting another application (picking it from the application list or clicking in one of its windows) are not available.

Hook Functions

ScreenConnected()	Can be implemented to do whatever is necessary when the BWindowScreen object obtains direct access to the frame buffer for the screen, and when it loses that access.

Constructor and Destructor

BWindowScreen()

BWindowScreen(const char **title*, ulong *space***)**

Initializes the BWindowScreen object by assigning the window a *title* and specifying a *space* configuration for the screen. The window won't have a visible border or a tab in which to display the title to the user. However, others—such as the Workspaces application—can use the title to identify the window.

The window is constructed to fill the screen; its frame rectangle contains every screen pixel when the screen is configured according to the *space* argument. That argument describes the pixel dimensions and bits-per-pixel depth of the screen that the BWindowScreen object should establish when it first obtains direct access to the frame buffer. It should be one of the following constants:

B_8_BIT_640x480	**B_16_BIT_640x480**	**B_32_BIT_640x480**
B_8_BIT_800x600	**B_16_BIT_800x600**	**B_32_BIT_800x600**
B_8_BIT_1024x768	**B_16_BIT_1024x768**	**B_32_BIT_1024x768**
B_8_BIT_1152x900	**B_16_BIT_1152x900**	**B_32_BIT_1152x900**
B_8_BIT_1280x1024	**B_16_BIT_1280x1024**	**B_32_BIT_1280x1024**
B_8_BIT_1600x1200	**B_16_BIT_1600x1200**	**B_32_BIT_1600x1200**

These are the same constants that can be passed to **set_screen_space()**, the Interface Kit function that preference applications call to configure the screen. < Sixteen-bit depths are not currently supported. >

The constructor assigns the window to the active workspace (**B_CURRENT_WORKSPACE**) and calls **Show()** to immediately place it on-screen, make it the active window, and have it take direct charge of the workspace screen. It fails if another BWindowScreen object in any application already has established a direct screen connection for the same workspace.

To be sure there wasn't an error in constructing the object, call the **Error()** function. If there is an error, it's likely to occur in this constructor, not the inherited BWindow constructor.

Since the object will probably have spawned a thread and will be running a message loop, you'll need to instruct it to quit. For example:

```
MyWindowScreen *screen =
               new MyWindowScreen("Glacier", B_8_BIT_1024x768);
if ( Error() != B_NO_ERROR )
    screen->PostMessage(B_QUIT_REQUESTED);
```

If all goes well, **Error()** will return **B_NO_ERROR** (0).

See also: **Error()**, **get_screen_info()** in the Interface Kit

~BWindowScreen()

virtual **~BWindowScreen(**void**)**

Closes the clone of the graphics card driver (through which the BWindowScreen object established its connection to the screen), unloads it from the application, and cleans up after it.

Member Functions

CanControlFrameBuffer()

bool **CanControlFrameBuffer(**void**)**

Returns **TRUE** if the graphics card driver permits applications to control the configuration of the frame buffer, and **FALSE** if not. Control is exercised through four functions:

ProposeFrameBuffer()
SetFrameBuffer()
SetDisplayArea()
MoveDisplayArea()

A return of **TRUE** means that all of these functions can communicate with the graphics card driver and at least the first two of them will do something useful. A return of **FALSE** means that none of them will work.

See also: **ProposeFrameBuffer()**, **SetDisplayArea()**

CardHookAt()

inline graphics_card_hook **CardHookAt(**long *index***)**

Returns a pointer to the function implemented by the graphics card driver and located at *index* in its list of hook functions, or **NULL** if the graphics card driver doesn't implement a function at that index or the index is out-of-range.

The hook functions are documented under “Hook Functions” on page 87 in *The Device Kit* chapter and are summarized briefly below. Currently, 12 functions are defined, from index 0 through index 11. However, the first three, which set and manipulate the cursor, are unavailable through the Game Kit; if you pass an index of 0, 1, or 2 to **CardHookAt()**, it will return **NULL**.

The other hook functions are summarized by index in the chart below:

Index:	What the function does:	What arguments it takes:
3	Draws a line (8-bit depth)	(long *startX*, long *startY*, long *endX*, long *endY*, uchar *colorIndex*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
4	Draws a line (32-bit depth)	(long *startX*, long *startY*, long *endX*, long *endY*, ulong *color*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
5	Draws a rectangle (8-bit depth)	(long *left*, long *top*, long *right*, long *bottom*, uchar *colorIndex*)
6	Draws a rectangle (32-bit depth)	(long *left*, long *top*, long *right*, long *bottom*, ulong *color*)
7	Copies pixel data (blits)	(long *sourceX*, long *sourceY*, long *destinationX*, long *destinationY*, long *width*, long *height*)
8	Draws a line array (8-bit depth)	(indexed_color_line **array*, long *numItems*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
9	Draws a line array (32-bit depth)	(rgb_color_line **array*, long *numItems*, bool *clipToRect*, short *clipLeft*, short *clipTop*, short *clipRight*, short *clipBottom*)
10	Waits for drawing to finish	*none*
11	Inverts the colors in a rectangle	(long *left*, long *top*, long *right*, long *bottom*)

You must ensure that all coordinate values passed to these functions lie somewhere in the frame buffer; the function will not do the checking for you. (An *x* coordinate value is a left-to-right index to a pixel column in the frame buffer and a *y* coordinate value is a top-to-bottom index to a pixel row.)

For example, before calling the function at index 7, which blits a rectangle *width* pixels wide and *height* pixels high from (*sourceX*, *sourceY*) to (*destinationX*, *destinationY*), you should be sure that the source and destination rectangles both lie entirely within the area defined by the frame buffer.

CardInfo()

inline graphics_card_info ***CardInfo(**void**)**

Returns a description of the current configuration of the graphics card, as kept by the driver for the card. The returned **graphics_card_info** structure is defined in **device/GraphicsCard.h** and contains the following fields:

short **version**	The version of the Be architecture for graphics cards; the current version is 2.
short **id**	An identifier for the driver.
void ***frame_buffer**	A pointer to the first byte of the frame buffer. Applications can use this pointer to draw directly to the frame buffer.
char **rgba_order**[4]	The characters ‘r’ (red), ‘g’ (green), ‘b’ (blue), and ‘a’ (alpha) ordered as those components are intermeshed for each pixel in the frame buffer. This field is valid only for screen depths of 32 bits per pixel.
short **flags**	A mask formed from three flags (**B_CRT_CONTROL**, **B_FRAME_BUFFER_CONTROL**, and **B_GAMMA_CONTROL**) that describe the ability of the graphics card driver to perform particular tasks. **B_FRAME_BUFFER_CONTROL** matches the **CanControlFrameBuffer()** function; the other two flags aren’t important to the control exercised through the BWindowScreen object.
short **bits_per_pixel**	The depth of the screen in bits per pixel.
long **bytes_per_row**	The offset, in bytes, between two adjacent rows of pixel data in the frame buffer (the number of bytes assigned to each row).
short **width**	The width of the frame buffer in pixels (the number of pixel columns it defines).
short **height**	The height of the frame buffer measured in lines of pixels (the number of pixel rows the frame buffer defines).

The returned structure belongs to the BWindowScreen object and is provided for information only; you should not modify any of its fields.

See **“B_GET_GRAPHICS_CARD_INFO”** on page 80 in *The Device Kit* chapter for a fuller description of the **graphics_card_info** structure.

ColorList() *see* SetColorList()

Disconnect() *see* Quit()

Error()

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

Returns the error code for the last BWindowScreen operation—including constructing the BWindowScreen object. The code will be **B_NO_ERROR** if the operation was successful and some other value (currently just **B_ERROR**) if not. Most functions also return the error code directly.

See also: the BWindowScreen constructor

FrameBufferInfo()

inline frame_buffer_info ***FrameBufferInfo(**void**)**

Returns a pointer to the **frame_buffer_info** structure that holds the application's current conception of the frame buffer. This may or may not capture the actual configuration of the frame buffer. If the application has proposed a configuration (**ProposeFrameBuffer()**) but not yet set it (**SetFrameBuffer()**), the returned structure will reflect the proposal, not the reality.

The **frame_buffer_info** structure is defined in **device/GraphicsCard.h** and contains the following fields:

short **bits_per_pixel**	The depth of the frame buffer; the number of bits assigned to a pixel.
short **bytes_per_row**	The number of bytes required to store one row of pixel data in the frame buffer.
short **width**	The width of the frame buffer in pixels (the number of columns).
short **height**	The height of the frame buffer in pixels (the number of rows).
short **display_width**	The width of the screen display in pixels (the number of pixel columns shown on-screen).
short **display_height**	The height of the screen display in pixels (the number of pixel rows shown on-screen).

short **display_x**	The horizontal position of the left top pixel shown on-screen, where 0 is the leftmost column of pixels in the frame buffer.
short **display_y**	The vertical position of the left top pixel shown on-screen, where 0 is the topmost row of pixels in the frame buffer.

Note that the first four fields of this structure are identical to the last four of **graphics_card_info**.

The returned structure belongs to the BWindowScreen object. Call functions like **ProposeFrameBuffer()** to modify its fields; don't modify them directly.

See "Control Operations for Manipulating the Frame Buffer" on page 85 in *The Device Kit* chapter for a fuller description of the **frame_buffer_info** structure.

See also: **ProposeFrameBuffer()**, **SetDisplayArea()**, **CardInfo()**

IOBase()

inline void ***IOBase(**void**)**

Returns a pointer to the base address for the input/output registers on the graphics card. Registers are addressed by 16-bit offsets from this base address.

MoveDisplayArea() *see* SetDisplayArea()

ProposeFrameBuffer(), SetFrameBuffer()

long **ProposeFrameBuffer(**short *depth*, short *width*,
short **height*, short **bytesPerRow* = NULL**)**

long **SetFrameBuffer(**short *height***)**

long **SetFrameBuffer(**short *height*, short *displayWidth*, short *displayHeight*,
short *displayX* = 0, short *displayY* = 0**)**

These functions, in a two-step process, configure the frame buffer on the graphics card. They work only if the driver for the graphics card allows custom configurations (as reported by **CanControlFrameBuffer()**).

The first function proposes a possible configuration for the frame buffer and in return receives information on how accommodating the graphics card driver will be. Based on that information, the second function can set the dimensions and depth of the frame buffer.

ProposeFrameBuffer() proposes two parameters for the frame buffer—*depth*, the number of bits assigned to each pixel, and *width*, the number of pixels in one row of data (the total number of pixel columns). If the driver can accommodate those two parameters, this function returns **B_NO_ERROR** and reports on the two other parameters that define the configuration. In the integer referred to by *height*, it writes the maximum number of pixel rows (the maximum number of pixels in one column) that the graphics card can provide at the proposed depth and width. If a *bytesPerRow* argument is provided, it reports the minimum number of bytes the driver must dedicate to one row of pixel data at the proposed width.

If the driver can't accommodate the proposed *depth* and *width*, **ProposeFrameBuffer()** returns **B_ERROR** and puts no useful information in the integers that *height* and *bytesPerRow* refer to.

An application can call **ProposeFrameBuffer()** any number of times to test possible configurations. This function doesn't make any changes in the frame buffer (though it does set values in the structure that **FrameBufferInfo()** returns). When the application finds a configuration that it wants to use, it can call **SetFrameBuffer()** to set the desired *height* of the frame buffer—the actual number of rows—plus the most recently proposed *depth* and *width*. The height set should not be greater than the maximum height reported by **ProposeFrameBuffer()**.

By default, **SetFrameBuffer()** sets the display area—the part of the frame buffer that's mapped to the screen—to be the same size as the frame buffer. In other words, it maps the entire frame buffer to the screen.

If you want a display area that's smaller than the frame buffer, you must set it explicitly by passing **SetFrameBuffer()** a *displayWidth* and *displayHeight* in pixels. The left top corner of the display area will be located at pixel (0, 0), the first pixel in the first row of data. If you want to locate it somewhere else, you must pass this function different *displayX* and *displayY* values.

The display area can subsequently be moved and resized through the **MoveDisplayArea()** and **SetDisplayArea()** functions.

To locate the display area, all these functions assume a coordinate system in which an *x* coordinate value is a left-to-right index to a pixel column in the frame buffer and a *y* coordinate value is a top-to-bottom index to a pixel row.

See also: **SetDisplayArea()**

Quit(), Disconnect()

virtual void **Quit(**void**)**

void **Disconnect(**void**)**

Quit() overrides the BWindow version of the same function to force the BWindowScreen object to disconnect itself from the screen, so that it doesn't quit while in control of the frame buffer.

Disconnect() similarly causes the BWindowScreen object to give up its authority over the graphics card driver, allowing the Application Server to reassert control. It doesn't force the application to quit.

Although **Quit()** disconnects the object before quitting, this may not be soon enough for your application. For example, if you need to destroy some drawing threads before the BWindowScreen object is itself destroyed, you should get rid of them after the screen connection is severed. You can force the object to disconnect itself by calling **Disconnect()**. For example:

```
void MyWindowScreen::Quit()
{
    Disconnect();
    kill_thread(drawing_thread_a);
    kill_thread(drawing_thread_b);
    BWindowScreen::Quit();
}
```

Before breaking the screen connection, both **Quit()** and **Disconnect()** cause the BWindowScreen object to receive a **ScreenConnected()** notification with a flag of **FALSE**. Neither function returns until **ScreenConnected()** returns and the connection is broken.

See also: **ScreenConnected()**

ScreenChanged()

virtual void **ScreenChanged(**BRect *frame*, color_space *mode***)**

Overrides the BWindow version of **ScreenChanged()** so that it does nothing. This function is called automatically when the screen configuration changes. It's not one that you should call (or override) in application code.

See also: **BWindow::ScreenChanged()**

ScreenConnected()

virtual void **ScreenConnected(**bool *connected***)**

Implemented by derived classes to take action when the application gains direct access to the screen and when it's about to lose that access.

This function is called with the *connected* flag set to **TRUE** immediately after the BWindowScreen object becomes the active window and establishes a direct connection to the graphics card driver for the screen. At that time, the Application Server's connection to the screen is suspended; drawing can only be accomplished through the screen access that the BWindowScreen object provides.

It's called with a flag of **FALSE** just before the BWindowScreen object is scheduled to lose its control over the screen and the Application Server's control is reasserted. The BWindowScreen's connection to the screen is not broken until **ScreenConnected()**

returns. It should delay returning until the application has finished all current drawing and no longer needs direct screen access.

Note that whenever **ScreenConnected()** is called, the BWindowScreen object is guaranteed to be connected to the screen; if *connected* is **TRUE**, it just became connected, if *connected* is **FALSE**, it's still connected but will be disconnected when the function returns.

Derived classes typically use this function to regulate access to the screen. For example, they may acquire a semaphore when the *connected* flag is **FALSE**, so that application threads won't attempt direct drawing when the connection isn't in place, and release the semaphore for drawing threads to acquire when the flag is **TRUE**. For example:

```
void MyWindowScreen::ScreenConnected(bool connected)
{
    if ( connected == FALSE )
        acquire_sem(directDrawingSemaphore);
    else
        release_sem(directDrawingSemaphore);
}
```

SetColorList(), ColorList()

void **SetColorList(**rgb_color **colors*, long *first* = 0, long *last* = 255**)**

inline rgb_color ***ColorList(**void**)**

These functions set and return the list of 256 colors that can be displayed when the frame buffer has a depth of 8 bits per pixel (the **B_COLOR_8_BIT** color space). **SetColorList()** passes an array of one or more colors to replace *colors* currently in the list. The first color in the array replaces the color at the specified *first* index in the list; the last color that's passed replaces the color at the *last* index. **ColorList()** returns a pointer to the entire list of 256 colors.

SetColorList() alters the list of colors kept on the graphics card. **ColorList()** doesn't return a pointer to that list, but to a local copy. This list belongs to the BWindowScreen object; it should be altered only by calling **SetColorList()**.

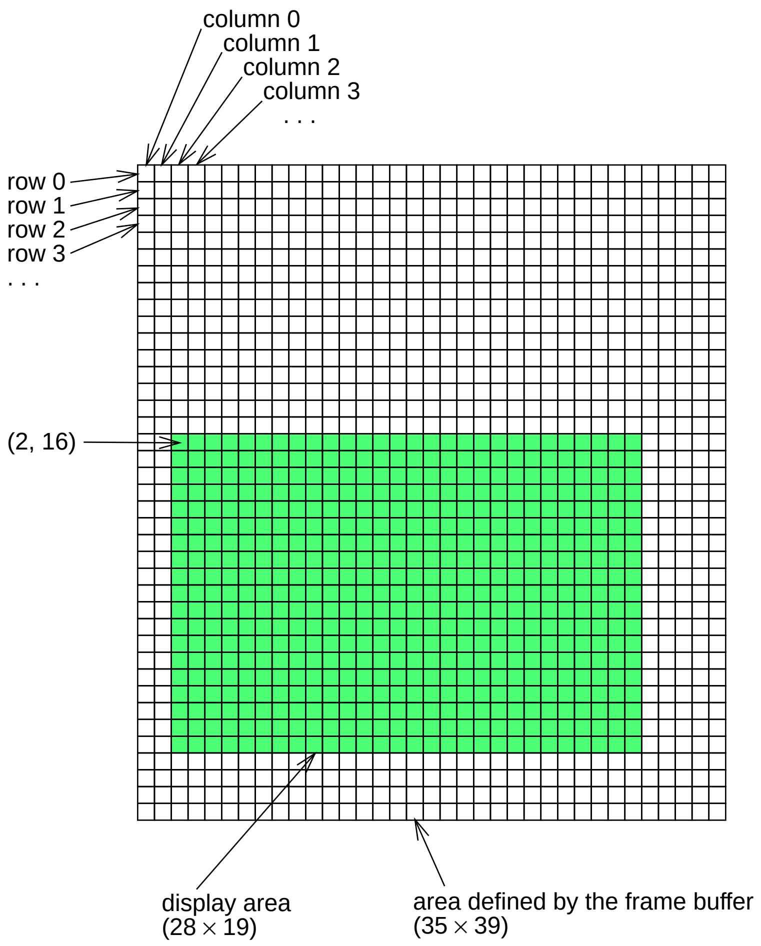
See also: **system_colors()** in the Interface Kit

SetDisplayArea(), MoveDisplayArea()

long **SetDisplayArea(**short *width*, short *height*, short *x* = 0, short *y* = 0**)**

long **MoveDisplayArea(**short *x*, short *y***)**

These functions resize and move the display area, the portion of the frame buffer that's mapped to the screen. The area is defined by a rectangle *width* pixels wide and *height* pixels high located entirely within the frame buffer, as illustrated in miniature below. The left top pixel in the rectangle is located at (*x*, *y*), where *x* coordinate values are left-to-right indices to a pixel column defined by the frame buffer and *y* coordinate values are top-to-bottom indices to a pixel row.



For example, the frame buffer might define twice as many pixel rows as the screen displays, so the display area can alternate between the top and bottom halves of the frame buffer for a smooth transition between images. Or the dimensions of the display area can be incrementally reduced to simulate a zoom effect as the size of on-screen pixels becomes bigger.

Like **ProposeFrameBuffer()** and **SetFrameBuffer()**, these functions work only if the graphics card driver permits application control over the frame buffer. It must also permit a display area that's smaller than the total area the frame buffer defines. If successful in moving or resizing the display area, they return **B_NO_ERROR**; if not, they return **B_ERROR**.

See also: **ProposeFrameBuffer()**, **CanControlFrameBuffer()**

SetFrameBuffer() *see* ProposeFrameBuffer()

SetSpace()

long **SetSpace(**ulong *space***)**

Configures the screen space to one of the standard combinations of width, height, and depth. The configuration is first set by the class constructor—permitted *space* constants are documented there—and it may be altered by the **SetFrameBuffer()** function in addition to this one.

If the requested configuration is refused by the graphics card driver, this function returns **B_ERROR**. If all goes well, it returns **B_NO_ERROR**.

See also: the BWindowScreen constructor, **ProposeFrameBuffer()**

WindowActivated()

virtual void **WindowActivated(**bool *active***)**

Overrides the BWindow version of **WindowActivated()** to connect the BWindowScreen object to the screen (give it control over the graphics card driver) when the *active* flag is **TRUE**.

This function doesn't disconnect the BWindowScreen when the flag is **FALSE**, because there's no way for the window to cease being the active window without the connection already having been lost.

Don't reimplement this function in your application, even if you call the inherited version; rely instead on **ScreenConnected()** for accurate notifications of when the BWindowScreen gains and loses control over the screen.

See also: **BWindow::WindowActivated()**, **ScreenConnected()**

WorkspaceActivated()

virtual void **WorkspaceActivated(**long *workspace*, bool *active***)**

Overrides the BWindow version of **WorkspaceActivated()** to connect the BWindowScreen object to the screen when the *active* flag is **TRUE** and to disconnect it when the flag is **FALSE**. User's typically activate the game by activating the workspace in which it's running, and deactivate it by moving to another workspace.

Don't override this function in your application; implement **ScreenConnected()** instead.

See also: **BWindow::WorkspaceActivated()**, **ScreenConnected()**

11 The Network Kit

11 The Network Kit

The Network Kit is divided into two domains:

- The Kit provides a collection of global C functions that let you communicate with other computers through the TCP or UDP protocols. With a few exceptions, the names and intents of the functions adhere to the precedent set by the BSD network/socket implementation. Note, however, that some BSD-defined functions are not yet implemented.

- The Kit also provides C functions and a class (BMailMessage) that let you talk to the mail daemon, and send and receive mail messages. With the functions and the class, you should be able to write a fully-featured mail-reading and -writing application.

The network and socket documentation can be found in the sections “Network Names, Addresses, and Services” on page 5, and “Network Sockets” on page 13. In addition, you can find some further socket examples and tips in the “Be Engineering Insights” column of the Be Newsletter, issues 19 and 30.

Functions that access the mail daemon, the process that makes the mail system run, are documented in “The Mail Daemon” on page 29. The BMailMessage class is documented in “Mail Messages (BMailMessage)” on page 37.

Network Names, Addresses, and Services

Declared in:	<net/netdb.h> <net/socket.h>

Overview

The functions described below let you look up the names, addresses, and other information about the computers and services that the local computer knows about, and let you retrieve information about the current user's account. Also defined here are functions that perform *Internet Protocol* (IP) address format conversion.

You use the functions defined here to find the information you need so you can form a connection to some other machine. Connecting to other machines is described in "Network Sockets" on page 13.

Terms and Tools

Throughout the following function descriptions, an *IP address* is the familiar four-byte, dot-separated numeric identifier. For example,

```
192.0.0.1
```

The bytes in a multi-byte address are always given in *network byte order* (big-endian). The current BeBox is also big-endian, so you don't have to convert IP address values—but for portability and forward-compatibility, you may want to. See the group of functions with the obsessively shortened names (**ntohs()**, **htohl()**, etc.) for more information on such transformations.

An *IP name* is the three-element "machine.domain.extension" case-insensitive text name:

```
decca.be.com
```

The two most important functions described below, **gethostbyname()** and **gethostbyaddr()**, retrieve information about computers ("hosts") that can be reached through the network. Host information is typically (and primarily) gotten from the *Domain Name Server* (DNS), a service that's usually provided by a server computer that's responsible for tasks such as mail distribution and direct communication with the *Internet Provider Service* (IPS).

You can also provide host information by adding to your computer's **/boot/system/hosts** file. This is a text file that contains the IP addresses and names of the hosts that you want your computer to know about. Each entry in the file lists, in order on a single line, a host's IP address, IP name, and other names (aliases) by which it's also known. For example:

```
# Example /boot/system/hosts entries
192.0.0.1 phaedo.racine.com fido phydough
205.123.5.12 playdo.mess.com plywood funfactory
```

The amount of whitespace separating the elements is arbitrary. The only killing point is that there mustn't be any leading whitespace before the IP address.

If you're connected to DNS, then you shouldn't need the **hosts** file. If you're not connected to a network at all, the only way to get information about other machines is through the **hosts** file, but it won't do you much good—you won't be able to use the information to connect to other machines. The archetypal situation in which the **hosts** file becomes useful is if your BeBox is connected to some other machine (we'll call it Brand X), and the Brand X machine is supposed to be connected to a DNS machine, but this latter connection is down (or the DNS machine isn't running). If you have an entry in your BeBox **hosts** file that identifies the Brand X machine, you'll still be able to look up the machine's address and connect to it, despite the absence of DNS.

Functions

gethostbyname(), gethostbyaddr(), herror()

struct hostent ***gethostbyname(**const char **name***)**
struct hostent ***gethostbyaddr(**const char **address*, int *length*, int *type***)**
void **herror(**const char **string***)**

The two **gethostby...()** functions retrieve information about a particular host machine, stuff the information into a global "host entry" structure, and then return a pointer to that structure. To get this information, the functions talks to the Domain Name Server. If DNS doesn't respond or doesn't know the desired host, the functions then look for an entry in the file **/boot/system/hosts**. See "Terms and Tools" on page 5 for more information on DNS and the **hosts** file.

herror() generates a human-readable message that describes the most recent **gethostby...()** error, and prints it to standard error.

Note: Because **gethostbyname()** and **gethostbyaddr()** use a global structure to return information, the functions are *not* thread safe.

The gethostbyname() Function

gethostbyname()'s *name* argument is a **NULL**-terminated, case-insensitive host name that must be no longer than **MAXHOSTNAMELEN** (64) characters (not counting the **NULL**). The name can be:

- An entire "machine.domain.extension" IP name—"mybox.me.com", for example.
- Just the machine name portion—"mybox" (DNS only). In this case, the domain and extension of the local machine are automatically appended. (If you're looking up an IP name in the **hosts** file, the domain and extension *aren't* appended for you.)
- A host name alias. Aliases are alternate names by which a host is known. Your DNS should provide a means for declaring aliases; you can also declare them in your **hosts** file.

The gethostbyaddr() Function

gethostbyaddr()'s *address* argument is a pointer to a complete IP address given in its natural format (but cast to a **char ***; note that the argument's type declaration *doesn't* mean that the function wants the address converted to a string). *length* is the length of *address* in bytes; *type* is a constant that gives the format of the address.

For IP format, the first argument is a four-byte integer, *length* is always 4, and type is **AF_INET** ("*A*ddress *F*ormat: *I*nter*NET*"). The following gets the **hostent** for a hard-coded address:

```
/* This is the hex equivalent of 192.0.0.1
ulong addr = 0xc0000001;
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

If you have an address stored as a string, you can use the **inet_addr()** function to convert it to an integer:

```
ulong addr = inet_addr("192.0.0.1");
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

The hostent Structure

If a **gethostby...()** function fails, it returns **NULL**; otherwise, it returns a pointer to a global **hostent** structure. The **hostent** structure (which isn't **typedef**'d) looks like this:

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addrtype;
    int h_length;
    char **h_addr_list;
};
```

The fields are:

- **h_name** is the IP name of the host (or the “official” name given in the **hosts** file).
- **h_aliases** is a **NULL**-terminated array of other names by which the host is known. These names aren’t necessarily in IP name format; typically, they’re single-word names.
- **h_addrtype** identifies the format of the addresses listed in **h_addr_list**. Currently, the type is always **AF_INET**.
- **h_length** is the length, in bytes, of the host’s address. In **AF_INET** format, the address is four bytes long
- **h_addr_list** is a **NULL**-terminated array of pointers to the addresses by which the host is known. Host addresses are given in network byte order.

As a convenience, the global **h_addr** constant is a fake field that points to the first item in the **h_addr_list** field. Keep in mind that **h_addr** must be treated as a structure field—it must point off a **hostent** structure. Also, make sure you dereference the **h_addr** “field” properly. For example:

```
ulong ip_address;
struct hostent *theHost;

theHost = gethostbyname("fido");
ip_address = *(ulong *)theHost->h_addr;
```

As a demonstration of the **h_addr** definition, the final line is the same as

```
ip_address = *(ulong *)theHost->h_addr_list[0];
```

Keep in mind that the **hostent** structure that’s pointed to by the **gethostby...()** functions is global to your application’s address space. If you want to cache the structure, you should copy it as soon as it’s returned to you.

h_errno and the herror() Function

The host look-up functions use a global error variable (an integer), called **h_errno**, to register errors. You can look at the **h_errno** value directly in your code after a host function fails (the potential **h_errno** values are listed below). Alternatively, you can use the **herror()** function which prints, to standard error, its argument followed by a system-generated string that describes the current state of **h_errno**.

The values that **h_errno** can take, and the corresponding **herror()** strings, are:

Value	Meaning
HOST_NOT_FOUND	"unknown host name"
TRY_AGAIN	"host name server busy"
NO_RECOVERY	"unrecoverable system error"
NO_DATA	"no address data is available for this host name"
anything else	"unknown error"

Note that while **h_errno** is set when something goes wrong, it isn't cleared if all is well. For example, if **gethostbyname()** can't find the named host, **h_errno** is set to **HOST_NOT_FOUND** and the function returns **NULL**. If, in an immediately subsequent call, the function succeeds, a pointer to a valid **hostent** is returned, but **h_errno** will *still* report **HOST_NOT_FOUND**.

The moral of this tale is that you should *only* check **h_errno** (or call **herror()**) if the network function call has failed, or clear it yourself before each **gethostby...()** call. Or both:

```
struct hostent *host_ent;

h_errno = 0;
if ( !(host_ent = gethostbyname("a.b.c"))
    herror("Error");
```

Furthermore, **h_errno** might be legitimately set to a new error code even if the **gethostby...()** function succeeds. For example, if DNS can't be reached but the desired host is found in the **hosts** file, **h_errno** will be set to **TRY_AGAIN**, yet the returned **hostent** will be legitimate (it won't be **NULL**).

Be aware that **TRY_AGAIN** is used as a blanket "DNS doesn't know" state, *regardless* of the reason why. In other words, **h_errno** is set to **TRY_AGAIN** if DNS is actually down, if your machine isn't connected to the network, or if DNS simply doesn't know the requested host. You can use this fact to tell whether a (successful) look-up was performed through DNS or the **hosts** file:

```
struct hostent *host_ent;

h_errno = 0;
if ( !(host_ent = gethostbyname("a.b.c"))
    herror("Error");
else
    if (h_errno == TRY_AGAIN)
        /* The hosts file was used. */
    else
        /* DNS was used. */
```

Keep in mind that **h_errno** is global; be careful if you're using it in a multi-threaded program.

gethostname(), getusername(), getpassword()

int **gethostname(**char **name*, unsigned int *length***)**
int **getusername(**char **name*, unsigned int *length***)**
int **getpassword(**char **password*, unsigned int *length***)**

These functions retrieve, and copy into their first arguments, the name of the local computer, the name of the current user, and the current user's encoded password, respectively. In all three case, *length* gives the maximum number of characters that the functions should copy. If the length of the desired element is less than *length*, the copied string will be **NULL**-terminated.

The functions return the number of characters that were actually copied (not counting the **NULL** terminator). If there's an error—and such should be rare—the **gethostname()** and **getusername()** functions return 0 and point their respective name arguments to **NULL**. **getpassword()**, sensing an error, copies "*" into the password argument and returns -1 (thus you can tell the difference between a **NULL** password—which would legitimately return 0—and an error).

All three bits of information (host name, user name, and password) are taken from the settings that are declared through the **Network** preferences application.

A typical use of **gethostname()** is to follow the call with **gethostbyname()** in order to retrieve the address of the local host, as shown below:

```
/* To fill a need, we invent the gethostaddr() function. */
long gethostaddr(void)
{
    struct hostent *host_ent;
    char host_name[MAXHOSTNAMELEN];

    if (gethostname(host_name, MAXHOSTNAMELEN) == 0)
        return -1;

    if ((host_ent = gethostbyname(host_name)) == NULL)
        return -1;

    return *(long *)host_ent.h_addr;
}
```

Keep in mind that since host name information is taken from Network preferences, there's no guarantee that the name that's returned by **gethostname()** will match an entry that DNS or the **hosts** file knows about.

getservbyname()

struct servent ***getservbyname(**const char **name***,** const char **protocol***)**

You pass in the name of a service (such as "ftp") that runs under a particular protocol (such as "tcp"), and **getservbyname()** returns a pointer to a **servent** structure that describes the service.

The **servent** structure is:

```
struct servent {
    char *s_name;
    char **s_aliases;
    int s_port;
    char *s_proto;
};
```

- **s_name** is the name of the service.
- **s_aliases** is a **NULL**-terminated array of other names by which the services is known.
- **s_port** is the port number on which the service runs (given in network byte order)
- **s_proto** names the protocol ("tcp", "udp", etc.) that supports the service.

Currently, the function recognizes only two services: "ftp" and "telnet". Both run under the "tcp" protocol; thus, the only valid calls to **getservbyname()** are:

```
getservbyname("ftp", "tcp");
```

and

```
getservbyname("telnet", "tcp");
```

Such calls point to (separate) pre-defined **servent** structures that look like this:

field	ftp structure	telnet structure
s_name	"ftp"	"telnet"
s_aliases	**NULL**	**NULL**
s_port	21	23
s_proto	"tcp"	"tcp"

If you ask for a service other than these two, the function returns **NULL**. Although the two **servent** structures are separate entities, they are both global to your application. In theory, this means the **getservbyname()** function isn't thread-safe. However, since the structures are hard-coded and separate, there's little danger in using them unprotected in a multi-threaded program.

inet_addr(), inet_ntoa()

unsigned long **inet_addr(**const char **addr***)**
char ***inet_ntoa(**struct in_addr *addr***)**

These functions convert addresses from ASCII to IP format and vice versa. Neither of them consults the DNS or the hosts file to perform the conversion—in other words, they perform the conversions without regard for an address' correspondence to an actual machine.

inet_addr() converts from ASCII to IP:

```
ulong addr = inet_addr("192.0.0.1");
```

The result of this call (**addr**) would be appropriate as the initial argument to **gethostbyaddr()** (for example). The returned address is in network byte order.

inet_ntoa() converts the other way: It takes an IP address and converts it into an ASCII string. Note that the address that you pass in must first be placed in the **s_addr** field of the argument **in_addr** structure (**s_addr** is the structure's only field). For example:

```
in_addr addr;
char addr_buf[16];

addr.s_addr = 0xc0000001;
strcpy(addr_buf, inet_ntoa(addr));
```

Here, **addr_buf** will contain the (**NULL**-terminated) string "192.0.0.1". **inet_ntoa()** isn't thread-safe; if you want to cache the string that it returns you must copy it, as shown in the example. Given the IP format, the string that **inet_ntoa()** returns is guaranteed to be no more than 16 characters long (four 3-character address components, three dots, and a **NULL**).

ntohs(), ntohl(), htons(), htonl()

short **ntohs(**short *val***)**
long **ntohl(**long *val***)**
short **htons(**short *val***)**
long **htonl(**long *val***)**

These macros convert values between host and network byte order:

Macro	Meaning
ntohs()	network short to host short
ntohl()	network long to host long
htons()	host short to network short
htonl()	host long to network long

Network byte order is big-endian; the host byte order is machine-dependent. The current BeBox is big-endian, so these macros are, essentially, no-ops: They return their respective arguments without conversion. To be scrupulous, however, you should convert all multi-byte values that you write to or get from the Internet. For example, a truly "safe" call to **gethostbyaddr()** (for example) would look like this:

```
ulong addr = htonl(inet_addr("192.0.0.1");
struct hostent *theHost;

theHost = gethostbyaddr((char *)&addr, 4, AF_INET);
```

Network Sockets

Declared in: <net/socket.h>

Overview

Sockets are entry ways onto a network. To transmit data to another machine, you create a socket, tell it how to find the other computer, and then tell it to send. To receive data, you do the opposite: You create a socket, tell it who to listen to (in some cases), and then wait for data to come pouring in.

Socket concepts are mixed in with regular function descriptions; the **socket()** function, which is where any socket user must start, is described first. The description gives a general overview of the different types of sockets, how you use them, and where to go to next. The other socket functions are then listed in a separate section, in the expected alphabetical order.

The socket implementation (and philosophy) follows the precedent established by 4.2BSD. In particular, the API presented here bends many of the Be naming and calling conventions in order to make porting existing programs easier.

The socket() Function

socket(), closesocket()

int **socket(**int *family*, int *type*, int *protocol***)**

int **closesocket(**int *socket***)**

The **socket()** function returns a token (a non-negative integer) that represents the local end of a connection to another machine. Freshly returned, the token is abstract and unusable; to put the token to use, you have to pass it as an argument to other functions—such as **bind()** and **connect()**—that know how to establish a connection (however temporary) over the network. (The function's arguments are examined in a separate section, below.)

A successful **socket()** call returns a non-negative integer—keep in mind that 0 is a valid socket token. Also keep in mind that socket tokens are *not* file descriptors (this violates the BSD tradition). Upon failure, **socket()** returns -1 and sets the global **errno** variable to one of these values:

Value	Meaning
EAFNOSUPPORT	*format* was other than **AF_INET.**
EPROTOTYPE	*type* and *protocol* mismatch.
EPROTONOSUPPORT	Unrecognized *type* or *protocol* value.

closesocket() closes a socket's connection (if it's the type of socket that can hold a connection) and frees the resources that have been assigned to the socket. When you're done with the sockets that you've created, you should pass each socket token to **closesocket()**—no socket, no matter how abstract or how you use it, is exempt from the need to be closed. In regard to this universal need, you should be aware that this extends to sockets that are created through the **accept()** function (which we'll get to later).

closesocket() returns less-than-zero if its argument is invalid.

The socket() Arguments

socket()'s three arguments, all of which take predefined constants as values, describe the type of communication the socket can handle:

- *family* takes a constant the describes the network address format that the socket understands. Currently, it must be **AF_INET** (the Internet address format).
- The *type* constant must be either **SOCK_STREAM** or **SOCK_DGRAM**. The constant describes (roughly) the "persistence" of the connection that can be formed through this socket. The **SOCK_STREAM** constant means the impending connection (which is formed through a **connect()** or **bind()** call) will remain open until told to close. **SOCK_DGRAM** describes a "datagram" socket; the connection through a datagram socket is open while data is being sent (typically through **sendto()**) or received (similarly, **recvfrom()**). It's closed at all other times (note, however, that you still have to call **closesocket()** on a datagram socket when you're done with it).
- *protocol* describes the "messaging" protocol, a description that's closely related to the socket type. Although there are three *protocol* constants (**IPPROTO_TCP**, **IPPROTO_UDP**, and **IPPROTO_ICMP**), values that you would actually use are either 0 or, less commonly, **IPPROTO_ICMP**. More specifically, if you set the *type* to be **SOCK_STREAM**, then a *protocol* of 0 automatically sets the messaging protocol to **IPPROTO_TCP**—this is the "natural" messaging protocol for a stream socket. Similarly, **IPPROTO_UDP** is the natural protocol for the **SOCK_DGRAM** type. Note that it's an error to ask for a "udp stream" or a "tcp datagram"—in other words, you can't specify **SOCK_STREAM** with **IPPROTO_UDP**, or **SOCK_DGRAM** with **IPPROTO_TCP**.

As implied by the preceding description, the most typical socket calls are:

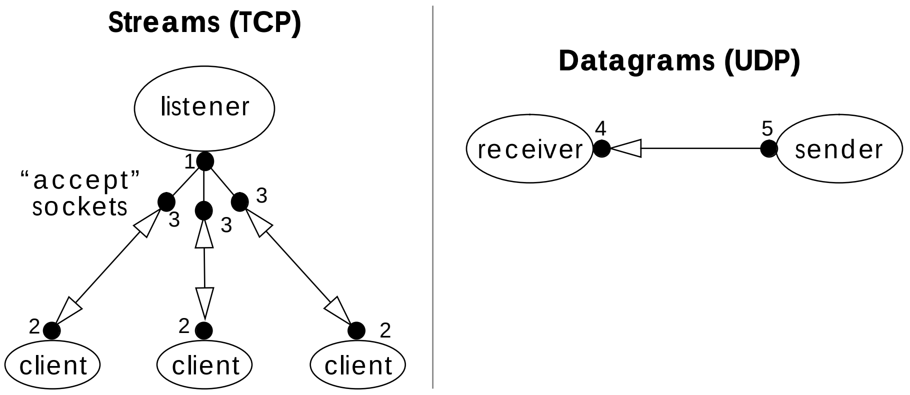
```
/* Create a stream TCP socket. */
long tcp_socket = socket(AF_INET, SOCK_STREAM, 0);

/* Create a datagram UDP socket. */
long udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
```

ICMP messages are, traditionally, sent through “raw” sockets. The Network Kit doesn’t currently support such sockets, so you should use datagram sockets instead:

```
/* Create a datagram icmp socket. */
long icmp_socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);
```

Sorts of Sockets

There are only two socket type constants: **SOCK_STREAM** and **SOCK_DGRAM**. However, if we look at the way sockets are used, we see that there are really five different categories of sockets, as illustrated below.



The labelled ovals represent individual computers that are attached to the network. The solid circles represent individual sockets. The numbers near the sockets are keys to the socket categories, which are examined in the following:

1. *The stream listener socket.* A stream listener socket provides access to a service that’s running on the “listener” machine (you might want to think of the machine as being a “server.”) The listener socket waits for client machines to “call in” and ask to be served. In order to listen for clients, the listener must call **bind()**, which “binds” the socket to an IP address and machine-specific port, and then **listen()**. Thus primed, the socket waits for a client message to show up by sitting in an **accept()** call.

2. *The stream client socket.* A stream client socket asks for service from a server machine by attempting to connect to the server’s listener socket. It does this through the **connect()** function. A stream client can be bound (you can call **bind()** on it), but it’s not mandatory.

3. *The “accept” socket.* When a stream listener hears a client in an **accept()** call, the function call creates yet another socket called the “accept” socket. Accept sockets are valid sockets, just like those you create through **socket()**. In particular, you have to remember to close accept sockets (through **closesocket()**) just as you would the sockets you explicitly create. Note that you can’t bind an accept socket—the socket is bound automatically by the system.

4. *The datagram receiver socket.* A datagram receiver socket is sort of like a stream listener: It calls **bind()** and waits for "senders" to send messages to it. Unlike the stream listener, the datagram receiver *doesn't* have to call **listen()** or **accept()**. Furthermore, when a datagram sender sends a message to the receiver, there's no ancillary socket created to handle the message (there's no UDP analog to the TCP accept socket).

5. *The datagram sender socket.* A datagram sender is the simplest type of socket—all it has to do is identify a datagram receiver and send messages to it, through the **sendto()** function. Binding a datagram sender socket is optional.

Returning to the illustration, notice that the paths connecting the stream socket clients to the stream listener (through the accept sockets) are "double arrow-headed." This indicates that TCP communication is two-way: Once the link between a client and the listener has been established (through **bind()**/**listen()**/**accept()** on the listener side, and **connect()** on the client side), the two machines can talk to each other through respective and complementary **send()** and **recv()** calls.

Communication along a UDP path, on the other hand, is one-way, as indicated by the direction of the arrow. The datagram sender can send messages (through **sendto()**), and the datagram receiver can receive them (through **recvfrom()**), but the receiver can't send message back to the sender. However, you can simulate a two-way UDP conversation by binding both sockets. This doesn't change the definition of the UDP path, or the capabilities of the two types of datagram sockets, it simply means that a bound datagram socket can act as a receiver (it can call **recvfrom()**) or as a sender (it can call **sendto()**).

Note: To be complete, it should be mentioned that datagram sockets can also invoke **connect()** and then pass messages through **send()** and **recv()**. The datagram use of these functions is a convenience; its advantages are explained in the description of the **sendto()** function.

Other Functions

bind()

int **bind(**int *socket*, struct sockaddr **interface*, int *size***)**

The **bind()** function creates an association between a socket and an "interface," where an interface is a combination of an IP address and a port number. Binding is, primarily, an in-coming message primer: When a message sender (whether it's a stream client or a datagram sender) sends a message, it tags the message with an IP address and a port number. The receiving machine—the machine with the tagged IP address—delivers the message to the socket that's bound to the tagged port.

The necessity of the bind operation, therefore, depends on the type of socket; referring to the five categories of sockets enumerated in the **socket()** function description (and illustrated in the charming picture found there), the “do I need to bind?” question is answered thus:

1. **Stream listener sockets *must* be bound**. Furthermore, after binding a listener socket, you must then call **listen()** and, when a client calls, **attach()**.

2. **Stream client sockets *can* be bound**, but they don’t have to be. If you’re going to bind a client socket, you should do so *before* you call **connect()**. The advantages of binding a stream client escape me at the moment. In any case, the client doesn’t have to bind to the same port number as the listener—the listener’s binding and the client’s binding are utterly separate entities (let alone that they are on different machines). However, the client does *connect* to the interface that the listener is bound to.

3. **Stream attach sockets *must not* be bound**.

4. **Datagram receiver sockets *must* be bound.**

5. **Datagram sender sockets don’t *have* to be bound**...but if you’re going to turn around and use the socket as a receiver, then you’ll have to bind it.

Once you’ve bound a socket, you can’t unbind it. If you no longer want the socket to be bound to its interface, the only thing you can do is close the socket (**closesocket()**) and start all over again.

Also, a particular interface can be bound to by only one socket at a time. Furthermore, in the current Be implementation of sockets, a single socket can only bind to one interface at a time. This differs with the BSD socket implementation which sets the expectation for a socket to be able to bind to more than one interface. Consider it a bug that will be fixed in a subsequent release. If you need to bind to more than one interface, you’ll need, instead, to create more than one socket and bind each one separately. An example of this is given later in this function description.

The bind() Arguments

bind()’s first argument is the socket that you’re attempting to bind. This is, typically, a socket of type **SOCK_STREAM**. The address/port combination (or “interface”) to which you’re binding the socket is passed through the *interface* argument. This is typed as a **sockaddr** structure, but, in reality, you have to create and pass a **sockaddr_in** structure cast as a **sockaddr**. The **sockaddr_in** structure is defined as:

```
struct sockaddr_in {
    unsigned short sin_family;
    unsigned short sin_port;
    struct in_addr sin_addr;
    char sin_zero[4];
};
```

- **sin_family** is the same as the address format constant that used to create the socket (the first argument to **socket()**). Currently, it’s always **AF_INET**.

- **sin_port** is the port number that the socket will bind to, given in network byte order. Valid port numbers are between 1 and 65535; numbers up to 1024 are reserved for services such as **ftp** and **telnet**. If you're not implementing a standard service, you should choose a port number above 1024. The actual value of the port number is meaningless, but keep in mind that the port number must be unique for a particular address; only one socket can be bound to a particular address/port combination.

  Note: Currently, there's no system-defined mechanism for allowing a client/sender machine to ask a listener/receiver machine for its port numbers. Therefore, when you create a networked application, you either have to hard-code the port numbers or, better yet, provide default port numbers that the user (or a system administrator) can easily change.

- **sin_addr** is an **in_addr** structure that stores, in its **s_addr** field, the IP address of the socket's machine. As always, the address is in network byte order. You can use an address of 0 to tell the binding mechanism to find an address for you. By convention, binding to address 0 (which is conveniently symbolized by the **INADDR_ANY** address) means that you want to bind to *every* address by which your computer is known, including the "loopback" (address 127.0.0.1, or the constant **INADDR_LOOPBACK**).

  On the BeBox, currently, this global-binding convention isn't implemented; instead, when you bind to **INADDR_ANY**, the **bind()** function binds to the *first* available interface (where "availability" means the address/port combination is currently unbound). Internet interfaces are considered before the loopback interface. If you want to bind to all interfaces, you have to create a separate socket for each. An example of this is given later.

- **sin_zero** is padding. To be safe, you should fill it with zeros.

The *size* argument is the size, in bytes, of the second argument.

If the **bind()** call is successful, the *interface* argument is set to contain the actual address that was used. If the socket can't be bound, the function returns less-than-zero, and sets the global **errno** to **EABDF** if the *socket* argument is invalid; for all other errors, **errno** is set to -1.

The following example shows an unexceptional use of the **bind()** function. The example uses the fictitious **gethostaddr()** function that was defined in the description of the **gethostname()** function in "Network Names, Addresses, and Services".

```
    struct sockaddr_in sa;
    int sock;
    long host_addr;

    /* Create the socket. */
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        /* error */

    /* Set the address format for the imminent bind. */
    sa.sin_family = AF_INET;

    /* We'll choose an arbitrary port number. */
    sa.sin_port = htonl(2125);

    /* Get the address of the local machine. If the address can't
     * be found (the function looks it up based on the host name),
     * then we use address INADDR_ANY.
     */
    if ((host_addr = (ulong)gethostaddr()) == -1)
        host_addr = INADDR_ANY;
    sa.sin_addr.s_addr = host_addr;

    /* Clear sin_zero. */
    memset(sa.sin_zero, 0, sizeof(sa.sin_zero));

    /* Bind the socket. */
    if (bind(sock, (struct sockaddr *)&sa, sizeof(sa)) < 0)
        /* error */
```

As mentioned earlier, the bind-to-all-interfaces convention (by asking to bind to address 0) isn't currently implemented. Thus, if the **gethostaddr()** call fails in the example, the socket will be bound to the first address by which the local computer is known.

But let's say that you really do want to bind to all interfaces. To do this, you have to create separate sockets for each interface, then call **bind()** on each one. In the example below, we create a series of sockets, and then bind each socket to an interface that specifies address 0. In doing this, we depend on the "first *available* interface" rule to find the next interface for us. Keep in mind that a successful **bind()** re-writes the contents of the **sockaddr** argument (most importantly, it resets the 0 address component). Thus, we have to re-initialize the structure each time through the loop:

```
    /* Declare an array of sockets. we’ll create as many as ten. /
    #define MAXSOCKETS
    int socks[MAXSOCKETS];
    int sockN;
    int bind_res;

    struct sockaddr_in sock_addr;

    for (sockN = 0; sockN < MAXSOCKETS; sockN++)
    {
        (socks[sockN] = socket(AF_INET, SOCK_STREAM, 0));
        if (socks[sktr] < 0) {
            perror("socket");
            goto sock_error;
        }

        /* Initialize the structure. */
        sa.sin_family =AF_INET;
        sa.sin_port = htonl(2125);
        sa.sin_addr.s_addr = 0;
        memset(sa.sin_zero,0,sizeof(sa.sin_zero));

        bind_res = bind(socks[sockN],
                        (struct sockaddr *)&sa,
                        sizeof(sa));

        /* A bind error means we've run out of addresses. */
        if (bind_res < 0) {
            closesocket(socks[sockN--]);
            break;
        }
    }

    /* Use the bound socket (listen, accept, recv/send). */
    ...

    sock_error:
        for (;sockN >=0 sockN--)
            closesocket(socks[sockN]);
```

To ask a socket about the address and port to which it is bound you use the **getsockname()** function, described elsewhere.

connect()

int **connect(**int *socket*, struct sockaddr **remote_interface*, int *remote_size***)**

The meaning of the **connect()** function depends on the type of socket that's passed as the first argument:

- If it's a stream client, then **connect()** attempts to form a connection to the socket that's specified by *remote_interface*. The remote socket must be a bound stream listener. A client socket can only be connected to one listener at a time. Note that you can't call **connect()** on a stream listener.
- If it's a datagram socket (either a sender or a receiver), **connect()** simply caches the *remote_interface* information in anticipation of subsequent **send()** and **recv()** calls. By using **connect()**, a datagram avoids the fuss of filling in the remote information that's needed by the "normal" datagram message functions, **sendto()** and **recvfrom()**. Note that a datagram may only call **send()** and **recv()** if it has first called **connect()**.

The *remote_interface* argument is a pointer to a **sockaddr_in** structure cast as a **sockaddr** pointer. The *remote_size* value gives the size of *remote_interface*. See the **bind()** function for a description of the **sockaddr_in** structure.

Currently, you can't disconnect a connected socket. If you want to connect to a different listener, or re-set a datagram's interface information, you have to close the socket and start over.

When you attempt to **connect()** a stream client, the listener must respond with an **accept()** call. Having gone through this dance, the two sockets can then pass messages to each other through complementary **send()** and **recv()** calls. If the listener doesn't respond immediately to a client's attempt to connect, the client's **connect()** call will block. If the listener doesn't respond within (about) a minute, the connection will time out. If the listener's acceptance queue is full, the client will be refused and **connect()** will return immediately.

If **connect()** fails, it returns less-than-zero, and sets **errno** to a descriptive constant:

errno Value	Meaning
EISCONN	The socket is already connected.
ECONNREFUSED	The listener rejected the connection.
ETIMEDOUT	The connection attempt timed out.
ENETUNREACH	The client can't get to the network.
EBADF	The *socket* argument is invalid.
-1	All other errors.

getsockname()

int **getsockname(**int *socket*, struct sockaddr **interface*, int *size***)**

getsockname() returns, by reference in *interface*, a **sockaddr_in** structure that contains the interface information for the bound socket given by *socket*. The **size* argument gives the size of the *interface* structure; **size* is reset, on the way out, to the size of the interface argument as it's passed back. Note that the **sockaddr_in** pointer that you pass as the second argument must be cast as a pointer to a **sockaddr** structure:

```
struct sockaddr_in interface;
int size = sizeof(interface);

/* We'll assume "sock" is a valid socket token. */
if (getsockname(sock, (struct sockaddr*)&interface, &size) < 0)
    /* error */
```

If **getsockname()** fails, the function returns less-than-zero and sets **errno** to one of the following constants:

errno Value	Meaning
EINVAL	The **size* value (going in) wasn't big enough.
EBADF	The *socket* argument is invalid.
-1	All other errors.

listen(), accept()

int **listen(**int *socket*, int *acceptance_count***)**;

int **accept(**int *socket*, struct sockaddr **client_interface*, int **client_size***)**

After you've bound a stream listener socket to an interface (through **bind()**), you then tell the socket to start "listening" for clients that are trying to connect. You then pass the socket to **accept()**; the function blocks until a client connects to the listener (the client does this by calling connect, passing a description of the interface to which the listener is bound).

When **accept()** returns, the value that it returns directly is a new socket token; this socket token represents an "accept" socket that was created as a proxy (on the local machine) for the client. To receive a message from the client, or to send a message to the client, the listener must pass the accept socket to the respective stream messaging functions, **recv()** and **send()**.

A listener only needs to invoke **listen()** once; however, it can accept more than one client at a time. Often, a listener will spawn an "accept" thread that loops over the **accept()** call.

Note that only stream listeners need to invoke **listen()** and **accept()**. None of the other socket types (enumerated in the **socket()** description) need to call these functions.

listen() Closer

int **listen(**int *socket*, int *acceptance_count***)**;

listen() takes two arguments: The first is the socket that you want to have start listening. The second is the length of the listener's "acceptance count." This is the number of clients that the listener is willing to accept at a time. If too many clients try to connect at the same time, the excess clients will be refused—the connection isn't automatically retried later.

After the listener starts listening, it must process the client connections within a certain amount of time, or the connection attempts will time out.

If **listen()** succeeds, the function returns 0; otherwise it returns less-than-zero and sets the global **errno** to a descriptive constant. Currently, the only **errno** value that **listen()** uses, other than -1, is **EBADF**, which means the socket argument is invalid.

accept() Examined

int **accept(**int *socket*, struct sockaddr **client_interface*, int **client_size***)**

The arguments to **accept()** are the socket token of the listener (*socket*), a pointer to a **sockaddr_in** structure cast as a **sockaddr** structure (*client_interface*), and a pointer to an integer that gives the size of the *client_interface* argument (*client_size*).

The *client_interface* structure returns interface information (IP address and port number) of the client that's attempting to connect. See the **bind()** function for an examination of the **sockaddr_in** structure.

The **client_size* argument is reset to give the size of *client_interface* as it's passed back by the function.

The value that **accept()** returns directly is a token that represents the accept socket. After checking the token value (where less-than-zero indicates an error), you must cache the token so you can use it in subsequent **send()** and **recv()** calls.

When you're done talking to the client, remember to call **closesocket()** on the accept socket that **accept()** returned. This frees a slot in the listener's acceptance queue, allowing a possibly frustrated client to connect to the listener.

If **accept()** fails, it returns less-than-zero (as mentioned above) and sets **errno** to one of the following constants:

errno Value	Meaning
EINVAL	The listener socket isn't bound.
EWOULDBLOCK	The acceptance queue is full.
EBADF	The *socket* argument is invalid.
-1	All other errors.

select()

int **select(**int *socket_range*,
struct fd_set **read_bits*,
struct fd_set **write_bits*,
struct fd_set **exception_bits*,
struct timeval **timeout***)**

The **select()** function returns information about selected sockets. The *socket_range* argument tells the function how many sockets to check: It only checks the first (*socket*_range - 1) sockets. You don't have to be exact with this value; typically, you set the argument to 32. Note that a *socket_range* value of 0 *doesn't* select the first socket (which will have a token of 0). You have to pass a value of at least 1.

The **fd_set** structure that types the next three arguments is simply a 32-bit mask that encodes the sockets that you're interested in; this refines the range of sockets that was specified in the first argument. You should use the **FD_***OP***()** macros to manipulate the structures that you pass in:

- **FD_ZERO(***set***)** clears the mask given by *set*.
- **FD_SET(***socket*, *set***)** adds a socket to the mask.
- **FD_CLEAR(***socket*, *set***)** clears a socket from the mask.
- **FD_ISSET(***socket*, *set***)** returns non-zero if the given socket is already in the mask.

The function passes socket information back to you by resetting the three **fd_set** arguments. The arguments themselves represent the types of information that you can check:

- *read_bits* tells you if a socket is "ready to read." In other words, it tells you if a socket has a in-coming message waiting to be read.
- *write_bits* tells you if a socket is "ready to write."
- *exception_bits* tells you if there's an exception pending on the socket.

Note: Currently, only *read_bits* is implemented. You should pass **NULL** as the *write_bits* and *exception_bits* arguments.

select() doesn't return until at least one of the **fd_set**-specified sockets is ready for one of the requested operations. To avoid blocking forever, you can provide a time limit in the final argument, passed as a **timeval** structure.

In the following example function implementation, we check if a given datagram socket has a message waiting to be read. The **select()** times out after two seconds:

```
bool can_read_datagram(int socket)
{
    struct timeval tv;
    struct fd_set fds;
    int n;

    tv.tv_sec = 2;
    tv.tv_usec = 0;

    /* Initialize (clear) the socket mask. */
    FD_ZERO(&fds);

    /* Set the socket in the mask. */
    FD_SET(socket, &fds);
    select(s + 1, &fds, NULL, NULL, &tv);

    /* If the socket is still set, then it's ready to read. */
    return FD_ISSET(socket, &fds);
}
```

If **select()** experiences an error, it returns -1; if the function times out, it returns 0. Otherwise—explicitly, if *any* of the selected sockets was found to be ready—it returns 1.

send(), recv()

int **send(**int *socket*, const char **buf*, int *size*, int *flags***)**
int **recv(**int *socket*, char **buf*, int *size*, int *flags***)**

These functions are used to send data to a remote socket, and to receive data that was sent by a remote socket. **send()** and **recv()** calls must be complementary: After socket A sends to socket B, socket B needs to call **recv()** to pick up the data that A sent. **send()** sends its data and returns immediately. **recv()** will block until it has some data to return.

The **send()** and **recv()** functions can be called by stream or datagram sockets. However, there are some differences between the way the functions work when used by these two types of socket:

- For a stream listener and a stream client to transmit messages, the listener must have previously called **bind()**, **listen()**, **accept()**, and the client must have called **connect()**. Having been properly connected, the two sockets can send and receive as if they were peers.

  For stream sockets, **send()** and **recv()** can both block: **send()** blocks if the amount of data that's sent overwhelms the receiver's ability to read it, and **recv()** blocks if there's no message waiting to be read. You can tell a **recv()** to be non-blocking by setting the sending socket's no-block socket option (see **setsockopt()**). The no-block option doesn't apply to sending.

- If you want to call **send()** or **recv()** through a datagram socket, you must first **connect()** the socket. In addition, a receiving datagram socket must also be bound to an interface (through **bind()**). See the **connect()** description for more information on what that function means to a datagram socket.

  Datagram sockets never block on **send()**, but they can block in a **recv()** call. As with stream sockets, you can set a datagram socket to be non-blocking (for the **recv()**, as well as for **recvfrom()**) through **setsockopt()**.

The Arguments

The arguments to **send()** and **recv()** are:

- *socket* is, for datagrams and stream client sockets, the local socket token. In other words, when a datagram or stream client wants to send or receive data, it passes its own socket token as the first argument. The recipient of a **send()**, or the sender of a **recv()** is, for these sockets, well-known: Its the socket that's identified by the previous **connect()** call.

  For a stream listener, *socket* is the "accept socket" that was previously returned by an **accept()** call. A stream listener can send and receive data from more than one client at the same time (or, at least, in rapid succession).

- *buf* is a pointer to the data that's being sent, or is used to hold a copy of the data that was received.

- *size* is the allocated size of *buf*, in bytes.

- *flags* is currently unused. For now, set it to 0.

A successful **send()** returns the number of bytes that were send; a successful **recv()** returns the number of bytes that were received. If a **send()** or **recv()** fails, it returns less-than-zero and sets **errno** to a descriptive constant:

errno Value	Meaning
EWOULDBLOCK	The call would block on a non-blocking socket (**recv()** only).
EINTR	The local socket was interrupted.
ECONNRESET	The remote socket disappeared (**send()** only).
ENOTCONN	The socket isn't connected.
EBADF	The *socket* argument is invalid.
EADDRINUSE	The interface specified in the previous connect is busy (datagram sockets only).
-1	All other errors.

sendto(), recvfrom()

int **sendto(**int socket,
char **buf*,
int *size*,
int *flags*,
struct sockaddr **to*,
int *tolen***)**

int **recvfrom(**int socket,
char **buf*,
int *size*,
int *flags*,
struct sockaddr **from,*
int **fromlen***)**

These functions are used by datagram sockets (only) to send and receive messages. The functions encode all the information that's needed to find the recipient or the sender of the desired message, so you don't need to call **connect()** before invoking these functions. However, a datagram socket that wants to receive message must first call **bind()** (in order to fix itself to an interface that can be specified in a remote socket's **sendto()** call).

The four initial arguments to these function are similar to those for **send()** and **recv()**; the additional arguments are the interface specifications:

- For **sendto()**, the *to* argument is a **sockaddr_in** structure pointer (cast as a pointer to a **sockaddr** structure) that specifies the interface of the remote socket that you're sending to. The *tolen* argument is the size of the *to* argument.

- For **recvfrom()**, the *from* argument returns the interface for the remote socket that sent the message that **recvfrom()** received. **fromlen* is set to the size of the *from* structure. As always, the interface structure is a **sockaddr_in** cast as a pointer to a **sockaddr**.

sendto() never blocks. **recvfrom()**, on the other hand, will block until a message arrives, unless you set the socket to be non-blocking through the **setsockopt()** function.

You can "broadcast" a message to all interfaces that can be found by setting **sendto()**'s target address to **INADDR_BROADCAST**.

As an alternative to these functions, you can call **connect()** on a datagram socket and then call **send()** and **recv()**. The **connect()** call caches the interface information provided in its arguments, and uses this information the subsequent **send()** and **recv()** calls to "fake" the analogous **sendto()** and **recvfrom()** invocations. For sending, the implication is obvious: The target of the send() is the interface supplied in the **connect()**. The implication for receiving bears description: When you **connect()** and then call **recv()** on a datagram socket, the socket will only accept messages from the interface given in the **connect()** call.

You can mix **sendto()**/**recvfrom()** calls with **send()**/**recv()**. In other words, connecting a datagram socket doesn't prevent you from calling **sendto()** and **recvfrom()**.

A successful **sendto()** returns the number of bytes that were send; a successful **recvfrom()** returns the number of bytes that were received. If a **sendto()** or **recvfrom()** calls fails, less-than-zero is returned and errno is set to a descriptive constant:

errno Value	Meaning
EWOULDBLOCK	The call would block on a non-blocking socket (**recvfrom()** only).
EINTR	The local socket was interrupted.
EBADF	The *socket* argument is invalid.
EADDRNOTAVAIL	The specified interface is unrecognized.
-1	All other errors.

setsockopt()

int **setsockopt(**int *socket*, int *level*, int *option*, char **data*, unsigned int *size***)**

setsockopt() lets you set certain "options" that are associated with a socket. Currently, the Network Kit only recognizes one option: It lets you declare a socket to be blocking or non-blocking. A blocking socket will block in a **recv()** or **recvfrom()** call if there's no data to retrieve. A non-blocking socket returns immediately, even if it comes back empty-handed.

Note that a socket's blocking state applies *only* to **recv()** and **recvfrom()** calls.

The function's arguments are:

- *socket* is the socket that you're attempting to affect.
- *level* is a constant that indicates where the option is enforced. Currently, *level* should always be **SOL_SOCKET**.
- *option* is a constant that represents the option you're interested in. The only option constant that does anything right now is **SO_NONBLOCK**. (Two other constants—**SO_REUSEADDR** and **SO_DEBUG**—are recognized, but they aren't currently implement.)
- *data* points to a buffer that's used to toggle or otherwise inform the option. For the **SO_NONBLOCK** option (and other boolean options), you fill the buffer with zeroes if you want to turn the option off (the socket will block), and non-zeros if you want to turn it on (the socket won't block). In the case of a boolean option, a single byte of zero/non-zero will do.
- *size* is the size of the *data* buffer.

The function returns 0 if successful; otherwise, it returns less-than-zero and sets errno to a descriptive constant:

errno Value	Meaning
ENOPROTOOPT	Unrecognized *level* or *option* argument.
EBADF	The *socket* argument is invalid.
-1	All other errors.

Keep in mind that attempting to set the **SO_REUSEADDR** or **SO_DEBUG** option won't generate an error, but neither will it do anything.

The Mail Daemon

Declared in: <net/E-Mail.h>

Overview

Every Be machine has a *mail daemon*; this is a local process that's responsible for retrieving mail from and sending mail to a mail server. The mail server that the daemon talks to is a networking application that's either part of your Internet Service Provider's services, or that's running on a local "mail repository" machine. The functions described in this section tell you how to manage the mail daemon's connection with the mail server—how to tell the daemon which mail server to talk to, how to command the daemon to send and retrieve mail, how to automate mail retrieval, and so on.

All the functions that are described here (but one) are promoted to user-land through the E-mail preferences application (the one exception is the **forward_mail()** function). Indeed, the operations that these functions perform are rightly regarded as belonging to the user. The only reason that you would need to call the daemon functions—with the exceptions of **forward_mail()** and, possibly, **check_for_mail()**—is if you want to build your own E-mail preferences application. (**forward_mail()** and **check_for_mail()** could legitimately be worked into a mail-reading or -composing application.)

The architecture of the E-mail message itself isn't discussed here; for such information see "Mail Messages (BMailMessage)" on page 37.

The Mail Daemon and the Mail Server

The mail daemon can talk to two different mail servers:

- The *Post Office Protocol* ("POP") server manages individual mail accounts. When the Be mail daemon wants to retrieve mail that's been sent to a user, it must tell the mail server which POP account it's retrieving mail for.
- The *Simple Mail Transfer Protocol* ("SMTP") server manages mail that's being sent out into the world (and that will, eventually, find its way to a POP server).

The POP and the SMTP servers are identified by their hosts' names (in other words, the names of the machines on which the servers are running). The mail daemon can only talk to one POP and one SMTP server at a time, but can talk to the two of them simultaneously. Typically—nearly exclusively—the POP and SMTP servers reside on the same machine, and so are identified by the same name.

To set the identities of the POP and SMTP mail servers, you fill in the fields of a **mail_account** structure and pass the structure to the **set_mail_account()** function. As the name of the structure implies, **mail_account** encodes more than just the names of the servers' hosts. It also identifies a specific user's POP mail account; the complete definition of the structure is this:

```
typedef struct
{
    char pop_name[B_MAX_USER_NAME_LENGTH];
    char pop_password[B_MAX_USER_NAME_LENGTH];
    char pop_host[B_MAX_HOST_NAME_LENGTH];
    char smtp_host[B_MAX_HOST_NAME_LENGTH];
} mail_account;
```

The POP user information that's stored in the **mail_account** structure (in other words, the **pop_name** and **pop_password** fields) is used only for the POP server; it has no significance for the SMTP server.

Sending and Retrieving Mail

Messages that are retrieved (from the mail server) by the mail daemon are stored in the database, from whence they are plucked and displayed by a mail-reading application (a "mail reader"; Be supplies a simple mail reader called BeMail). Similarly, messages that the user composes (in a mail composition application) and sends are placed in the database until the mail daemon comes along and passes them on to the mail server.

Sending and retrieving mail is the mail daemon's most important function. Both actions (server-to-database and database-to-server transmission) are performed through the **check_for_mail()** function. This is the mail daemon's fundamental "do something" function. All other function either prime the daemon

Other Mail Daemon Features

The other mail structures and functions define the other features that are provided by the mail daemon. These features are:

- *A mail delivery schedule.* The **mail_schedule** structure (passed through the **set_mail_schedule()** function) lets you tell the daemon how often and during which periods (week days only, every day, and so on) it should automatically check for newly arrived mail and send newly composed mail. Technically, the mail schedule tells the daemon how often to invoke **check_for_mail()**.

- *Mail notification.* The **mail_notification** structure (passed through the **set_mail_notification()** function) lets you tell the daemon how you would like it to tap you on the shoulder when it has new mail for you to read. Would you like it to display an alert panel? Squawk at you? Both?

- *A settable mail reader.* The **set_mail_reader()** function lets you identify the application that you would like to use to read in-coming mail. (Be provides a default mail reader/composition program called BeMail.)
- *Mail forwarding*. The **forward_mail()** function lets you re-send in-coming mail to some other account.

All of these features (less mail-forwarding) can also be set by the user through the E-mail preferences panel.

Functions

check_for_mail()

long **check_for_mail(**long **incoming_count***)**

Sends and retrieves mail. More specifically, this functions asks the mail daemon to retrieve in-coming messages from the POP server and send out-going messages to the SMTP server. The number of POP messages that were retrieved is returned, by reference, in the argument. If you don't need to know the in-coming count, you can (and should) pass **NULL** as the *incoming_count* argument; the function is (potentially) much faster if you ignore the count in this manner.

If the mail world is unruffled, the function returns **B_NO_ERROR**; otherwise, it returns one of the following:

- **B_MAIL_NO_DEMON**. The mail demon isn't running.
- **B_MAIL_UNKNOWN_HOST**. The named POP or SMTP mail server can't be found.
- **B_MAIL_ACCESS_ERROR**. The connection to the POP or SMTP mail server failed.
- **B_MAIL_UNKNOWN_USER**. The POP server doesn't recognize the user name.
- **B_MAIL_WRONG_PASSWORD**. The POP server doesn't recognize the password.

In the cases where a name or password is unrecognized (**B_MAIL_UNKNOWN_HOST**, **...UNKNOWN_USER**, and **...WRONG_PASSWORD**), the (mis)information is taken from the **mail_account** structure that was passed to the daemon in the most recent **set_mail_account()** call. Note that the validity of the **mail_account** information isn't checked when you set the structure—it's only checked when you actually attempt to use the information (as, for example, here).

forward_mail()

long **forward_mail(**BRecord **msg*,
char **recipients*,
bool *reset_sender* = TRUE,
bool *queue* = TRUE**)**

Forwards the mail message represented by *msg* to the list of users given by *recipients*. *msg* is a BRecord object that encapsulates a single in-coming mail message. The user account names listed in *recipients* must be separated from each other by whitespace and/or commas; the entire list must be **NULL**-terminated. Both of these entities (the BRecord-as-mail-message, and the recipients list) are further explained in "Mail Messages (BMailMessage)" on page 37.

If *reset_sender* is **TRUE**, the sender of the forwarded message is reset to be the current recipient; otherwise the sender is left as is. For example, if Anton sends a message to Bertrand and Bertrand forwards the message to Camille with *reset_sender* set to **TRUE**, the message that Camille receives will appear to have been sent by Bertrand; if set to **FALSE**, it will appear to have been sent by Anton.

The *queue* argument determines whether the messages is sent now (**TRUE**) or queued for later transmission (**FALSE**). If you send the message now, all other out-going and in-coming mail messages are transmitted as a matter of course (sending now is like calling **check_for_mail()**). If the message is queued, it waits for the daemon to perform its automatic check, or for the next explicit **check_for_mail()** call.

set_mail_account(), get_mail_account()

long **set_mail_account(**mail_account **account*, bool *save* = TRUE**)**
long **get_mail_account(**mail_account **account***)**

set_mail_account() function lets you set the identities of the POP and SMTP mail servers that you want the mail daemon to use, and lets you set the (user-specific) POP account that the daemon should monitor (when it looks for in-coming mail). All this information is set by filling in the fields of the **mail_account** structure which you pass as the first argument to the function. The structure is defined as

```
typedef struct
{
    char pop_name[B_MAX_USER_NAME_LENGTH];
    char pop_password[B_MAX_USER_NAME_LENGTH];
    char pop_host[B_MAX_HOST_NAME_LENGTH];
    char smtp_host[B_MAX_HOST_NAME_LENGTH];
} mail_account;
```

- **pop_name** and **pop_password** are **NULL**-terminated strings (with a maximum length of 32 characters) that identify the user account on the POP server. The account must already exist; you can't create a new POP account simply by filling a **mail_account** structure and passing it through **set_mail_account()**. Creating a POP account is the responsibility of the Internet Service Provider.

- **pop_host** is a **NULL**-terminated string (64 characters, max) that names the machine on which resides the POP server, and **smtp_host** is a similarly constructed string that names the SMTP server's machine. Normally, the servers are run on the same machine. Again, you can't make up a name here; you have to get the host names from the Internet Service Provider.

The *save* argument sets the persistence of the mail account:

- If you save, this account will be used for all subsequent transactions with the mail servers, and also becomes the *default mail account*. In this role, the account information is remembered when you restart your computer (or otherwise kill and restart the mail daemon).
- If you don't save, this account will be used for subsequent transactions, but will be forgotten when you shut down.

You can set the default mail account even if the mail daemon isn't running. Currently, the **set_mail_account()** function always returns **B_NO_ERROR**.

get_mail_account() returns, by reference in its argument, a copy of the mail account information that the daemon is currently set to use. If the daemon isn't running, this function returns the default mail account. In this case, the function returns **B_MAIL_NO_DAEMON**, otherwise it returns **B_NO_ERROR**.

Note that the validity of the **mail_account** that you pass to **set_mail_account()** or that's copied into the **get_mail_account()** argument isn't checked by these functions. The mail account is only checked when you actually attempt to use the information; in other words, when you attempt to send or retrieve mail.

set_mail_notification(), get_mail_notification()

long **set_mail_notification(**mail_notification **notification*, bool *save* = TRUE**)**
long **get_mail_notification(**mail_notification **notification***)**

set_mail_notification() establishes how you would like to be notified when new mail arrives. There are two notification signals: the mail alert panel and the system beep. You encode your preference by setting the fields of the argument **mail_notification** structure:

```
typedef struct
{
    bool alert;
    bool beep;
} mail_notification;
```

The *save* argument, if **TRUE**, registers the notification setting as the default—in other words, the daemon will remember it when you shutdown the computer. This function always returns **B_NO_ERROR.**

get_mail_notification() returns, by reference, a copy of the **mail_notification** structure that's currently being used by the mail daemon. If the daemon isn't running, the function hands you the default notification setting, and returns (directly) **B_MAIL_NO_DAEMON**; otherwise it returns **B_NO_ERROR**.

set_mail_reader(), get_mail_reader()

long **set_mail_reader(**ulong *reader_sig*, bool *save* = TRUE**)**
long **get_mail_reader(**ulong **reader_sig***)**

set_mail_reader() tells the system which application to launch (or find) to display newly-arrived mail. The application is identified by its signature. The *save* argument, if **TRUE**, registers the reader signature as the default—in other words, the daemon will remember it when you shutdown the computer. This function always returns **B_NO_ERROR**; note that the function doesn't check to make sure that the argument identifies an actual application.

get_mail_reader() returns, by reference, the signature of the application that the mail daemon is currently using (or will next use) to display mail. If the daemon isn't running, the function hands you the default reader, and returns (directly) **B_MAIL_NO_DAEMON**; otherwise it returns **B_NO_ERROR**.

In the absence of any other provision, the mail daemon uses the Be mail reader, BeMail (signature 'MAIL').

set_mail_schedule(), get_mail_schedule()

long **set_mail_schedule(**mail_schedule **schedule*, bool *save* = TRUE**)**
long **get_mail_schedule(**mail_schedule **schedule***)**

set_mail_schedule() lets you tell the mail daemon during what days and hours it should automatically check for new mail, and how often it should check. You encode this information by filling in the fields of the argument **mail_schedule** structure:

```
typedef struct
{
    long  days;
    long  interval;
    long  start_time;
    long  end_time;
} mail_schedule;
```

- **days** is a constant that encodes the range of days. It can be one of **B_CHECK_DAILY**, **B_CHECK_WEEKDAYS**, or **B_CHECK_NEVER**. The first two should be obvious; setting the **days** field to **B_CHECK_NEVER** turns off the daemon's automatic mail-checking capability (and the other fields of the structure are ignored).

- **start_time** and **end_time** define the range of minutes, within the candidate days, that the daemon checks for mail. For example, if you want the daemon to check for mail only between 8 am and 6 pm, you would set **start_time** to 480 (8 hours * 60 minutes) and **end_time** to 1080 (18 hours * 60 minutes). If **start_time** and **end_time** are the same, then the daemon works around the clock.

- **interval** is the frequency, in minutes, at which the mail daemon checks for mail. For example, setting interval to 15 means that the daemon will automatically check for new mail (and send out any unsent, recently composed messages) every 15 minutes (within the range of minutes of the candidate days, as set in the other fields).

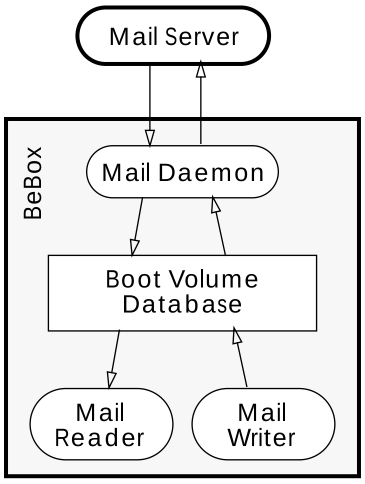
Mail Messages (BMailMessage)

Derived from:	public BObject
Declared in:	<net/E-mail.h>

Overview

When the mail daemon retrieves new mail from the mail server, it stores the retrieved messages in the boot volume's database, creating a single record (a "mail record") for each message. A mail-reading program can then pull the mail record out of the database (as a BRecord object) and display its contents.

Similarly, when the user composes new mail (on the BeBox) and submits the message for sending, the message-composing application adds the message (again, encapsulated in a mail record) to the database where it waits for the daemon to pick it up and send it to the mail server. The scheme looks something like this:



If you're writing a mail-reading or mail-writing application, then all you really need to know is the definition of the table to which the mail message records conform. With this knowledge, you can retrieve ("fetch") and parse in-coming messages, and create and submit (to the database) out-going messages. The mail message table is called "E-mail", and is described in "The E-Mail Table" on page 43.

The Network Kit also supplies a BMailMessage class that acts as a convenient wrapper around mail records.

The following sections give you a "mail message" tutorial; we'll step through the database and mail message operations that you need to create a generic mail application. If you're already comfortable with database programming (and understand SMTP and POP), you can skip the tutorial and head straight for the "E-mail" and BMailMessage specifications.

Creating a Mail Reader

The design of a Be mail reader should follows this outline:

1. Ask the mail daemon to retrieve mail from the mail server.
2. Get the newly retrieved mail messages from the database.
3. Display the contents of the mail messages.

Throughout the following step-by-step explanations, we'll give both the general approach and also look at what BeMail does.

(Note that this tutorial is incomplete.)

Asking the Daemon to Get New Mail

There are a couple of ways to ask the mail daemon to retrieve newly arrived mail from the mail server:

- You can ask it explicitly by calling **check_for_mail()**
- You can wait for the mail schedule's automatic invocation of **check_for_mail()**.

You probably want to do both of these: You should provide a means for the user to ask that mail be retrieved right now, while also allowing the schedule to do its thing. **check_for_mail()** and **set_mail_schedule()**, which declares the periodicity of automatic mail retrieval, are described in "The Mail Daemon" on page 29. As explained there, the mail schedule "belongs" to the user; its default presentation is through the E-Mail preferences application.

BeMail doesn't actually do anything about retrieving mail. It relies on the mail schedule, and on the mail daemon's "E-Mail Status" alert panel, which provides a "Check Now" button (as in "check for mail now").

Getting Messages from the Database

When new mail arrives, the mail daemon creates a database record to hold each new message, and then commits the records to the database. The table that a mail record conforms to is named "E-Mail". This table is kept in the database that corresponds to the boot volume. As a demonstration of these principles, the following example function counts the number of mail messages that currently reside in the database:

```
#include <Database.h>
#include <Table.h>
#include <Volume.h>
#include <Query.h>

long count_all_email()
{
    BVolume bootVol = boot_volume();
    BDatabase *bootDb = boolVol.Database();
    BTable *emailTable = bootDb->FindTable("E-Mail");
    BQuery *emailQuery = new BQuery();
    long result=0;

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);
        emailQuery->PushOp(B_ALL);
        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();
    }
    delete emailQuery;
    return result;
}
```

Obviously, this example requires some knowledge of how the database works. You can mosey on over to the Storage Kit documentation for the Tolstoy version, or you can read between the lines of the following:

As mentioned above, the mail daemon transforms mail messages into “E-Mail” conforming records, and then “commits” (in database lingo) these records to the boot volume’s database. The first few lines of the example assemble the suspects: The boot volume, the database from the boot volume, and the “E-Mail” table from the boot database. If the table is found, then we construct a “query”—this is the vehicle that will let us retrieve our records. The query is told which table to look in and which records in that table to look for. This is done through **AddTable(**emailTable**)** and the **PushOp(**B_ALL**)** calls; in other words, we tell the query to look for all records in the “E-Mail” table. Then we tell the query to “fetch,” or go out and actually get the records. Technically, it doesn’t actually get records (this would be inefficient); instead, it gets record ID numbers. We count the record ID numbers that it has retrieved (the **CountRecordIDs()** call) and return the count.

The Example Refined—E-Mail Status

For the purposes of a mail reader—in other words, an application that wants to display messages that are received from the mail server—retrieving *all* mail messages isn’t quite right. The “E-Mail” table is used to store both in-coming and out-going messages. So we have to fix our query to only count in-coming messages.

The in-coming/out-going nature of a particular message is stored as a string in the “Status” field of the “E-Mail” table. The mail daemon understands three states: “New”, “Pending”, and “Sent” (a fourth state, “Read”, is used by the BeMail application; we’ll get to it later). We're only interested in "New" messages, so we change our query accordingly:

```
long count_incoming_email()
{
    /* declarations as above */

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);

        emailQuery->PushField("Status");
        emailQuery->PushString("New");
        emailQuery->PushOp(B_EQ);

        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();
    }
    delete emailQuery;
    return result;
}
```

Here we've replaced the **PushOp(**B_ALL**)** call with a more refined predicate. Again, you can turn to the Storage Kit (the BQuery class, specifically) for the full story on query predicates. Briefly, predicates are expressed in RPN ("Reverse Polish Notation"). According to RPN, the operands of an operation are "pushed" first, followed by the operator. The evaluation of an operation becomes a valid operand for another operation. The series of "pushes" in the example expresses the boolean evaluation

```
(status == "New")
```

In other words, we're going to fetch all records (again, record IDs) that have a "Status" field value of "New".

Let the Browser do the Work

There's one other way to identify mail records: Let the Browser do it. When you "launch" the Browser-defined Mailbox, a query that looks a lot like the one we created above is formed and fetched. The result of the query, the list of found record ID numbers, is turned into a list of BRecord objects that are symbolically listed in the Mailbox window. If the user double-clicks on one of the record icons, the mail daemon passes the record's **record_ref** to the user-defined "mail reader." By default, the mail reader is BeMail. The user can select a different reader (yours) by dropping the reader's icon in the appropriate "icon well" in the E-Mail preferences panel. You can set the reader identity programmatically through the **set_mail_reader()** function, although, as with all E-Mail preferences, it's nicer to let the user make the decision.

A mail reader application needs to be able "catch" the refs that are passed to it. It does this in its implementation of **MessageReceived()**. A simple implementation would look for the message type **B_REFS_RECEIVED**. The rest of this thought is left as an exercise for the mind reader.

Creating BMailMessage Objects

So far, we've determined where we have to go to find in-coming messages, and counted the messages that we found there, but we haven't actually retrieved the messages themselves. Here, we complete our message-retrieving example by fetching record IDs (as before) and then constructing a BRecord for each ID. Having done that, we pass the BRecord to the BMailMessage constructor. In the example, we'll add each BMailMessage to a BList (which is passed in to the function):

```
#include <E-mail.h>
/* and the others */

long get_new_email(BList *list)
{
    BRecord *emailRecord;
    BMailMessage *newMail;
    long count;
    record_id rec_id;

    /* and the others */

    if (emailTable != NULL) {
        emailQuery->AddTable(emailTable);

        emailQuery->PushField("Status");
        emailQuery->PushString("New");
        emailQuery->PushOp(B_EQ);

        emailQuery->Fetch();
        result = emailQuery->CountRecordIDs();

        for (count=0; count < result; count++) {
            rec_id = emailQuery->RecordIdAt(count);
            emailRecord = new BRecord(bootDb, rec_id);
            newMail = new BMailMessage(emailRecord);
            list->AddItem(newMail);
            delete emailRecord;
        }
    delete EmailQuery;
    return result;
}
```

In the **for** loop, we step through the query's "record ID" list, creating a BRecord for each ID. To construct a BRecord from a record ID, you need to pass the appropriate BDatabase object; this is because record ID numbers are only valid within a specific database. Having gotten a BRecord, we pass the object to the BMailMessage constructor; the BMailMessage object copies all the data from the record into itself, such that the BRecord is no longer needed. The BRecord object can (and, unless you have something up your sleeve, should) be deleted after the BMailMessage object is constructed.

When our example function returns, the argument BList will contain all the BMailMessage objects that we constructed, and the function will return the number of messages directly (as before). Note that we should be a bit pickier about checking for errors; you will, no doubt, correct this oversight in your own mail reader. Also—and here we're just being fussy—keep in mind that by adding the BMailMessages to a BList, we have implied that the BList is now responsible for these objects. More precisely, the entity that called **get_new_mail()** must delete the contents of the list when it's done doing whatever it does.

Displaying the Contents of a Message

The BMailMessage class provides a convenient object cover for mail records. By using BMailMessage objects, you avoid most of the fuss of parsing database records.

When you construct a BMailMessage to represent an in-coming (or otherwise existing) mail message, the contents of the message are copied into the object's "fields." BMailMessage fields are similar to table fields in that they represent named categories of data. The fields that are defined by the BMailMessage class approximate those of the "E-Mail" table; however, you can add new fields that have no complement in the table—adding a field to a BMailMessage object won't extend the "E-Mail" table definition.

The **FindField()** member function retrieves the data that's stored for a particular field within a BMailMessage object. The full protocol goes something like this:

long **FindField(**const char **field_name*, void ***data*, long **length*, long *index*=0**)**

The function works as you would expect: You pass in a field name, and the function points **data* at the contents of that field. The length of the data (in bytes) is returned in *length*. The final argument (*index*) is used to disambiguate between fields that have the same name (the exact value of *index* has no meaning other than ordinal position).

To use **FindField()** properly, you have to know the names of the fields that you can expect to find there. The BMailMessage class defines a number of field names and provides constants to cover them:

Field Name	Constant
"To: "	**B_MAIL_TO**
"Cc: "	**B_MAIL_CC**
"Bcc: "	**B_MAIL_BCC**
"From: "	**B_MAIL_FROM**
"Date: "	**B_MAIL_DATE**
"Reply: "	**B_MAIL_REPLY**
"Subject: "	**B_MAIL_SUBJECT**
"Priority: "	**B_MAIL_PRIORITY**
"Content"	**B_MAIL_CONTENT**

Each of the defined fields stores some number of bytes of "raw" (untyped) data. When you call **FindField()**, the function points the second argument (*data*) to the raw data for the named field, and returns the number of bytes of data in the third argument (*length*).

A particular field (i.e. a field with a particular name) can store more than one entry. The final argument to **FindField()** (the argument named *index*) can be used to distinguish between multiple entries in the same field.

(Here the master died. We'll complete this tutorial and post it on the Be Web site very soon.)

The E-Mail Table

The "E-Mail" database table defines records that hold mail messages. The fields in the table mimic the information that's found in an SMTP or POP mail message header. (See "The Mail Daemon" on page 29 for more information on SMTP and POP). The table's fields are:

- "Status" takes a string that describes the "seen it" state of the message. The mail daemon sets newly arrived in-coming messages to be "New". Out-going messages must have a status of "Pending" (this cues the daemon to send the message). After it has sent a message, the daemon sets the status to "Sent". Beyond these three states, an application is free to invent and use its own—for example, BeMail uses "Read" to mean a message that used to be "New", but which the user has already looked at.
- "Priority" is an integer (a **long**) that rates the message's urgency.
- "From" is a string that names the sender of the message.
- "Subject" is a string that describes the topic of the letter.
- "Reply" is a string that gives the e-mail name to which a response to this message should be sent.
- "When" is a **double** that encodes the date and time at which this message was sent.
- "Enclosures" is an integer count of the number of MIME enclosures that the message contains.
- "header" is the unaltered POP header from a received message; if you're creating mail records yourself (as opposed to using the BMailMessage class), you should construct an SMTP header and add it to this field.
- "content" as a string is the unaltered content of the message.
- "content_file" as a record ID is used if the size of the content threatens to broach the maximum size of a record. In this case, the content is written to a file, and "content_file" gives the ID of that file.
- "enclosures" is a list of attributes (the field itself is typed as raw data) that describe the individual MIME enclosures. There are three attributes per enclosure: a **record_ref** that gives the location of the enclosure, stored as a file; a **NULL-**

terminated string that gives the MIME type, and a **NULL**-terminated MIME subtype string.

- "mail_flags" is a long that encodes the message-is-pending and save-after-sending states of the message. If the message is waiting to go out, the "mail_flags" value is **B_MAIL_PENDING**; if it should be saved after it's sent, then **B_MAIL_SAVE** is added in. After the message is sent, the record is destroyed if "mail_flags" doesn't include **B_MAIL_SAVE**, otherwise the "mail_flags" valued is set to **B_MAIL_SENT**. In all other cases—if the message is in-coming, for example—"mail_flags" is 0.

Constructor and Destructor

BMailMessage()

BMailMessage(void**)**
BMailMessage(BRecord **record***)**
BMailMessage(BMailMessage **mail_message***)**

Creates and returns a new BMailMessage object.

The first version creates an empty, "abstract" message: The object doesn't correspond to the second creates an object that acts as a cover for the given BRecord, and the third creates a copy (more or less) of its argument.

~BMailMessage()

virtual **~BMailMessage(**void**)**

Destroys the BMailMessage, even if the object's fields are "dirty." For example, let's say you create a new BMailMessage with the intention of sending a message. You start to edit the object—perhaps you fill in the "To: " field—but then you delete the object. The message that you were composing isn't sent. In other words, the BMailMessage object doesn't try to second-guess your intentions: When you destroy the object, it lies down and dies without whining about it.

Member Functions

CountFields(), GetFieldName(), FindField()

long **CountFields(**char **name* = NULL**)**

long **GetFieldName(**char ***field_name*, long *index***)**

long **FindField(**char **field_name*,
void ***data*,
long **length*,
long *index* = 0**)**

These functions are used to step through and inspect the fields in a BMailMessage object. A field is identified, primarily, by its name. However, a field can have more than one entry, so a secondary identifier (an index) is also necessary. Through the combination of a field name and an index, you can identify and retrieve a specific piece of data. The names of the "standard" mail fields are listed in the **SetField()** description.

CountFields() returns the entry count for the named field. If the name argument is **NULL**, the function returns the number of uniquely named fields in the object. Note that the **NULL** argument version doesn't necessarily return a count of *all* fields. For example, if a BMailMessage contains two **B_MAIL_TO** fields (only), the call

```
    CountFields(B_MAIL_TO);
```

will return 2, while the call

```
    CountFields();
```

will return 1.

GetFieldName() returns, by reference in the *field_name* argument, the name of the field that occupies the *index*'th place in the object's list of uniquely named fields. If *index* is out-of-bounds, the function returns (directly) **B_BAD_INDEX**; otherwise, it returns **B_NO_ERROR**.

FindField() return the data that lies in the field that's identified by *field_name*. If the object contains more than one entry, you can use the index argument to differentiate them. The data that's found is returned by reference through **data*; the **length* value returns the amount of data (in bytes) that **data* is pointing to. It's not a great idea to alter the pointed-to data, but as long as you don't exceed the existing length you'll probably get away with it.

If *field_name* doesn't identify an existing field (in this object), **B_MAIL_UNKNOWN_FIELD** is returned; if the index is out-of-bounds, **B_BAD_INDEX** is returned. Otherwise, **B_NO_ERROR** is your reward.

See also: **SetField()**

Ref()

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

Returns the **record_ref** structure that identifies the record that lies behind this BMailMessage object. Not every object corresponds to a record. In general, an in-coming message (a BMailMessages that was constructed from a BRecord object) will have a ref, but an out-going message won't have a ref until the message is actually sent. As always when dealing with refs, you mustn't assume that the ref that's returned here is actually valid—the record may have been removed since the BMailMessage object was constructed (or since the message was sent).

Send()

long **Send(**bool *queue* = TRUE, bool *save* = TRUE**)**

Creates a record for this BMailMessage object, fills in the object's fields as appropriate for an out-going message in SMTP format, and then adds the record to the "E-Mail" table. If *queue* is **TRUE**, the record lies in the database until the mail daemon comes along of its own accord; if *queue* is **FALSE**, the mail daemon is told to send the message (and all other queued messages) right now. The BMailMessage's internal status (as returned by **Status()**) is set **B_MAIL_QUEUED** if queue is **TRUE**.

If save is **TRUE**, the record that holds the message remains in the database after the mail daemon has done its job. Otherwise, the record is destroyed after the message is sent.

The mail record's status is set to "Pending" by this function; when the mail daemon picks up the message, it (the daemon) will destroy the record (if it's not being saved), or change the status to "Sent".

If the BMailMessage doesn't appear to have any recipients, the **Send()** function returns **B_MAIL_NO_RECIPIENT** and the message isn't sent. If *queue* is **FALSE**, the function sends the message and returns the value returned by its (automatic) invocation of **check_for_mail()**. If the message is queued, the function returns **B_NO_ERROR**.

SetField(), RemoveField()

```
void SetField(char *field_name,
              void *data,
              long length,
              bool append = FALSE)

long RemoveField(char *field_name, long index = 0)
```

These functions add and remove fields (or field entries) from the object.

SetField() adds a field named *field_name*. The *data* and *length* arguments point to and describe the length of the data that you want the field to contain (the length is given in bytes). The final argument, *append*, states whether you want the data to be added (as a separate entry) to the data that already exists under the same name. If *append* is **FALSE**, the new data (the data that you're passing in this function call) becomes the field's only entry; if it's **TRUE**, and the field already exists, the "old" data isn't clobbered, and the field's "entry count" is increased by one.

RemoveField() removes the data that corresponds to the given field name. If the field contains more than one entry, you can selectively remove a specific entry through the use of the *index* argument. If *field_name* doesn't identify an existing field (in this object), **B_MAIL_UNKNOWN_FIELD** is returned; if the index is out-of-bounds, **B_BAD_INDEX** is returned. Otherwise, **B_NO_ERROR** is returned.

The field names that are defined by the class are:

Field Name	Constant
"To: "	**B_MAIL_TO**
"Cc: "	**B_MAIL_CC**
"Bcc: "	**B_MAIL_BCC**
"From: "	**B_MAIL_FROM**
"Date: "	**B_MAIL_DATE**
"Reply: "	**B_MAIL_REPLY**
"Subject: "	**B_MAIL_SUBJECT**
"Priority: "	**B_MAIL_PRIORITY**
"Content"	**B_MAIL_CONTENT**

See also: **FindField()**

SetEnclosure(), GetEnclosure(), RemoveEnclosure(), CountEnclosures()

void **SetEnclosure(**record_ref **ref*,
const char **mime_type*,
const char **mime_subtype***)**

long **GetEnclosure(**record_ref ***ref*,
char ***mime_type*,
char ***mime_subtype*,
long *index* = 0**)**;

long **RemoveEnclosure(**record_ref **ref***)**

long **CountEnclosures(**void**)**

These functions deal with a BMailMessage's "enclosures." An enclosure is a separate file that's included in the mail message. Enclosures are identified by index only—unlike a BMailMessage's fields, enclosures don't have names. Every enclosure is tagged with a MIME typifier. The MIME typifier is a human-readable string in the form "type/subtype" that attempts to describe the data that the enclosure contains. As shown in the protocol above, the BMailMessage class breaks the two MIME components apart so they can be set (or retrieved) separately.

SetEnclosure() adds an enclosure to the object. The *ref* argument locates the enclosure's data; currently, the refs that you add may only refer to files. The other two arguments let you tag the enclosure with MIME type and subtype strings. (Note that BeMail currently tags all out-going enclosures as "application/befile".)

GetEnclosure() returns, by reference through **ref*, a pointer to the ref that represents the object's *index*'th enclosure. The enclosure's MIME type strings are pointed to by **mime_type* and **mime_subtype*. The MIME strings that the arguments point to are **NULL**-terminated for you. If index is out-of-bounds **B_BAD_INDEX** is returned (this includes the no-enclosure case). Otherwise, **B_NO_ERROR** is returned.

RemoveEnclosure() removes the enclosure that's identified by the argument. If *ref* doesn't identify an existing enclosure, this function returns **B_BAD_INDEX** (look for the error return to change in a subsequent release). Otherwise, it returns **B_NO_ERROR**.

CountEnclosures() returns the number of enclosures that are currently contained in the object.

Status()

long **Status(**void**)**

Every BMailMessage has an internal state (that mustn't be confused with its record's status field) that tells whether the record that represents the object is currently queued to be sent. If it is, the status is **B_MAIL_QUEUED**, otherwise it's **B_MAIL_NOT_QUEUED**.

12 The Support Kit

12 The Support Kit

The Support Kit contains classes and utilities that any application can take advantage of—regardless of what kind of application it is or what it does. Among other things, it includes:

- The root BObject class
- The BList class
- A system for getting class information at run time
- Debugging tools including the BStopWatch class
- Common defined types, macros, and error codes

Class Information

Declared in: <support/ClassInfo.h>

The class-information system is a set of macros that you use to discover information about an object's class, and cast an object to pose as an instance of some other class.

Information

An object can supply three kinds of information about itself:

- What the name of its class is,
- Whether it's an instance of a particular class, and
- Whether its class derives from some other class (or perhaps *is* the other class).

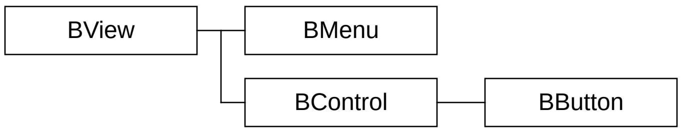
These three capabilities are embodied in the following macros,

> const char ***class_name(***object***)**
>
> bool **is_instance_of(***object*, *class***)**
>
> bool **is_kind_of(***object*, *class***)**

where *object* is a pointer to any type of object and *class* is a class designator—it's *not* a string name (for example, you would use **BView**, not "BView").

The **class_name()** macro returns the name of the object's class. **is_instance_of()** returns **TRUE** if *object* is an instance of *class*, and **FALSE** otherwise. **is_kind_of()** returns **TRUE** if *object* is an instance of a class that inherits from *class* or an instance of *class* itself, and **FALSE** if not.

For example, given this slice of the inheritance hierarchy from the Interface Kit,



and code like this that creates an instance of the BButton class,

```
BButton *anObject = new BButton(...);
```

these three macros would work as follows:

- The **class_name()** macro would return the string “BButton”:

  ```
  const char *s = class_name(anObject);
  ```

- The **is_instance_of()** macro would return **TRUE** only if the *class* passed to it is BButton. In the following example, it would return **FALSE**, and the message would not be printed. Even though BButton inherits from BView, the object is an instance of the BButton class, not BView:

  ```
  if ( is_instance_of(anObject, BView) )
      printf("The object is an instance of BView.\n");
  ```

- The **is_kind_of()** macro would return **TRUE** if *class* is BButton or any class that BButton inherits from. In the following example, it would return **TRUE** and the message would be printed. A BButton is a kind of BView:

  ```
  if ( is_kind_of(anObject, BView) )
      printf("The object is a kind of BView.\n");
  ```

Note that class names are not passed as strings.

Safe Casting

An object whose class participates in the class-information system will permit itself to be cast to that class or to any class that it inherits from. The agent for this kind of “safe casting” is the following macro,

class ***cast_as(***object*, *class***)**

where *object* is an object pointer and *class* is a class designator.

cast_as() returns a pointer to *object* cast as a pointer to an object of *class*, provided that *object* is a kind of *class*—that is, provided that it’s an instance of a class that inherits from *class* or is an instance of *class* itself. If not, *object* cannot be safely cast as pointer to *class*, so **cast_as()** returns **NULL**.

This macro is most useful when you have a pointer to a generic object and you want to treat it as a pointer to a more specific class.

Suppose, for example, that you retrieve a BWindow from a BView:

```
BWindow *window = myView->Window();
```

Furthermore, let's say that you suspect that the object that was returned is really an instance of a BWindow-derived class called MyWindow. If it is, you want to cast the object to be a MyWindow pointer. The **cast_as()** macro accomplishes this in one step:

```
MyWindow *mine;

if ( mine = cast_as(window, MyWindow) )
   /* mine is cast as a MyWindow object. */
else
   /* mine is set to NULL. */
```

Debugging Tools

Declared in: <support/Debug.h>

The Support Kit provides a set of macros that help you debug your application. These tools let you print information to standard output or to the serial port, and conditionally enter the debugger.

To enable the Support Kit's debugging tools you have to do two things:

- Compile your code with the **DEBUG** compiler variable defined
- Turn on the debug flag in your code through the **SET_DEBUG_ENABLED()** macro

These two acts represent, respectively, a compile time and a run time decision about the effectiveness of the debugging tools. The compile time decision overrides the run time decision: Turning on the debug flag (**SET_DEBUG_ENABLED(**TRUE**)**) has no affect if the **DEBUG** variable isn't defined.

The DEBUG Compiler Variable

Defining the **DEBUG** compiler variable can be done by adding the following line to your **makefile**:

```
USER_DEBUG_C_FLAGS := -DDEBUG
```

(The MetroWerks IDE will undoubtedly supply a means for setting the **DEBUG** variable through its interface. Check your local papers.)

When you're through debugging your application, simply remove the **DEBUG** definition and all of the debugging macros will be compiled away—you don't have to actually go into the code and remove the macros or comment them out.

The Debug Flag

The **SET_DEBUG_ENABLED()** macro turns on (or off) the debug flag. When you call a debugging tool, the state of the debugger flag is checked; if it's turned on, the tool does what it's designed to do (and, in with some tools, you could end up in the debugger). If the flag is off, the tool is ignored.

Note: The debug flag is on by default. If you want to (initially, at least) turn it off, you should call **SET_DEBUG_ENABLED(**FALSE**)** as one of your first acts in **main()**.

The run time aspect of the debug flag is particularly convenient if your application is large and you want to concentrate on certain sections of the code. Note, however, that the scope of the flag is application-wide. You can't, for example, disable the debugging tools across an object's member functions by simply calling **SET_DEBUG_ENABLED(**FALSE**)** in the object's constructor.

Macros

DEBUGGER(), ASSERT()

DEBUGGER(*var_args***)**
ASSERT(*condition***)**

These macros cause your program to enter the debugger: **DEBUGGER()** always enters the debugger, **ASSERT()** enters if *condition* (which can be any normal C or C++ expression) evaluates to **FALSE**.

DEBUGGER() takes a **printf()**-style variable-length argument that must be wrapped inside a second set of parentheses; for example:

```
DEBUGGER(("What time is it?  %f\n", system_time()));
```

The argument is evaluated and printed in the debugger's shell.

If **ASSERT()** enters the debugger, the following message is printed:

```
Assert failed: File: filename, Line: number. condition
```

Note that **ASSERT()**'s argument needn't be wrapped in a second set of parentheses.

HEAP_STATS()

HEAP_STATS()

Prints, to standard out, a message that gives statistics about your application's memory heap. The message appears in this format:

```
Heap Size: size bytes
Used blocks: count (size bytes)
Free blocks: count (size bytes)
```

PRINT(), SERIAL_PRINT()

> **PRINT(***var_args***)**
> **SERIAL_PRINT(***var_args***)**

These macros print the message given by *var_args*. The argument takes the variable argument form of a **printf()** call and must be wrapped inside a second set of parenthesis; for example:

```
    PRINT(("The time is %f\n", system_time()));
```

PRINT() sends the message to standard out; **SERIAL_PRINT()** to serial port 4 (the bottom-most serial port on the back of the computer).

PRINT_OBJECT()

> **PRINT_OBJECT(***object***)**

Prints information about the argument *object* (which must be a pointer to a C++ object) by calling the object's **PrintToStream()** function. The macro doesn't check to make sure that object actually implements the function, so you should use this macro with care.

Object information is always printed to standard out (there isn't a serial port version of the call).

SET_DEBUG_ENABLED(), IS_DEBUG_ENABLED()

> **SET_DEBUG_ENABLED(***flag***)**
> **IS_DEBUG_ENABLED(***void***)**

The **SET_DEBUG_ENABLED()** macro sets the state of the run time debug flag: A **TRUE** argument turns it on, **FALSE** turns it off. The utility of the other debugging macros depends on the state of the debug flag: When the flag is on, the macros work; when it's off, they're ignored. The debug flag is set to **TRUE** by default.

The debug flag is only meaningful if your code was compiled with the **DEBUG** compiler variable defined. Without the variable definition, the flag is always **FALSE**.

IS_DEBUG_ENABLED() returns the current state of the debug flag.

TRACE(), SERIAL_TRACE()

> **TRACE(***void***)**
> **SERIAL_TRACE(***void***)**

These macros print the name of the source code file that contains the currently executing code (in other words, the file that contains the **TRACE()** call itself), the line number of the code, and the **thread_id** of the calling thread. The information is printed in this form:

```
      File: filename, Line: number, Thread: id
```

TRACE() sends the message to standard out; **SERIAL_TRACE()** to serial port 4 (the bottom-most serial port on the back of the computer).

BList

Derived from:	public BObject
Declared in:	<support/List.h>

Overview

A BList object is a compact, ordered list of data pointers. BList objects can contain pointers to any type of data, including—and especially—objects.

Items in a BList are identified by their ordinal position, or index, starting with index 0. Indices are neither arbitrary nor permanent. If, for example, you insert an item into the middle of a list, the indices of the items at the tail of the list are incremented (by one). Similarly, removing an item decrements the indices of the following items.

A BList stores its items as type **void ***, so it's necessary to cast an item to the correct type when you retrieve it. For example, items retrieved from a list of BBitmap objects must be cast as BBitmap pointers:

```
BBitmap *theImage = (BBitmap *)myList->ItemAt(anIndex);
```

Note: There's nothing to prevent you from adding a **NULL** pointer to a BList. However, functions that retrieve items from the list (such as **ItemAt()**) return **NULL** when the requested item can't be found. Thus, you can't distinguish between a valid **NULL** item and an invalid attempt to access an item that isn't there.

Constructor and Destructor

BList()

BList(long *blockSize* = 20**)**
BList(const BList& *anotherList***)**

Initializes the BList by allocating enough memory to hold *blockSize* items. As the list grows and shrinks, additional memory is allocated and freed in blocks of the same size.

The copy constructor creates an independent list of data pointers, but it doesn't copy the pointed-to data. For example:

```
BList *newList = new BList(oldList);
```

Here, the contents of *oldList* and *newList*—the actual data pointers—are separate and independent. Adding, removing, or reordering items in *oldList* won't affect the number or order of items in *newList*. But if you modify the data that an item in *oldList* points to, the modification will be seen through the analogous item in *newList*.

The block size of a BList that's created through the copy constructor is the same as that of the original BList.

~BList()

virtual **~BList(**void**)**

Frees the list of data pointers, but doesn't free the data that they point to. To destroy the data, you need to free each item in an appropriate manner. For example, objects that were allocated with the **new** operator should be freed with **delete**:

```
void *anItem;
for ( long i = 0; anItem = myList->ItemAt(i); i++ )
    delete anItem;
delete myList;
```

See also: **MakeEmpty()**

Member Functions

AddItem()

bool **AddItem(**void **item*, long *index***)**
inline bool **AddItem(**void **item***)**

Adds an item to the BList at *index*—or, if no index is supplied, at the end of the list. If necessary, additional memory is allocated to accommodate the new item.

Adding an item never removes an item already in the list. If the item is added at an index that's already occupied, items currently in the list are bumped down one slot to make room.

If *index* is out-of-range (greater than the current item count, or less than zero), the function fails and returns **FALSE**. Otherwise it returns **TRUE**.

AddList()

> bool **AddList(**BList **list*, long *index***)**
> bool **AddList(**BList **list***)**

Adds the contents of another BList to this BList. The items from the other BList are inserted at *index*—or, if no index is given, they're appended to the end of the list. If the index is out-of-range, the function fails and returns **FALSE**. If successful, it returns **TRUE**.

See also: **AddItem()**

CountItems()

> inline long **CountItems(**void**)** const

Returns the number of items currently in the list.

DoForEach()

> void **DoForEach(**bool (**func*)(void *)**)**
> void **DoForEach(**bool (**func*)(void *, void *), void **arg2***)**

Calls the *func* function once for each item in the BList. Items are visited in order, beginning with the first one in the list (index 0) and ending with the last. If a call to *func* returns **TRUE**, the iteration is stopped, even if some items have not yet been visited.

func must be a function that takes one or two arguments. The first argument is the currently-considered item from the list; the second argument, if *func* requires one, is passed to **DoForEach()** as *arg2*.

FirstItem()

> inline void ***FirstItem(**void**)** const

Returns the first item in the list, or **NULL** if the list is empty. This function doesn't remove the item from the list.

See also: **LastItem()**, **ItemAt()**

HasItem()

> inline bool **HasItem(**void **item***)** const

Returns **TRUE** if *item* is in the list, and **FALSE** if not.

IndexOf()

long **IndexOf(**void **item***)** const

Returns the ordinal position of *item* in the list, or **B_ERROR** if *item* isn't in the list. If the item is in the list more than once, the index returned will be the position of its first occurrence.

IsEmpty()

inline bool **IsEmpty(**void**)** const

Returns **TRUE** if the list is empty (if it contains no items), and **FALSE** otherwise.

See also: **MakeEmpty()**

ItemAt()

inline void ***ItemAt(**long *index***)** const

Returns the item at *index*, or **NULL** if the index is out-of-range. This function doesn't remove the item from the list.

See also: **Items()**, **FirstItem()**, **LastItem()**

Items()

inline void ***Items(**void**)** const

Returns a pointer to the BList's list. You can index directly into the list if you're certain that the index is in-range:

```
myType item = (myType)Items()[index];
```

Although the practice is discouraged, you can also step through the list of items by incrementing the list pointer that's returned by **Items()**. Be aware that the list isn't null-terminated—you have to detect the end of the list by some other means. The simplest method is to count items:

```
void *ptr = myList->Items();

for ( long i = myList->ItemCount(); i > 0; i-- )
{
    . . .
    *ptr++;
}
```

You should *never* use the list pointer to change the number of items in the list.

See also: **DoForEach()**, **SortItems()**

LastItem()

inline void ***LastItem(**void**)** const

Returns the last item in the list without removing it. If the list is empty, this function returns **NULL**.

See also: **RemoveLastItem()**, **FirstItem()**

MakeEmpty()

void **MakeEmpty(**void**)**

Empties the BList of all its items, without freeing the data that they point to.

See also: **IsEmpty()**, **RemoveItem()**

RemoveItem()

bool **RemoveItem(**void **item***)**
void ***RemoveItem(**long *index***)**

Removes an item from the list. If passed an *item*, the function looks for the item in the list, removes it, and returns **TRUE**. If it can't find the item, it returns **FALSE**. If the item is in the list more than once, this function removes only its first occurrence.

If passed an *index*, the function removes the item at that index and returns it. If there's no item at the index, it returns **NULL**.

The list is compacted after an item is removed. Because of this, you mustn't try to empty a list (or a range within a list) by removing items at monotonically increasing indices. You should either start with the highest index and move towards the head of the list, or remove at the same index (the lowest in the range) some number of times. As an example of the latter, the following code removes the first five items in the list:

```
for ( long i = 0; i <= 4; i++ )
    myList->RemoveItem(0);
```

See also: **MakeEmpty()**

SortItems()

void ***SortItems(**int (**compareFunc*)(const void *, const void *)**)**

Rearranges the items in the list. The items are sorted using the *compareFunc* comparison function passed as an argument. This function should take two items as arguments. It should return a negative number if the first item should be ordered before the second, a positive number if the second should be ordered before the first, and 0 if the two items should be ordered equivalently.

See also: **Items()**

Operators

= (assignment)

BList& **operator =(**const BList&**)**

Copies the contents of one BList object into another:

```
BList newList = oldList;
```

After the assignment, each object has its own independent copy of list data; destroying one of the objects won't affect the other.

Only the items in the list are copied, not the data they point to.

BLocker

Derived from:	public BObject
Declared in:	<support/Locker.h>

Overview

The BLocker class provides a locking mechanism that protects a section of code. The code that you want to protect should be placed between BLocker's **Lock()** and **Unlock()** calls:

```
BLocker *aLock = new BLocker();

...
aLock->Lock();
/* Protected code goes here. */
aLock->Unlock();
```

This disposition of calls guarantees that only one thread at a time will pass through the lock. After a thread has locked the BLocker object, subsequent attempts to lock by other threads are blocked until the first thread calls **Unlock()**.

BLocker keeps track of its lock's "owner"—the thread that's currently between **Lock()** and **Unlock()** calls. It lets the lock owner make nested calls to **Lock()** without blocking. Because of this, you can wrap a BLocker's lock around a series of functions that might, themselves, lock the same BLocker object.

For example, let's say you have a class called BadDog that's declared thus:

```
class MyObject : public BObject
{
public:
    void DoThis();
    void DoThat();
    void DoThisAndThat();

private:
    BLocker lock;
};
```

And let's implement the member functions as shown below:

```
void BadDog::DoThis()
{
    lock.Lock();
```

```
        /* Do this here. */
        lock.Unlock();
    }

    void BadDog::DoThat()
    {
        lock.Lock();
        /* Do that here. */
        lock.Unlock();
    }

    void BadDog::DoThisAndThat()
    {
        lock.Lock();
        DoThis();
        DoThat();
        lock.Unlock();
    }
```

Notice that **DoThisAndThat()** wraps the lock around its calls to **DoThis()** and **DoThat()**, both of which contain locks as well. A thread that gets past the **Lock()** call in **DoThisAndThat()** will be consider the lock's owner, and so it won't block when it calls the nested **Lock()** calls that it runs into in **DoThis()** and **DoThat()**.

Keep in mind that nested **Lock()** calls must be balanced by equally-nested **Unlock()** calls.

Constructor and Destructor

BLocker()

BLocker(void**)**

BLocker(const char **name***)**

Sets up the object. The optional name is purely for diagnostics and debugging.

~BLocker()

virtual **~BLocker(**void**)**

Deletes the object. If there are any threads blocked waiting to lock the object, they're immediately unblocked.

Member Functions

IsLocked()

inline bool **IsLocked(**void**)** const

Checks to see whether the calling thread is the thread that currently owns the lock. If it is, **IsLocked()** returns **TRUE**. If it's not, **IsLocked()** returns **FALSE**

Lock(), Unlock()

void **Lock(**void**)**

void **Unlock(**void**)**

These functions lock and unlock the BLocker.

Lock() attempts to lock the BLocker and set the lock's owner to the calling thread. The function doesn't return until it has succeeded. While the BLocker is locked, non-owner calls to **Lock()** will block. The owner, on the other hand, can make additional, nested calls to **Lock()** without blocking.

Unlock() releases one level of nested locks and returns immediately. When the BLocker is completely unlocked—when all nested **Lock()** calls have been matched by calls to **Unlock()**—the lock's owner is "unset", allowing some other thread to lock the BLocker. If there are threads blocked in **Lock()** calls when the lock is released, the thread that's been waiting the longest acquires the lock.

Although you're not prevented from doing so, it's not good form to call **Unlock()** from a thread that doesn't own the lock. For debugging purposes, you can call **IsLocked()** before calling **Unlock()** to make sure this doesn't happen in your code.

See also: **LockOwner()**

LockOwner()

inline thread_id **LockOwner(**void**)** const

Returns the thread that currently owns the lock, or –1 if the BLocker isn't currently locked.

See also: **Lock()**

BObject

Derived from:	*none*
Declared in:	<support/Object.h>

Overview

BObject is the root class of the inheritance hierarchy. All Be classes (with just a handful of significant exceptions) are derived from it.

The primary reason for a single, shared base class is to provide common functionality to all objects. Currently, the BObject class is empty (except for its constructor and destructor), so there's no significant functionality to report. Subsequent releases will probably introduce new functions to the class; in anticipation of this, it's suggested that the classes you design derive from BObject (if no other Be class is a fit base).

In addition, when all objects are derived from BObject, the class can provide a generic type classification (BObject *) that simply means "an object." This can be a useful substitute for type **void ***.

Constructor and Destructor

BObject()

BObject(void**)**

Does nothing. Because the BObject class has no data members to initialize, the BObject constructor is empty.

~BObject()

virtual **~BObject(**void**)**

Does nothing. Because the BObject class doesn't declare any data members, the BObject destructor has nothing to free.

BStopWatch

Derived from: public BObject

Declared in: <support/StopWatch.h>

Overview

The BStopWatch class is a debugging tool that you can use to time the execution of portions of your code. The class has no member functions or (public) member data. When a BStopWatch object is constructed, it starts its internal timer. When it's deleted it stops the timer and prints the elapsed time to standard out in this format:

StopWatch "*name*": *f* usecs.

Where *name* is the name that you gave to the object when you constructed it, and *f* is the elapsed time in microseconds reckoned to one decimal place.

For example ...

```
#include <StopWatch.h>
...
BStopWatch *myWatch = new BStopWatch("Timer 0");
/* The code you want to time goes here. */
delete myWatch;
...
```

... would produce, on standard out, a message that goes something like this:

```
StopWatch "Timer 0": 492416.3 usecs.
```

This would indicate that the timed code took about half a second to execute—remember, you're looking at microseconds.

BStopWatch objects are handy little critters. They're particularly useful if you want to get a general idea of where your cycles are going. But you shouldn't rely on them for painfully accurate measurements.

Important: Unlike the other debugging tools defined by the Support Kit, there's no run-time toggle to control a BStopWatch. Make sure you remove your BStopWatch objects after you're done debugging your code.

Constructor and Destructor

BStopWatch()

BStopWatch(const char **name***)**

Creates a BStopWatch object, names it name, and starts its internal timer.

~BStopWatch()

virtual **~BStopWatch(**void**)**

Stops the object's timer, spits out a timing message to standard out, and then destroys the object and everything it believes in.

Functions, Constants, and Defined Types

This section lists the Support Kit's general-purpose functions (including function-like macros), constants, and defined types. These elements are used throughout the Be application-programming interface.

Not listed here are constants that are used as error codes. These are listed in "Error Codes" on page 33.

Functions and Macros

atomic_add(), atomic_and(), atomic_or()

long **atomic_add(**long **atomic_variable*, long *add_value***)**
long **atomic_and(**long **atomic_variable*, long *and_value***)**
long **atomic_or(**long **atomic_variable*, long *or_value***)**

These functions perform the named operations (addition, bitwise AND, or bitwise OR) on the value found in *atomic_variable*, thus:

```
*atomic_variable += add_value
*atomic_variable &= and_value
*atomic_variable |= or_value
```

The functions return the previous value of **atomic_variable* (in other words, they return the value that *atomic_variable* pointed to before the operation was performed).

The significance of these functions is that they're guaranteed to be *atomic*: If two threads attempt to access the same atomic variable at the same time (through these functions), one of the two threads will be made to wait until the other thread has completed the operation and updated the *atomic_variable* value.

class_name(), is_instance_of(), is_kind_of(), cast_as()

class_name(*object***)**
is_instance_of(*object*, *class***)**
is_kind_of(*object*, *class***)**
cast_as(*object*, *class***)**

These macros are part of the class information mechanism. In all cases, *object* is a pointer to an object, and class is a class designator (such as, literally, **BView** or **BFile**) and *not* a string (not “BView” or “BFile”).

class_name() returns a pointer to the name of *object*’s class.

is_instance_of() returns **TRUE** if object is a direct instance of class.

is_kind_of() returns **TRUE** if object is an instance of class, or if it inherits from class.

cast_as() if object is a kind of class (in the **is_kind_of()** sense), then **cast_as()** returns a pointer to object cast as an instance of class. Otherwise it returns **NULL**.

min(), max()

<support/SupportDefs.h>

min(*a*, *b***)**
max(*a*, *b***)**

These macros compare two integers or floating-point numbers. **min()** returns the lesser of the two (or *b* if they’re equal); **max()** returns the greater of the two (or *a* if they’re equal).

read_16_swap(), read_32_swap(), write_16_swap(), write_32_swap()

short **read_16_swap(**short **address***)**
long **read_32_swap(**long **address***)**

void **write_16_swap(**short **address*, short *value***)**
void **write_32_swap(**long **address*, long *value***)**

The **read...** functions read a 16- or 32-bit value from *address*, reverse the order of the bytes in the value, and return the swapped value directly.

The **write...** functions swap the bytes in *value* and write the swapped value to *address*.

real_time_clock(), set_real_time_clock(), time_zone(), set_time_zone()

long **real_time_clock(**void**)**
void **set_real_time_clock(**long *seconds***)**

long **time_zone(**void**)**
void **set_time_zone(**long *seconds***)**

These functions measure and set time in seconds:

- **real_time_clock()** returns a measure of the number of seconds that have elapsed since the beginning of January 1st, 1970. **time_zone()** is a time-zone based offset, in seconds, that you can add to the value returned by **real_time_clock()** to get a notion of the actual (current) time of day.

- **set_real_time_clock()** and **set_time_zone()** set the values for the system's clock and time zone variables.

Warning: The **time_zone()** and **set_time_zone()** functions are currently unimplemented. If you call them, you will crash.

These functions aren't intended for scrupulously accurate measurement.

See also: **system_time()** in the Kernel Kit

write_16_swap() *see* read_16_swap()

write_32_swap() *see* read_16_swap()

Constants

Boolean Constants

<support/SupportDefs.h>

Defined constant	Value
FALSE	0
TRUE	1

These constants are used as values for **bool** variables (the **bool** type is listed in the next section).

Empty String

<support/SupportDefs.h>

const char ***B_EMPTY_STRING**

This constant provides a global pointer to an empty string (“”).

NULL and NIL

<support/SupportDefs.h>

Defined constant	Value
NIL	0
NULL	0

These constants represent “empty” values. They’re synonyms that can be used interchangeably.

Defined Types

bool

<support/SupportDefs.h>

typedef unsigned char **bool**

This is the Be version of the basic boolean type. The **TRUE** and **FALSE** constants (listed above) are defined as boolean values.

Fumction Pointers

<support/SupportDefs.h>

typedef int (***B_PFI**)()
typedef long (***B_PFL**)()
typedef void (***B_PFV**)()

These types are pointers to functions that return **int**, **long**, and **void** values respectively.

Unsigned Integers

<support/SupportDefs.h>

typedef unsigned char **uchar**
typedef unsigned int **uint**

typedef unsigned long **ulong**
typedef unsigned short **ushort**

These type names are defined as convenient shorthands for the standard unsigned types.

Volatile Integers

<support/SupportDefs.h>

typedef volatile char **vchar**
typedef volatile int **vint**
typedef volatile long **vlong**
typedef volatile short **vshort**

These type names are defined as shorthands for declaring volatile data.

Volatile and Unsigned Integers

<support/SupportDefs.h>

typedef volatile unsigned char **vuchar**
typedef volatile unsigned int **vuint**
typedef volatile unsigned long **vulong**
typedef volatile unsigned short **vushort**

These type names are defined as shorthands for specifying an integral data type to be both unsigned and volatile.

Error Codes

Error codes are returned by various functions to indicate the success or to describe the failure of a requested operation. All Be error constants except for **B_NO_ERROR** are negative integers; any function that returns an error code can thus be generally tested for success or failure by the following:

```
if ( funcCall() < B_NO_ERROR )
    /* failure */
else
    /* success */
```

Furthermore, all constants (except **B_NO_ERROR** and **B_ERROR**) are less than or equal to the value of the **B_ERRORS_END** constant. If you want to define your own negative-valued error codes, you should begin with the value (**B_ERRORS_END** + 1) and work your way toward 0.

General Error Codes

<support/Errors.h>

Error Code	Meaning
B_NO_MEMORY	There's not enough memory for the operation.
B_IO_ERROR	A general input/output error occurred.
B_PERMISSION_DENIED	The operation isn't allowed.
B_FILE_ERROR	A file error occurred.
B_FILE_NOT_FOUND	The specified file doesn't exist.
B_BAD_INDEX	The index is out of range.
B_BAD_VALUE	An illegal value was passed to the function.
B_MISMATCHED_VALUES	Conflicting values were passed to the function.
B_BAD_TYPE	An illegal argument type was named or passed.
B_NAME_NOT_FOUND	There's no match for the specified name.
B_NAME_IN_USE	The requested (unique) name is already used.
B_TIMED_OUT	Time expired before the operation was finished.
B_INTERRUPTED	A signal interrupted the operation.
B_ERROR = –1	This is a convenient catchall for general errors.
B_NO_ERROR = 0	Everything's OK.
B_ERRORS_END	Marks the end of all Be-defined error codes.

Application Kit Error Codes

<support/Errors.h>

Error Code	Meaning
B_DUPLICATE_REPLY	A previous reply message has already been sent.
B_BAD_REPLY	The reply message is inappropriate and can't be sent
B_BAD_HANDLER	The designated message handler isn't valid.
B_MESSAGE_TO_SELF	A thread is trying to send a message to itself.
B_ALREADY_RUNNING	The application can't be launched again.
B_LAUNCH_FAILED	The attempt to launch the application failed.

These constants are defined for the messaging classes of the Application Kit. The messaging system also makes use of some of the general errors and kernel errors described above.

See also: **BMessage::Error()** and **BMessenger::Error()**

Debugger Error Codes

<support/Errors.h>

Error Code
B_DEBUGGER_ALREADY_INSTALLED

This constant signals that the debugger has already been installed for a particular team and can't be installed again.

Kernel Kit Error Codes

<support/Errors.h>

Error Code	Meaning
B_BAD_THREAD_ID	Specified thread identifier (**thread_id**) is invalid.
B_BAD_THREAD_STATE	The thread is in the wrong state for the operation.
B_NO_MORE_THREADS	All thread identifiers are currently taken.
B_BAD_TEAM_ID	Specified team identifier (**team_id**) is invalid.
B_NO_MORE_TEAMS	All team identifiers are currently taken.
B_BAD_PORT_ID	Specified port identifier (**port_id**) is invalid.
B_NO_MORE_PORTS	All port identifiers have been taken.
B_BAD_SEM_ID	Semaphore identifier (**sem_id**) is invalid.
B_NO_MORE_SEMS	All semaphores are currently taken.

B_BAD_IMAGE_ID	Specified image identifier (image_id)is inavlid.

These error codes are returned by functions in the Kernel Kit, and occasionally by functions defined in higher level kits.

Media Kit Error Codes

<support/Errors.h>

Error Code	Meaning
B_STREAM_NOT_FOUND	The attempt to locate the stream failed.
B_SERVER_NOT_FOUND	The attempt to locate the server failed.
B_RESOURCE_NOT_FOUND	The attempt to locate the resource failed.
B_RESOURCE_UNAVAILABLE	Permission to access the resource was denied.
B_BAD_SUBSCRIBER	The BSubscriber is invalid.
B_SUBSCRIBER_NOT_ENTERED	The BSubscriber hasn't entered the stream.
B_BUFFER_NOT_AVAILABLE	The attempt to acquire the buffer failed.

These error codes are defined for the Media Kit. See the classes and functions in that kit for an explanation of how they're used.

A. Message Protocols

A. Message Protocols

This appendix details the formats for all public messages produced and understood by Be system software. The list includes all system messages, all other messages that might find their way to your application (for example, through a drag and drop operation), and all messages that you can deliver to a Be application or a Be-defined class.

For information on the messaging system, see “Messaging” in *The Application Kit* chapter.

System Messages

Messages that are dispatched and handled in a message-specific manner are known as *system messages*. For the most part, these are messages that the system produces and that applications are expected to respond to (by implementing a hook function matched to the message), but some are messages that applications must produce themselves. They fall into three categories:

- *System-management messages* can be delivered to any BLooper,
- *Application messages* are consigned to the BApplication object, and
- *Interface messages* are reported to BWindow objects.

For information on the place of system messages in the messaging system, see “System Messages” in the introduction to *The Application Kit* chapter.

System Management Messages

System management messages are concerned with running the messaging system. The BLooper class in the Application Kit declares hook functions for two such messages. (See also “System Management Messages” on page 15 of *The Application Kit* chapter.)

B_HANDLERS_REQUESTED

This message asks a target BHandler to supply BMessenger objects as proxies for other BHandlers. The BLooper dispatches it by calling the target's **HandlersRequested()** function; the target should respond with a **B_HANDLERS_INFO** reply.

The **HandlersRequested()** functions implemented in the Application and Interface Kits look for the following data entries in the message. See those functions for details.

Data name	Type code	Description
"index"	**B_LONG_TYPE**	An index into a list of BHandlers kept by the target BHandler.
"name"	**B_STRING_TYPE**	The name of a BHandler.
"class"	**B_STRING_TYPE**	The name of a class derived from BHandler or "BHandler" itself.

Since applications initiate **B_HANDLERS_REQUESTED** messages, they are free to use whatever protocols prove useful for requesting BHandler proxies. The data entries listed above are simply those that the Be-defined functions expect.

B_QUIT_REQUESTED

This message contains no data. It simply asks a BLooper to quit its message loop and destroy itself. The Blooper dispatches the message by calling its own **QuitRequested()** function.

This message is reinterpreted by the BApplication object to mean a request to quit the application and by a BWindow object to mean a request to close the window. It's therefore also listed under "Application Messages" and "Interface Messages" below.

Application Messages

Application messages concern the application as a whole, rather than one specific window or thread. They're all received and handled by the BApplication object. See "Application Messages" on page 16 in the introduction to *The Application Kit* chapter for information on when they're produced and how they should be handled.

B_ABOUT_REQUESTED

This message contains no data entries. It requests the BApplication object to put a window on-screen with information about the application. Applications should produce it when the user chooses the "About . . ." item in the main menu. The BApplication object dispatches the message by calling its own **AboutRequested()** function.

B_ACTIVATE

This message contains no data entries. It instructs the application to make itself the active application. The BApplication object dispatches it by calling **Activate()**, defined in the BApplication class.

B_APP_ACTIVATED

This message informs the application that it has become the active application, or that it has ceded that status to another application. The BApplication object dispatches the message by calling **AppActivated()**.

It contains one data entry:

Data name	Type code	Description
"active"	**B_BOOL_TYPE**	**TRUE** if the application has just become the active application, and **FALSE** if it just gave up that status.

B_ARGV_RECEIVED

This message passes the BApplication object command-line strings, typically ones the user typed in a shell. The BApplication object dispatches it by calling **ArgvReceived()**.

The message has the two expected data entries for command-line arguments:

Data name	Type code	Description
"argc"	**B_LONG_TYPE**	The number of items in the "argv" array. This will be the same number that **BMessage::GetInfo()** for "argv" would report.
"argv"	**B_STRING_TYPE**	The command-line strings. Each argument is stored as an independent item under the "argv" name—that is, there's an array of data items, each of type **char ***, rather than a single item of type **char ****.

B_PANEL_CLOSED

This message notifies the application that the file panel has been removed from the screen. The BApplication object dispatches it by calling **FilePanelClosed()**.

The message has these data entries:

Data name	Type code	Description
"frame"	**B_RECT_TYPE**	The frame rectangle of the panel at the time it was closed. (The user may have resized it and relocated it on-screen.) The rectangle is recorded in screen coordinates.
"directory"	**B_REF_TYPE**	A **record_ref** reference to the last directory displayed in the panel.
"marked"	**B_STRING_TYPE**	The item that was selected in the Filters list when the panel closed.
"canceled"	**B_BOOL_TYPE**	**TRUE** if the panel was closed because the user operated the "Cancel" button and **FALSE** otherwise.

B_PULSE

This message contains no data entries. It's posted at regularly spaced intervals as a kind of timing mechanism. The BApplication object dispatches it by calling the **Pulse()** function declared in the BApplication class.

B_QUIT_REQUESTED

This message contains no data entries. Its dispatching (by calling **QuitRequested()**) is defined in the BLooper class. When it gets the message, the BApplication object interprets it to be a request to shut the entire application down, not just one thread. It consequently promulgates similar messages to all BWindow objects.

B_READY_TO_RUN

This message contains no data entries. It's delivered to the BApplication object to mark the application's readiness to accept message input after being launched. The BApplication object dispatches it by calling **ReadyToRun()**.

B_REFS_RECEIVED

This message passes the application one or more references to database records. It's typically produced by the Browser when the user chooses some files for the application to open. The BApplication object dispatches it by calling **RefsReceived()**.

The message has one data entry, which might be an array of more than one item:

Data name	Type code	Description
"refs"	**B_REF_TYPE**	One or more **record_ref** items referring to database records. Typically, the records are for documents the application is expected to open.

B_REFS_RECEIVED messages can also be dragged to and from Browser windows.

Interface Messages

Interface messages inform BWindow objects and their BViews about activity in the user interface. Unlike application messages, most of which consist only of a command constant, most interface messages contain data entries describing an event. They're all delivered to a BWindow object, which dispatches some to itself but most to its BViews.

See "Interface Messages" on page 41 in *The Interface Kit* chapter for a discussion of the events these messages report.

B_KEY_DOWN

This message reports that the user pressed a character key on the keyboard. It's dispatched by calling the **KeyDown()** function of the target BView, generally the window's focus view. Most keys produce repeated **B_KEY_DOWN** messages—as long as the user keeps holding the key down and doesn't press another key.

Each message contains the following data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the key went down, as measured in microseconds from the time the machine was last booted.
"key"	**B_LONG_TYPE**	The code for the key that was pressed.
"modifiers"	**B_LONG_TYPE**	A mask that identifies which modifier keys the user was holding down and which keyboard locks were on at the time of the event.
"char"	**B_LONG_TYPE**	The character that's generated by the combination of the key and modifiers.

"states"	**B_UCHAR_TYPE**	A bitfield that records the state of all keys and keyboard locks at the time of the event. Although declared as **B_UCHAR_TYPE**, this is actually an array of 16 bytes.

For most applications, the "char" code is sufficient to distinguish one sort of user action on the keyboard from another. It reflects both the key that was pressed and the effect that the modifiers have on the resulting character. For example, if the Shift key is down when the user presses the *A* key, or if Caps Lock is on, the "char" produced will be uppercase 'A' rather than lowercase 'a'. If the Control key is down, it will be the **B_HOME** character. A section of *The Interface Kit* chapter, "Keyboard Information" on page 47, discusses the mapping of keys to characters in more detail.

The "modifiers" mask explicitly identifies which modifier keys the user is holding down and which keyboard locks are on at the time of the event. The mask is formed from the following constants, which are explained under "Modifier Keys" on page 51 in the introduction to *The Interface Kit* chapter.

B_SHIFT_KEY	**B_COMMAND_KEY**	**B_CAPS_LOCK**
B_LEFT_SHIFT_KEY	**B_LEFT_COMMAND_KEY**	**B_SCROLL_LOCK**
B_RIGHT_SHIFT_KEY	**B_RIGHT_COMMAND_KEY**	**B_NUM_LOCK**
B_CONTROL_KEY	**B_OPTION_KEY**	
B_LEFT_CONTROL_KEY	**B_LEFT_OPTION_KEY**	**B_MENU_KEY**
B_RIGHT_CONTROL_KEY	**B_RIGHT_OPTION_KEY**	

The mask is empty if no keyboard locks are on and none of the modifiers keys are being held down.

The "key" code is an arbitrarily assigned number that identifies which character key the user pressed. All keys on the keyboard, including modifier keys, have key codes (but only character keys produce key-down events). The codes for the keys on a standard keyboard are shown in the "Key Codes" section on page 48 in *The Interface Kit* chapter.

The "states" bitfield captures the state of all keys and keyboard locks at the time of the key-down event. (At other times, you can obtain the same information through BView's **GetKeys()** function.)

Although it's declared as **B_UCHAR_TYPE**, the bitfield is really an array of 16 bytes,

```
uchar states[16];
```

with one bit standing for each key on the keyboard. For most keys, the bit records whether the key is up or down. However, the bits corresponding to keys that toggle keyboard locks record the current state of the lock. To learn how to read the "states" array, see "Key States" on page 56 in *The Interface Kit* chapter.

B_KEY_UP

< Key-up messages are not currently reported. >

B_MINIMIZE

This message instructs a BWindow to "minimize" itself—to replace the window on-screen with a small token—or to remove the token and restore the full window. The message is produced when the user double-clicks the window tab or the window token and is dispatched by calling the BWindow's **Minimize()** function.

It contains the following data:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the user acted, as measured in microseconds from the time the machine was last booted.
"minimize"	**B_BOOL_TYPE**	A flag that's **TRUE** if the window should be minimized to a token representation, and **FALSE** if it should be restored to the screen from its minimized state.

B_MOUSE_DOWN

This message reports that the user pressed a mouse button while the cursor was over the content area of a window. It's produced only for the first button the user presses—that is, only if no other mouse buttons are down at the time. The BWindow dispatches it by calling the target BView's **MouseDown()** function.

The message contains the following information:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the mouse button went down, as measured in microseconds from the time the machine was last booted.
"where"	**B_POINT_TYPE**	Where the cursor was located when the user pressed the mouse button, expressed in the coordinate system of the target BView—the view where the cursor was located at the time of the event.
"modifiers"	**B_LONG_TYPE**	A mask that identifies which modifier keys were down and which keyboard locks were on when the user pressed the mouse button.

"buttons"	**B_LONG_TYPE**	A mask that identifies which mouse button went down.
"clicks"	**B_LONG_TYPE**	An integer that counts the sequence of mouse-down events for multiple clicks. It will be 1 for a single-click, 2 for a double-click, 3 for a triple-click, and so on.

The "modifiers" mask is the same as for key-down events and is described under "Modifier Keys" on page 51 in *The Interface Kit* chapter.

The "buttons" mask identifies mouse buttons by their roles in the user interface. It may be formed from one or more of the following constants:

B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

Because a mouse-down event is reported only for the first button that goes down, the mask will usually contain just one constant.

The "clicks" integer counts clicks. It's incremented each time the user presses the mouse button within a specified interval of the previous mouse-down event, and is reset to 1 if the event falls outside that interval. The interval is a user preference that can be set with the Mouse preferences application.

Note that the only test for a multiple-click is one of timing between mouse-down events. There is no position test—whether the cursor is still in the vicinity of where it was at the time of the previous event. It's left to applications to impose such a test where appropriate.

B_MOUSE_MOVED

This message is produced when the user moves the cursor into, within, or out of a window. Each message captures a small portion of that movement. Messages aren't produced if the cursor isn't over a window or isn't moving. The BWindow dispatches each message by calling the **MouseMoved()** function of every BView the cursor touched in its path from its last reported location.

The message contains the following data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the event occurred, as measured in microseconds from the time the machine was last booted.

"where"	**B_POINT_TYPE**	The new location of the cursor, where it has moved to, expressed in window coordinates.
"area"	**B_LONG_TYPE**	The area of the window where the cursor is now located.
"buttons"	**B_LONG_TYPE**	Which mouse buttons, if any, are down.

The "area" constant records which part of the window the cursor is over. It will be one of the following constants:

B_CONTENT_AREA	The cursor is over the content area of the window.
B_CLOSE_AREA	The cursor is over the close button in the title tab.
B_ZOOM_AREA	The cursor is over the zoom button in the title tab.
B_TITLE_AREA	The cursor is inside the title tab, but not over either the close button or zoom button.
B_RESIZE_AREA	The cursor is over the area in the right bottom corner where the window can be resized.
B_MINIMIZE_AREA	< *Currently unused.* >
B_UNKNOWN_AREA	It's unknown where the cursor is, probably because it just left the window.

The "buttons" mask is formed from one or more of the following constants:

B_PRIMARY_MOUSE_BUTTON
B_SECONDARY_MOUSE_BUTTON
B_TERTIARY_MOUSE_BUTTON

If no buttons are down, the mask is 0.

B_MOUSE_UP

This message reports that the user released a mouse button. It's produced only for the last button the user releases—that is, only if no other mouse button remains down. The BWindow does not dispatch this message.

The message contains the following data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the mouse button went up again, as measured in microseconds from the time the machine was last booted.

"where"	**B_POINT_TYPE**	Where the cursor was located when the user released the mouse button, expressed in the coordinate system of the target BView—the view where the cursor was located when the button went up.
"modifiers"	**B_LONG_TYPE**	A mask that identifies which of the modifier keys were down and which keyboard locks were in effect when the user released the mouse button.

The "modifiers" mask is the same as for key-down events and is described under "Modifier Keys" on page 51 in *The Interface Kit* chapter.

B_PANEL_CLOSED

This message is delivered to the BWindow when the application or the user closes the save panel associated with the window. The BWindow dispatches it by calling its own **SavePanelClosed()** function.

The message contains the following data entries:

Data name	Type code	Description
"frame"	**B_RECT_TYPE**	The frame rectangle of the save panel at the time the panel was closed. (The user may have resized it and relocated it on-screen before it was closed.) The rectangle is specified in the screen coordinate system.
"directory"	**B_REF_TYPE**	A **record_ref** reference to the last directory displayed in the panel.
"canceled"	**B_BOOL_TYPE**	An indication of whether or not the panel was closed by user. It's **TRUE** if the user closed the panel by operating the "Cancel" button and **FALSE** otherwise.

B_PULSE

This message serves as a simple timing mechanism. It's posted at regularly spaced intervals and is dispatched by calling the **Pulse()** function of every BView that wants to participate.

The message typically lacks any data entries, but may contain this one:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the event occurred, as measured in microseconds from the time the machine was last booted.

B_QUIT_REQUESTED

This message is interpreted by a BWindow object as a request to close the window. It's dispatched by calling **QuitRequested()**, which is generally implemented by application classes derived from BWindow.

When the Application Server produces the message (for example, when the user clicks the window's close button), it adds the following data entry:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the event occurred, as measured in microseconds from the time the machine was last booted.

However, this information is not crucial to the interpretation of the event. You don't need to add it to **B_QUIT_REQUESTED** messages that are posted in application code.

B_SAVE_REQUESTED

This message is delivered to a BWindow when the user operates the save panel to request that a document be saved. It has the following data entries:

Data name	Type code	Description
"directory"	**B_REF_TYPE**	A **record_ref** reference to the directory where the document should be saved.
"name"	**B_STRING_TYPE**	The name of the file in which the document should be saved.

These entries are added to all messages reporting save-requested events. Generally, the message has **B_SAVE_REQUESTED** as its **what** data member. However, you can define a custom message to report the event, one with another constant and additional data entries.

If the command constant is **B_SAVE_REQUESTED**, the message is dispatched by calling the BWindow's **SaveRequested()** function; otherwise, it's not treated as a system message. See **RunSavePanel()** in the BWindow class of the Interface Kit.

B_SCREEN_CHANGED

This message reports that the screen configuration has changed. The BWindow dispatches it by calling its own **ScreenChanged()** function.

The message contains these data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the screen changed, as measured in microseconds from the time the machine was last booted.
"frame"	**B_RECT_TYPE**	A rectangle with the same dimensions as the pixel grid the screen displays.
"mode"	**B_LONG_TYPE**	The color space of the screen—currently **B_COLOR_8_BIT** or **B_RGB_32_BIT**.

B_VALUE_CHANGED

This message reports that the Application Server changed a value associated with a scroll bar—something that will happen repeatedly as the user drags the scroll knob and presses the scroll buttons. The BWindow dispatches it by calling the BScrollBar object's **ValueChanged()** function.

The message has these data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the value changed, as measured in microseconds from the time the machine was last booted.
"value"	**B_LONG_TYPE**	The new value of the object.

B_VIEW_MOVED

This message reports that a view moved within its parent's coordinate system. Repeated messages may be produced if the movement is caused by the user resizing the window, which in turn resizes the parent view. The BWindow dispatches each one by calling its **FrameMoved()** function.

The message contains the following data:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the view moved, as measured in microseconds from the time the machine was last booted.

"where"	**B_POINT_TYPE**	The new location of the left top corner of the view's frame rectangle, expressed in the coordinate system of its parent.

B_VIEW_RESIZED

This message reports that a view has been resized. Repeated messages are produced if the resizing is an automatic consequence of the window being resized. The BWindow dispatches each one by calling its **FrameResized()** function.

The message holds the following data.

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the view was resized, as measured in microseconds from the time the machine was last booted.
"width"	**B_LONG_TYPE**	The new width of the view's frame rectangle.
"height"	**B_LONG_TYPE**	The new height of the view's frame rectangle.
"where"	**B_POINT_TYPE**	The new location of the left top corner of the view's frame rectangle, expressed in the coordinate system of its parent. (A "where" entry is present only if the view was moved while being resized.)

The message has a "where" entry only if resizing the view also served to move it. The new location of the view would first be reported in a **B_VIEW_MOVED** BMessage.

B_WINDOW_ACTIVATED

This message reports that the window has become the active window or has relinquished that status. The BWindow dispatches the message by calling its **WindowActivated()** function, which notifies every BView with a similar function call.

The message contains two data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the window's status changed, as measured in microseconds from the time the machine was last booted.

"active"	**B_BOOL_TYPE**	A flag that records the new status of the window. It's **TRUE** if the window has become the active window, and **FALSE** if it is giving up that status.

B_WINDOW_MOVED

This message reports that the window has been moved in the screen coordinate system. Repeated messages are generated when the user drags a window. The BWindow dispatches each one by calling its **WindowMoved()** function.

The message has the following entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the window moved, as measured in microseconds from the time the machine was last booted.
"where"	**B_POINT_TYPE**	The new location of the left top corner of the window's content area, expressed in screen coordinates.

B_WINDOW_RESIZED

This message reports that the window has been resized. It's generated repeatedly as the user moves a window border. The BWindow dispatches each message by calling **WindowResized()**.

The message holds these data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the window was resized, as measured in microseconds from the time the machine was last booted.
"width"	**B_LONG_TYPE**	The new width of the window's content area.
"height"	**B_LONG_TYPE**	The new height of the window's content area.

B_WORKSPACE_ACTIVATED

This message reports that the active workspace has changed. It's delivered to all BWindow objects associated with the workspace that was previously active and with the one just activated. Each BWindow dispatches the message by calling its own **WorkspaceActivated()** function.

The message contains the following data:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the workspace was activated or deactivated, as measured in microseconds from the time the machine was last booted.
"workspace"	**B_LONG_TYPE**	The workspace that's the subject of the message.
"active"	**B_BOOL_TYPE**	A flag that records the new status of the workspace—**TRUE** if it has become the active workspace, and **FALSE** if it has ceased being the active workspace.

B_WORKSPACES_CHANGED

This message informs a BWindow object that the set of workspaces with which it is associated has changed. The BWindow dispatches the message by calling its own **WorkspacesChanged()** function.

The message has three data entries:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the set of workspaces associated with the window changed, as measured in microseconds from the time the machine was last booted.
"old"	**B_LONG_TYPE**	The set of workspaces where the window could appear before the change.
"new"	**B_LONG_TYPE**	The set of workspaces where the window can appear after the change.

B_ZOOM

This message instructs the BWindow object to zoom the on-screen window to a larger size—or to return it to its normal size. The message is produced when the user operates the zoom button in the window's title tab. The BWindow dispatches it by calling **Zoom()**, declared in the BWindow class.

The message has just one data entry:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the zoom button was clicked, as measured in microseconds from the time the machine was last booted.

Standard Messages

The software kits produce a few standard messages that aren't system messages—that aren't matched to a specific hook function. They're classified below as:

- Messages that are sent as replies, sometimes automatically, to other messages, and
- Messages that convey editing instructions.

Reply Messages

The following three messages are sent as replies to other messages.

B_HANDLERS_INFO

The various **HandlersRequested()** functions implemented in the Application and Interface Kits send this message as a reply to a **B_HANDLERS_REQUESTED** system message, which requests BMessenger proxies for BHandler objects. The reply message will contain one of two possible data entries:

Data name	Type code	Description
"handlers"	**B_MESSENGER_TYPE**	An array of one or more BMessenger objects corresponding to the BHandlers specified in the **B_HANDLERS_REQUESTED** message.
"error"	**B_LONG_TYPE**	An error code explaining why there is no "handlers" array.

B_MESSAGE_NOT_UNDERSTOOD

This message doesn't contain any data entries. It's sent as a reply to messages that the receiving thread's chain of BHandlers does not recognize. See **MessageReceived()** in the BHandler class.

B_NO_REPLY

This message doesn't contain any data entries. It's sent as a default reply to another message when the original message is about to be deleted. The default reply is sent only if a synchronous reply is expected and none has been sent. See the **SendReply()** function in the BMessage class.

Editing Messages

A handful of messages pass editable data or give an instruction to edit currently selected data. Because BTextViews are the only kit-defined objects that know how to display editable data, they're the only ones who can respond to these messages.

B_CUT, B_COPY, and B_PASTE

A BWindow posts these messages to its focus view (or to itself, if none of its views is currently in focus) when the user presses the Command-*x*, Command-*c*, and Command-*v* shortcuts. It puts only one data entry in the message:

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the user pressed the keyboard shortcut, as measured in microseconds from the time the machine was last booted.

BTextView objects respond to these messages. See the BTextView class in the Interface Kit for details.

B_SIMPLE_DATA

This message is a package for a single data element. It can theoretically contain any type of data, but only two entries are currently understood:

Data name	Type code	Description
"text"	**B_ASCII_TYPE**	A null-terminated string of characters.
"char"	**B_LONG_TYPE**	A single character.

A BTextView object can put this message together for a drag-and-drop operation, and can understand the message when it's dropped on or targeted to the view. When it produces the message, it puts the text that's currently selected into a "text" data entry, as described above. It understands the message with either a "text" or a "char" data entry; it inserts the characters at the current selection.

Interapplication Messages

The messages that a user drags and drops on a view might have their source in any application, including applications that come with the Be Operating System. Currently, the Browser is the only source for a published, public message. It will probably be a common source, since it permits users to drag representations of database records. The message in which the Browser packages the dragged information is identical to one that reports a refs-received event. It has a single entry named "refs" containing one or more **record_ref** (**B_REF_TYPE**) items and **B_REFS_RECEIVED** as the command constant. See "**B_REFS_RECEIVED**" above.

B. Application APIs

You can extend some Be applications by providing them with add-on modules, which they will load and integrate into the set of features they provide for the user. Currently, the Browser is the only Be application with a public API for add-on extensions.

The Browser

The Browser can accept add-on modules that deal with database records and that can be invoked from menu items to carry out discrete tasks. The modules should be compiled as on-add images (described in *The Kernel Kit* chapter), and should be placed in the **/system/add-ons/Browser** directory.

The Browser will create an item for its Add-Ons menu with the same name as the add-on file. If the name ends in a hyphen plus one character, that character will be the keyboard shortcut for the item. For example, if the file is

/system/add-ons/Browser/Recede in Time-r

the Browser will add a “Recede in Time” item to its Add-Ons menu and assign it Command-r as a keyboard shortcut. The shortcut should not conflict with any that the Browser already uses.

The Browser loads the add-on module whenever the user operates the item. The add-on must provide the Browser with a single entry point, a function named **process_refs()**. It has the following syntax:

void **process_refs(**record_ref *directory*, BMessage **message*, void **data***)**

The *directory* is a reference to the directory the Browser is currently displaying in the active window. The *message* is a standard **B_REFS_RECEIVED** BMessage. It has a “refs” entry with **record_ref**s for all the items in the directory that are currently selected. The *data* argument is unused at present; ignore it.

After it loads the add-on image, the Browser creates a thread for it and calls its **process_refs()** function in that thread. When **process_refs()** returns, the Browser unloads the image. The add-on should make sure that any additional threads that it spawned are destroyed before it returns—especially any windows it displayed to the user.

< The Browser invokes **process_refs()** each time the user operates the menu item. If the user operates the item a second time before the first invocation of **process_refs()** returns, two instances of the function will be executing, each in its own thread. Unfortunately, when either instance returns, the Browser will unload the add-on image, leading to predictable undesired consequences. This is a known bug that will be repaired in a future release. >

To compile the add-on image, follow the directions for shared libraries in the Metrowerks *CodeWarrior* manual. In summary, you should specify the following options to the linker:

- **–G**, to tell the linker to produce an add-on image.
- **–export pragma**, to tell it that your source code has **#pragma** directives exporting the **process_refs()** symbol. Then surround the definition of the function with directives that turn exporting on and off:

  ```
  #pragma export on
  void process_refs(record_ref dir, BMessage *msg, void *data)
  {
      . . .
  }
  #pragma export off
  ```

  This gives the Browser the access it needs to call the function.

You can link the module against the system library; you shouldn't link it against the Browser.

Once compiled, place the module in the **/system/add-ons/Browser** directory, as discussed above. This is the only place the Browser looks for modules to load.

C. User Interface Guidelines

< The user interface guidelines are forthcoming and will be posted on the Be web site when ready. >