The Be Book™

The Software Development Environment for the BeBox™
Release 1.1 d6

The Be Book: The Software Development Environment for the BeBox
reference documentation for Be system software release 1.1 d6
revised December 1995

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 built into the hardware. Among other things, the Be machine offers:

- Parallel processing on two high-performance CPUs.
- An operating system designed for efficient multitasking. It automatically splits assignments between the CPUs and will give priority to applications that need uninterrupted service.
- A built-in ability to connect to and interact with the telephone.
- 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.

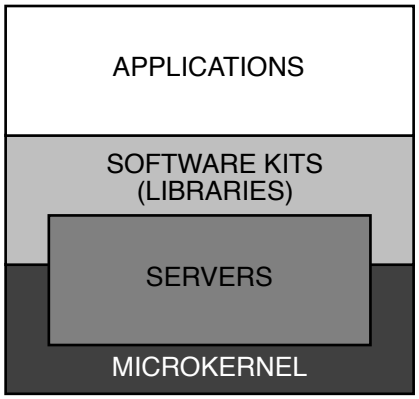
Be system software is designed to make the features of the BeBox readily and efficiently available to all applications. The application programming interface (API) is written in the C++ language and takes advantage of the opportunities C++ offers for object-oriented programming. It includes numerous 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":

- An 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 application programming interface (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 interacting with a telephone line >, and so on.

With the exception of the Kernel and Network Kits, 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.

Your 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.

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 <and some of those currently under development> 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 build 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. < It will come under the umbrella of the Media Kit in a future release. >

- < The *Telephone Kit* will be used to make connections to a telephone line, and to read and write data over the connection. >

- 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* provides programming interfaces to the various connectors and devices that can be attached to a BeBox. It currently consists of only the BSerialPort class.
- The *Network Kit* is for linking to and communicating with other computers through the TCP/IP and UDP/IP protocols. With only a couple of exceptions, its API is compatible with the BSD (Berkeley Software Distribution) UNIX socket architecture.
- 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 (application programming interface) is currently available. The present version covers eight kits—the Application, Storage, Interface, Media, Midi, Kernel, Device, and Support Kits summarized above. It doesn't yet cover the API for printing or for connecting to the telephone. 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. 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 die 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 **Name()** function like this one,

```
char *name = someObject->Name();
```

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. The object might modify the string, change its location in memory, reallocate it, or free it at any time. If your application needs continued access to the string, it should make a copy for itself or call **Name()** each time the string is needed.

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

```
char name[MAX_LENGTH + 1];
someObject->GetName(name);
```

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 BFile 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("name", B_RAW_TYPE, &numBytes);
```

The BFile 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 (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 always 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 finishing 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 non-hook 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.

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.

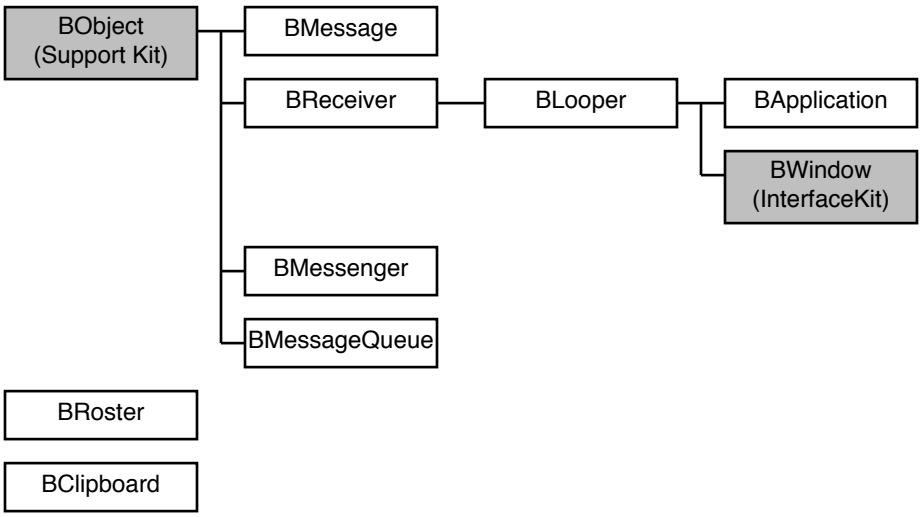
In addition, you can rely on our continuing to follow the lexical conventions established in the current API. For example, we never abbreviate "point" or "message," but always abbreviate "rectangle" as "rect" and "information" as "info." We use "begin" and "end," never "start" or "finish," in function 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 name the Interface Kit assigns to a window's root view is "_topview_". If you were to assign the same name to one of your own views, it 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. The classes in this Kit 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 or sent to another application. BLooper objects run message loops in threads, getting messages as they arrive and dispatching them to BReceiver objects. BReceivers are the ultimate message handlers.

  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 be prepared 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 remote messages from other applications and internal 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 and handling them when they arrive. This task is entrusted to BLooper objects.
- Letting applications define their own message-handling code. An arriving message is dispatched by calling a "hook" function of a BReceiver object. Each application can implement these functions as it sees fit.
- Making a connection to a remote application. BMessenger objects send messages to remote destinations. The BRoster object helps by providing information about other applications, launching them if necessary.

Messages

BMessage objects are containers for 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. The data is stored in named arrays along with information on the type, size, and number of items. 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. 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. 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 specialties.

The software kits define protocols for a number of system messages. They're discussed later in this and following chapters.

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 posted 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 that run *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 run 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 BReceiver objects. BReceivers are the primary handlers for arriving messages. Everything that a thread does begins with a BReceiver's response to a message.

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

- BLooper's **DispatchMessage()** function is called to pass responsibility for a message to a BReceiver object. It's fully implemented by BLooper (and kit classes derived from BLooper) and is only rarely overridden by applications.
- **MessageReceived()** is the BReceiver 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 BReceiver role of responding to messages. The BLooper class inherits from BReceiver, so the same object can fill both roles.

System Messages

Applications are typically designed to respond to 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.

These events are reported to applications as messages—BMessage objects. The system produces these messages as it monitors <changes to the file system, the making and breaking of connections on the telephone line,> user actions on the keyboard and mouse, <the flow of real-time audio and video data,> and other basic events that can affect what an application does and the environment in which it does it.

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.

Kinds of System Messages

System messages can be divided into two groups:

- Those that name an external event, such as **B_KEY_DOWN**, **B_SCREEN_CHANGED**, and **B_REFS_RECEIVED**.
- Those that name an action the receiver is expected to take, such as **B_ZOOM** or **B_ACTIVATE**.

Most system messages fall in the first group. Even those that fall in the second group 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.

Specialized BLoopers

System messages aren't delivered to just any BLooper object. Each message is matched to an object that's concerned with the particular event it reports or the particular instruction it delivers. Certain kinds of messages are delivered to certain BLoopers.

The software kits derive a few specialized classes from BLooper to give significant entities in the application their own message loops. These objects are the ones that handle system messages.

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 13 below) and BWindow objects with *interface messages* (discussed in the chapter on the Interface Kit). Both classes arrange for special handling of the system messages they receive.

Message-Specific Dispatching

Every system message is dispatched by calling a specific virtual 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 BReceiver 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.

Default Dispatching

System messages are identified 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 receiving BApplication or BWindow object will dispatch it by calling the message-specific hook function—regardless what data entries the message may have. However, if the constant doesn't match one of those defined for a system message, BApplication and BWindow objects dispatch it just like other BLoopers do—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.

Posting 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. Messages can either be *posted* to a thread of the same application or sent to another application.

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.

This 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.

Posting System Messages

A posted message might even match one that the system defines. For example, an application might interpret a user action such as clicking a "Quit" menu item as a "quit-requested" event and post the appropriate system message to the BApplication object. As will be noted under "Application Messages" on page 13 below, some system messages are designed to be posted within the application. (Most, however, are posted by the system.)

Linking Receivers to Loopers

PostMessage() permits a message to be targeted to a particular BReceiver object. To dispatch the message, the BLooper calls the targeted receiver's **MessageReceived()** function. The BLooper acts as the default receiver for untargeted messages. (Note, however, that targeting works only for application-defined messages; a system message is always dispatched to a BReceiver that's chosen on the basis of the content of the message, not to the target proposed by **PostMessage()**.)

A BReceiver can be tied to a particular message loop (by implementing a **Looper()** function) and a BLooper can name the object it prefers to receive messages (by implementing a **PreferredReceiver()** function). See the BLooper and BReceiver class descriptions for details on these functions.

Sending Remote 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 application, it's necessary to first set up a BMessenger object that knows how to contact the remote application. Each BMessenger is linked to one remote destination; it represents the remote application in the local address space.

A BMessenger's **SendMessage()** function delivers messages to the main thread of the remote application. There, the BApplication object receives the message and determines how to respond to it, including whether to dispatch it to another object. All remote messages are received first by the BApplication object.

Unlike **PostMessage()**, **SendMessage()** can't name a receiver for the message; that's left up to the remote application.

Two-Way Communication

A BMessage sent to a remote destination carries the identity of the source application with it. The receiver can send an answer back to the source by calling the BMessage's **SendReply()** function. **SendReply()** succeeds only if the message comes from a remote source.

A message sender can ask for a reply when calling **SendMessage()**, for example:

```
BMessage *reply;
MyMessenger->SendMessage(original, &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.) If a reply isn't requested, **SendMessage()** returns immediately.

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

The receiver can send a reply even if the sender isn't waiting for one. In this case, the BApplication object receives the reply message and dispatches it by calling **ReplyReceived()**. **ReplyReceived()** is an alternative to **MessageReceived()**; it's called only for reply messages that are matched to an original message.

Thus, the messaging mechanism supports both synchronous and asynchronous messaging protocols. Synchronous return messages are requested when calling **SendMessage()** and are received by that function. Asynchronous return messages are received by the BApplication object and handled by **ReplyReceived()**.

The Roster

A global BRoster object, shared by all applications on the BeBox, maintains a roster of running applications. It can provide you with any information you might need to set up a BMessenger for a particular application. It can also find information about applications that haven't yet been launched and, if need be, launch them so that they can receive messages.

The BRoster is accessed through a global variable, **be_roster**.

Application Messages

Although the Application Kit implements the messaging mechanism and defines all the system messages, it handles only a few of them itself—eleven to be exact. The others are handled by other kits, especially the Interface Kit, and are documented in the chapters on those kits.

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

- An *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 *ready-to-run* event occurs when the application has finished launching and configuring itself and its main thread is ready to respond to messages.
- An *application-activated* event occurs 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 events are requests that the application usually makes of itself:

- A *quit-requested* event occurs when there's a request that the application shut itself down. An application that has a user interface usually lets the user make this decision. It must interpret some user action (such as clicking a "Quit" menu item) as a request to quit and, in response, post a **B_QUIT_REQUESTED** message to the BApplication object. An application that serves at the pleasure of other applications may get the request from a remote source.

- An *about-requested* event occurs when the user requests information about the application, usually through an "About..." item in the application's master menu. The application should set up this item to post a **B_ABOUT_REQUESTED** message to the BApplication object.

Other application messages report information from remote sources:

- An *argv-received* event occurs either on-launch or after-launch when the application receives strings of characters the user typed on the command line. It also occurs when the application is launched by another application and is passed a similar array of character strings.

- A *refs-received* event occurs when the application is passed 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 *panel-closed* event occurs when the file panel is removed from the screen.

- A *volume-mounted* event occurs when a new volume (possibly a floppy disk) is mounted.

- A *volume-unmounted* event occurs when a volume is about to be unmounted.

The system is the source of one event:

- Periodic *pulse* events occur at regularly spaced intervals. They can be used to arrange repeated actions when precise timing is not critical.

Hook Functions for Application Messages

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 receiver. The following charts list the hook functions that are called to initiate the application's response to system messages and the base class where the function is declared:

Instruction type	Virtual function	Class
Activate	**Activate()**	BApplication
Event type	**Virtual function**	**Class**
Ready-to-run	**ReadyToRun()**	BApplication
Application-activated	**AppActivated()**	BApplication
Quit-requested	**QuitRequested()**	BLooper and BApplication
About-requested	**AboutRequested()**	BApplication
Argv-received	**ArgvReceived()**	BApplication
Refs-received	**RefsReceived()**	BApplication
Panel-closed	**FilePanelClosed()**	BApplication
Volume-mounted	**VolumeMounted()**	BApplication
Volume-unmounted	**VolumeUnmounted()**	BApplication
Pulse	**Pulse()**	BApplication

QuitRequested() is first declared in the BLooper class. It's reinterpreted (and reimplemented) by BApplication to mean a request to quit the whole application, not just one thread.

Message Protocols

Each system message has a **what** data member that names the instruction it gives or the event it reports:

B_ACTIVATE

B_READY_TO_RUN
B_APP_ACTIVATED

B_QUIT_REQUESTED
B_ABOUT_REQUESTED

B_ARGV_RECEIVED
B_REFS_RECEIVED
B_PANEL_CLOSED
B_VOLUME_MOUNTED
B_VOLUME UNMOUNTED

B_PULSE

The messages that report ready-to-run, quit-requested, about-requested, and pulse events are empty, as is the message for an activate request. The entire message is conveyed by the **what** constant.

The remaining messages contain data in the formats listed below.

Application-Activated Events

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.

Argv-Received Events

Data name	Type code	Description
"argc"	**B_LONG_TYPE**	The number of items in the "argv" array. This will be the same number that **GetInfo()** for "argv" would report.
"argv':	**B_STRING_TYPE**	The command-line arguments. 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 ****.
'vol"	**B_LONG_TYPE**	The identifier for the current volume of the message sender.
"dir"	**B_LONG_TYPE**	The identifier for the message sender's current directory.

The "vol" and "dir" entries can be used to interpret any relative pathnames in the "argv'' array.

Refs-Received Events

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.

Panel-Closed Events

Data name	Type code	Description
“frame”	**B_RECT_TYPE**	The frame rectangle of the panel in screen coordinates at the time it was closed. (The user may have resized it and relocated it onscreen.)
“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.

Volume-Mounted Events

Data name	Type code	Description
< “volume id”>	**B_LONG_TYPE**	The volume identifier.
< “db id” >	**B_LONG_TYPE**	The identifier for the database corresponding to the volume.

Volume-Unmounted Events

Data name	Type code	Description
< “volume id”>	**B_LONG_TYPE**	The volume identifier.

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 resources associated with the executable file. 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 x 32 pixel) version of the icon and a smaller (16 pixel x 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) 462-4118). 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. For example, 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 and not be confusing, and similar considerations.

The best place to choose a launch behavior for your application is in Icon World's "App Info" menu. If a choice isn't made, **B_MULTIPLE_LAUNCH** is assumed.

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 menu.

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 are sent by other applications (including the Browser), as well as "internal" messages that affect the entire application (such as a message requesting the application to quit). 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 15.

If your application expects to get messages from remote sources, it should also implement **MessageReceived()** and **ReplyReceived()** functions 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 BReceiver 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 ('abed') 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. **Quit()** is called 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.

Locking

Since a single BApplication object serves the entire application, and since different parts of the application will be running in separate threads (windows, in particular), you sometimes have to coordinate access to the BApplication object. 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).

BApplication inherits the locking mechanism—the **Lock()** and **Unlock()** 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 **SetMenuMenu()** description on page 38. The default main menu includes such an item.

See also: SetMenuMenu(), 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 the user, for example, 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() in the BWindow class of 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: WindowActivated() in the BWindow and BView classes of the Interface Kit, "Application-Activated Events" on page 16 of the chapter introduction

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(), "Argv-Received Events" on page 16 of the chapter introduction

CloseFilePanel() *see* RunFilePanel()

CountWindows()

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

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

See also: the BWindow class in the Interface Kit

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message,* BReceiver **receiver***)**

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 13 of the chapter introduction.

Other messages—those defined by the application rather than the Application Kit—are forwarded to the *receiver's* **MessageReceived()** function or to **ReplyReceived()**. Note that the *receiver* is ignored for all system messages and for all replies.

See also: DispatchMessage() in the BLooper class, **MessageReceived()** in the BReceiver class, **ReplyReceived()**

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: "Panel-Closed Events" on page 17 of the chapter introduction, **RunFilePanel()**

GetAppInfo()

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

Writes information about the application into the **appinfo** 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->Thread(),
                                  &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: GetAppInfo() in the BRoster class

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()

IdleTime()

long **IdleTime(**void**)** const

Returns the number of seconds since the user last manipulated the mouse or keyboard. This information isn't specific to a particular application; in other words, the function tells you when the user last directed an action at *any* application, not just yours.

IsCursorHidden()

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

Returns **TRUE** if the cursor is hidden, 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: MenusWillShow() in the BWindow class of the Interface Kit, **SetMainMenu()**

Modifiers()

ulong **Modifiers(**void**)**

Returns a mask indicating which modifier keys are down and which keyboard locks are on. This function works just like the BView and BWindow functions of the same name. See those functions for information on the modifiers mask.

See also: Modifiers() in the BWindow and BView classes of the Interface Kit

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 **PULSE** messages, as long as no other messages are pending. By default, it's called about every 500 milliseconds, but you can set a different frequency by calling the **SetPulseRate()** function.

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

The default version of this function is empty.

See also: Pulse() in the Window class of 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: Quit() in the BLooper class, **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 ongoing 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: QuitRequested() in the BLooper class, Quito, **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 the 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 (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.OpenData() == 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()**, "Refs-Received Events" on page 16 of the chapter introduction

ReplyReceived()

virtual void **ReplyReceived(**BMessage **message*, BMessage **original*,
ulong *signature*, thread_id *thread***)**

Implemented by derived classes to respond to a remote *message* that was sent in reply to an earlier message originating from your application. The *original* message is provided as the second argument, so you can match it to the reply.

The reply comes from the application that's identified both by its *signature* and its main *thread*; these values can be used to construct a BMessenger so that you can continue the exchange of messages. A reply to the reply can also be sent by calling the *message's* **SendReply()** function.

ReplyReceived() is called only for messages that are sent as replies to an earlier message (by calling the original message's **SendReply()** function) and only if the original sender isn't waiting for a reply.

< Reply messages that match system messages are dispatched by calling this function, rather than as system messages. >

See also: SendMessage() in the BMessenger class, **SendReply()** in the BMessage class

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, **Run()** in the BLooper class, **ReadyToRun()**, **QuitRequested()**

RunFilePanel(), CloseFilePanel(), IsFilePanelRunning()

long **RunFilePanel(**const char **windowTitle* = NULL,
const char **buttonLabel* = 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 a *buttonLabel* isn't provided, the principal button in the panel (the default button) will be labeled "Open". (The panel also has a "Cancel" button.)

- 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_REFTYPE**	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 die 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_B_FILE_TYPE)
model->AddLong("Picture files only" MY_IMAGE_A_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 *buttonLabel*) 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 sends 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 doesn't automatically disappear when the user operates the "Open" (or *buttonLabel*) button; 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" 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 default cursor is the hand image that's seen when the computer first boots. To reset the cursor to this image, you can pass the constant **B_HAND_CURSOR** to **SetCursor()**.

SetPulseRate()

void **SetPulseRate(**long *milliseconds***)**

Sets how often **Pulse()** is called. By default, **PULSE** messages are posted every 500 milliseconds, as long as no other message are pending. Each message causes **Pulse()** to be called once.

SetPulseRate() permits you to set a different interval. The interval set should be a multiple of 100; differences less than 100 milliseconds will not be noticeable. A finer granularity can't be guaranteed.

See also: Pulse()

ShowCursor() *see* HideCursor()

VolumeMounted(), VolumeUnmounted()

virtual void **VolumeMounted(**BMessage **message***)**

virtual void **VolumeUnmounted(**BMessage **message***)**

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 is stored in the *message* as a **long** integer under the name < "volume_id" >. It can be passed to **volume_for()** to get the corresponding BVolume object.

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

< Don't rely on these functions; they're likely to change in a future release. >

See also: the BVolume class in the Storage Kit, "Volume-Mounted Events" and "Volume-Unmounted Events" on page 17

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**.

Lock(), Unlock()

void **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.

See also: Lock() in the BLooper class

FindData(), FindText()

void ***FindData(**ulong *type*, long **numBytes*, long *index* = 0**)**

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

Returns a pointer to a particular item that lies on the clipboard. In the **FindData()** function, the item is specified by the first and last arguments:

- *type* is one of the data type constants defined in AppDefs.h.
- *index*, if supplied, specifies the (zero-based) index of the item that you want to retrieve. This is only important if the clipboard holds more than one item of the specified 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 *index* argument (to **FindData()**) is out-of-bounds, the function returns a pointer to **NULL** and, perhaps more tellingly, sets *numBytes* to 0. If you don't lock the clipboard before invoking either **FindData()** or **FindText()**, you'll find the debugger.

Unlock() *see* Lock()

BLooper

Derived from: public BReceiver

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(int priority) : BLooper(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.)

Posting and Receiving Messages

Messages are posted to the BLooper's thread by calling its **PostMessage()** function. This simply puts the message in a queue. 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 BReceiver object; the BReceiver kicks off the thread's specific response to the message.

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

Acting as the Receiver

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

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

However, it's not necessary to derive a class from BLooper. A BLooper can be used without change, as it's defined in the Kit—as long as all messages are posted to a another receiver.

Hook Functions

DispatchMessage()	Dispatches messages to a BReceiver; can be overridden to change the way certain messages or classes of messages are handled.
PreferredReceiver()	Can be implemented to indicate a preference for a particular BReceiver to which messages should be posted.
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(long *priority* = B_DISPLAY_PRIORITY**)**

Initializes the BLooper object 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. Four discrete priority levels are defined (in **kernel/OS.h**), but intermediate priorities are also possible. The defined priorities are:

Constant	Value	Usage
B_REAL_TIME_PRIORITY	40	For threads that control real-time processes that shouldn't be interrupted.
B_DISPLAY_PRIORITY	30	For threads associated with objects in the user interface, including window threads.
B_NORMAL_PRIORITY	20	For all ordinary threads, including the main thread.
B_LOW_PRIORITY	10	For threads that don't have much importance and shouldn't interrupt other threads.

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()

~BLooper()

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

Frees the message queue and all pending messages, stops the message loop, and destroys the thread in which it ran.

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

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.

In addition to returning the current message, **DetachCurrentMessage()** detaches it 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 the thread is 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—assuming the message sender is waiting for a reply. 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.

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 BReceiver'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: MessageReceived() in the BReceiver class

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BReceiver **receiver***)**

Dispatches messages as they're received by the BLooper's thread. **B_QUIT_REQUESTED** messages are dispatched by calling the **QuitRequested()** virtual function. All others are forwarded to the *receiver's* **MessageReceived()** function.

The *receiver* may be the BReceiver object that was named when the *message* was posted, or it may be the BLooper (acting in its capacity as the default receiver). It is never **NULL**.

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.

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, and
- Exclude all messages that you've handled yourself from the base version call.

For example:

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

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

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

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. Functions you define in classes derived 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()

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

Returns the thread that currently has the BLooper locked, or -1 if the BLooper isn't locked.

See also: Lock()

Looper()

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

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

See also: Looper() in the BView class of the Interface Kit and in the BReceiver class, **PostMessage()**

MessageQueue()

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

Returns the queue that holds messages posted (or sent, in the case of a BApplication object) 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()

virtual long **PostMessage(**BMessage **message*, BReceiver **receiver* = NULL**)**

long **PostMessage(**ulong *command*, BReceiver **receiver* = NULL**)**

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

However, if the named *receiver* is associated with a different BLooper (if the *receiver's* **Looper()** function returns some other BLooper object), the posting fails and the *message* is deleted. (If a BReceiver is associated with a particular BLooper, the only messages it can receive are ones posted to that object. For example, BViews in the Interface Kit are restricted to receiving messages posted to the BWindows to which they're attached.)

Although the named *receiver* is passed through to **DispatchMessage()**, that function may ignore the argument, especially if the message matches one defined by the system. System messages are dispatched to a BReceiver that's determined by the content of the message, not by the value passed to **PostMessage()**.

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 virtual version—the version that takes a full BMessage argument. Thus, if you override just the virtual 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 receiver is invalid, and **B_ERROR** if it fails because the BLooper itself is invalid.

See also: Looper() in the BReceiver class, **DispatchMessage()**

PreferredReceiver()

virtual BReceiver ***PreferredReceiver(**void**)** const

Implemented by derived classes to return a preferred BReceiver 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 receiver. For example:

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

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

In the Interface Kit, BWindow objects name the current focus view as the preferred receiver. 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 receiver, 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 receiver. Note, however, that when a **NULL** receiver is passed to **PostMessage()**, that function designates the BLooper itself as the receiver. For example, if **PreferredReceiver()** 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 receiver, even though **PreferredReceiver()** returns **NULL**.

See also: SetTarget() in the BControl and BMenuItem classes of 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.

Quito 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 Quito 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: QuitRequested() in the BApplication class, **Quit()**

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()**

Thread()

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

Returns the thread that runs the message loop—or **B_ERROR** if **Run()** hasn't yet been called to spawn the thread and begin the loop.

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(3.1416, "pi");
```

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_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_24_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 also 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. 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***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: DetachCurrentMessage() in the BLooper class

~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(), 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 **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**
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 nine 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 the error code 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_UNEXPECTED_REPLY	Trying to send a reply to a message that isn't from a remote source
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(), 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**)**

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 **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**
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 "fiat" (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 reports the number of bytes that were allocated in 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.

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. 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(), 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 **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()

IsSenderWaiting() *see* Sender()

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()

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***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(), 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 **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()

Sender(), IsSenderWaiting()

thread_id **Sender(**void**)**

bool **IsSenderWaiting(**void**)**

These functions provide information about the sender of a remote message. **Sender()** identifies the main thread of the sending application. It returns –1 if the message was posted from within the application rather than sent from a remote source.

IsSenderWaiting() returns **TRUE** if the message sender is waiting for a reply, and **FALSE** if not. The sender can request and wait for a reply when calling BMessenger's **SendMessage()** function.

See also: SendMessage() in the BMessenger class, **SendReply()**

SendReply()

long **SendReply(**BMessage **message***)**
long **SendReply(**ulong *command***)**

Sends a reply message back to the sender of the BMessage. This function works only for BMessage objects that have been:

- Processed through a message loop and delivered to you, and
- Sent from a remote source.

The BMessage object you receive contains information identifying its source, so the reply *message* you construct and pass to **SendReply()** can be delivered to the application that initiated the communication.

The reply message can be delivered synchronously or asynchronously:

- It's delivered asynchronously if the message sender asked for a reply when calling BMessenger's **SendMessage()** function. **SendMessage()** waits for the reply to arrive before returning. 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.

- It's delivered asynchronously if the message sender isn't waiting for a reply. In this case, **SendMessage()** will have returned immediately after putting the message in the pipeline, so the reply can't be delivered to that function. It's delivered instead to the main thread of the sending application. The remote BApplication object dispatches it by calling **ReplyReceived()**-

If **SendReply()** is called when a reply is inappropriate—perhaps because one was already sent, or because the original message was posted from within the application—message is deleted and an error is recorded.

If you wish to delay sending a reply and keep the message beyond the time it's scheduled to be deleted, you may be able to detach the message from the message loop. See **DetachCurrentMessage()** in the BLooper class.

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 to **SendReply()** rather than a *message*, the function constructs the reply BMessage, initializes its what data member with the *command* constant, and sends it just like any other reply.

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: SendMessage() in the BMessenger class, **DetachCurrentMessage()** in the BLooper class, **Sender()**, **Error()**

Unflatten() *see* Flatten()

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.

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, hands it off to a BReceiver, 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 BReceiver'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, **MessageQueue()** in the BLooper class, **RemoveMouseMessages()** in the BWindow class of the Interface Kit

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 **~MessageQueue(**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***)** 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. 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: Lock() in the BLooper class

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***)**
void **RemoveMessage(**ulong *what***)**

Removes a particular *message* from the queue and deletes it—or removes and deletes all messages with a **what** data member matching the *what* argument passed.

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 another application. Each BMessenger can deliver messages to one, and only one, remote destination. The messages it sends are received in the other application's main thread of execution (by the remote BApplication object). They're likely to be dispatched by calling the BApplication object's **MessageReceived()** function.

An application can construct as many BMessengers as it needs to communicate with other applications. The only restriction is that the remote application must be running at the time the BMessenger is constructed.

An application that's designed to receive remote messages can facilitate the process in either of two ways:

- It can publish its signature and protocols for the messages it responds to. Other applications can use the signature to construct a BMessenger, then create BMessage objects in the correct format and have the BMessenger send them.
- It can provide the code for a proxy object based on the BMessenger class (or a proxy that incorporates a BMessenger object as a data member). Other applications would simply need to include the proxy and call its functions. The functions would send the required remote messages and wait for replies as appropriate.

A proxy hides the underlying messaging that takes place. For example, a proxy class that has a BMessenger as a **sender** data member might have functions that would work something like this:

```
bool MyProxy::GetSynonym(const char *word, char *buffer)
{
    BMessage *msg;
    BMessage *reply;
    bool result = FALSE;
    . . .
    msg = new BMessage(GET_SYNONYM);
    . . .
    msg->AddString("word", some_word);
    sender->SendMessage(msg, &reply);
```

```
    . . .
    if ( reply->what == SYNONYM_FOUND ) {
        strcpy(buffer, reply->FindString("synonym"));
        result = TRUE;
    }
    delete reply;
    return result;
}
```

A function like this keeps all the messaging details from the caller. Consequently, it presents a simplified interface to the services that the remote application provides.

Constructor and Destructor

BMessenger()

BMessenger(ulong *signature*, thread_id *thread* = –1**)**

Initializes the BMessenger so that it can send messages to an application identified by its *signature* or by its main *thread* of execution. If the *signature* passed is **NULL**, the application is identified by its main thread only. If the *thread* specified is –1, as it is by default, the application is identified by its signature only.

If both a real *signature* and a valid *thread* identifier are passed, they must match—the *thread* must belong to the application that the *signature* identifies. If more than one instance of the *signature* application happens to be running, the *thread* picks out a particular instance as the BMessenger's target. Without a valid *thread* argument, the constructor arbitrarily picks one of the instances.

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 *thread* identifier, it registers a **B_BAD_THREAD_ID** error. If the *thread* and the *signature* don't match, it registers a **B_MISMATCHED_VALUES** 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 application. 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).

The BRoster object can provide signature and thread information about possible destinations.

See also: the BRoster and BMessage classes, **Error()**

~BMessenger()

virtual **~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 thread arguments designated two different applications.
B_BAD_THREAD_ID	The constructor can't establish a connection to the specified thread, perhaps because there is no such thread or perhaps because the thread is not a main thread.
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.

SendMessage()

long **SendMessage(**BMessage **message***)**
long **SendMessage(**ulong *command***)**
long **SendMessage(**BMessage **message***, BMessage ***reply***)**
long **SendMessage(**ulong *command*, BMessage ***reply***)**

Sends a *message* to the destination application. The message 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 just 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));
```

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. If a reply isn't requested, **SendMessage()** returns immediately.

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 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 the main thread of an application to send a message to itself and expect a reply. The thread can't respond to the message and wait for a reply at the same time.)

See also: SendReply() in the BMessage class

Operators

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.

BReceiver

Derived from: public BObject

Declared in: <app/Receiver.h>

Overview

BReceiver objects are the primary handlers for messages received in a message loop, whether posted from within an application or sent from a remote source.

This is an abstract class. It declares just two functions, both of them hook functions that derived classes can override:

- The principal hook—**MessageReceived()**—is called by BLooper's **DispatchMessage()** function to pass an incoming message from the BLooper to the BReceiver. A derived class must implement this function to handle expected messages.
- The second hook function—**Looper()**—permits derived classes to tie a BReceiver to one particular thread, so that the only messages it can receive are ones posted to that thread.

A receiver can be designated for a message when BLooper's **PostMessage()** function is called to post it. The messaging mechanism eventually passes this receiver to **DispatchMessage()**, so that the message can be delivered to its designated destination.

Hook Functions

Looper()	Can be implemented to associate the BReceiver with a particular BLooper object.
MessageReceived()	Implemented to handle received messages.

Constructor and Destructor

BReceiver()

BReceiver(void**)**

Initializes the BReceiver by registering it with the messaging system.

~BReceiver()

virtual **~BReceiver(**void**)**

Removes the BReceiver's registration.

Member Functions

Looper()

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

Implemented by derived classes to associate the BReceiver with a particular BLooper object. By default, this function returns **NULL** to indicate that the BReceiver is not tied to any BLooper, but is free to receive messages posted to any thread.

However, if a derived class implements this function to return a particular BLooper object, messages can be targeted to the BReceiver only if they're posted to that BLooper. This discipline is imposed by BLooper's **PostMessage()** function.

In the Interface Kit, the BView class overrides **Looper()** so that a BView's **MessageReceived()** function will be called only for messages posted to the BWindow object where the view is located. The BLooper class overrides it so that a BLooper can't act as a message receiver for messages posted to any other thread but its own.

See also: PostMessage() in the BLooper class

MessageReceived()

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

Implemented by derived classes to respond to messages that BLooper objects dispatch to the BReceiver.

All messages are passed to BReceiver objects, but system messages are passed by calling a message-specific function, not **MessageReceived()**. These specific functions are defined by classes derived from BReceiver—especially BWindow and BView in the Interface Kit and BLooper and BApplication in this Kit. For example, the BApplication

class defines a **ReadyToRun()** function to respond to **B_READY_TO_RUN** messages, and the BView class defines a **KeyDown()** function to respond to **B_KEY_DOWN** messages.

Every system message is matched to a specific hook function. All other messages—those defined by applications rather than the kits—are dispatched by calling **MessageReceived()**.

The default (BReceiver) implementation of **MessageReceived()** does nothing; it's empty. You must implement it to handle all the various messages that might be dispatched to the BReceiver. It can distinguish between messages by the value recorded in the **what** data member of the BMessage object. For example:

```
void MyReceiver::MessageReceived(BMessage *message)
{
    switch ( message->what ) {
    case COMMAND_ONE:
        . . .
        break;
    case COMMAND_TWO:
        . . .
        break;
    case COMMAND_THREE:
        . . .
        break;
    default:
        MyBaseClass::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 that any messages handled by base versions of the function are not overlooked.

See also: PostMessage() and **DispatchMessage()** in the BLooper class

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 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 main threads. The main thread is the thread in which the application is launched and in which its **main()** function is executed.

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 main threads.

In one case, the BRoster also recognizes running applications by their team identifiers. Each application is a "team" of threads sharing an address space.

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(**thread_id *thread*, 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 main *thread*, 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 for which *thread* was created 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, **GetRunningAppInfo()** returns **B_BAD_THREAD_ID** if *thread* isn't, on the face of it, a valid thread identifier or if it doesn't identify the main thread of a running application. < **GetActiveAppInfo()** returns the same thing if there's no active 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. (This will be the same as the main *thread* 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, **GetAppInfo()** in the BApplication class, **Launch()**

GetThreadList()

void **GetThreadList(**BList ******threads***)** const
void **GetThreadList(**ulong *signature*, BList ******threads***)** const

Fills in the *threads* BList with the main **thread_id**s of running applications. The main thread is the thread that executes the application's **main()** function and that receives messages from remote sources.

Each item in the *threads* list will be of type **thread_id**. It must be cast to that type when retrieving it from the list, as follows:

```
thread_id who = (thread_id)threads->ItemAt(some index);
```

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 main **thread_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: ThreadFor(), the BMessenger constructor

IsRunning() *see* ThreadFor()

Launch()

long **Launch(**ulong *signature*, BMessage **message* = NULL,
thread_id **mainThread* = NULL**)**
long **Launch(**ulong *signature*, BList **messages*,
thread_id **mainThread* - NULL**)**
long **Launch(**ulong *signature*, long *argc*, char ***argv*,
thread_id **mainThread* = NULL**)**
long **Launch(record_ref** *executable*, BMessage **message* = NULL,
thread_id **mainThread* = NULL**)**
long **Launch(record_ref** *executable*, BList **messages*,
thread_id **mainThread* = NULL**)**
long **Launch(record_ref** *executable*, long *argc*, char ***argv*,
thread_id **mainThread* = 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 and will be received and responded to before the application is notified that it's ready to run. This is appropriate only for a message that 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.

Similarly, if a list of *messages* is specified, each one will be delivered on-launch. (Like the BMessages, the BList object will be deleted for you).

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, this function places the identifier for the newly launched application's main thread in the variable referred to by *mainThread* and returns **B_NO_ERROR**. If unsuccessful, it sets the *mainThread* 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()**

RemoveApplication(), RemoveTeam()

void **RemoveApplication(**thread_id *thread***)**

void **RemoveTeam(**team_id *team***)**

< These functions remove an application from the roster of running applications. **RemoveApplication()** removes the application identified by its main *thread*. **RemoveTeam()** removes the application corresponding to the *team* identifier. >

ThreadFor(), IsRunning()

thread_id **ThreadFor(**ulong *signature***)** const
thread_id **ThreadFor(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. **ThreadFor()** returns its main thread 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 main thread (to set up a BMessenger, for example). Therefore, it's most economical to simply call **ThreadFor()** 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, **ThreadFor()** arbitrarily picks one of the instances and returns its main thread.

See also: GetThreadList()

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 one defined type, **app_info**, 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 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: GetAppInfo() in the BRoster class, "Launch Constants" below

Application Messages

<app/AppDefs .h>

Enumerated constant	Enumerated constant
B_ACTIVATE	**B_ARGV_RECEIVED**
	B_REFS_RECEIVED
B_READY_TO_RUN	**B_PANEL_CLOSED**
B_APP_ACTIVATED	**B_VOLUME_MOUNTED**
	B_VOLUME_UNMOUNTED
B_ABOUT_REQUESTED	
B_QUIT_REQUESTED	**B_PULSE**

These constants represent the system messages that the Application Kit recognizes. See the introduction to this chapter and the BApplication class for details.

This isn't a complete list of all the message constants defined by the Application Kit—they're just the constants for system messages that the Kit expects the main thread to get and the BApplication object to handle. The Application Kit defines constants for all

system-defined messages, but handles just a few. The others are handled by other kits (especially the Interface Kit) and are documented in the chapters on those kits.

See also: "Application Messages" on page 13 of the chapter introduction

Cursor Constants

<app/AppDefs.h>

const unsigned char **B_HAND_CURSOR**[]

This constant contains all the data needed to set the cursor to the standard hand image.

See also: SetCursor() in the BApplication class

Data Type Codes

<app/AppDefs.h>

Enumerated constant	Enumerated constant
B_CHAR_TYPE	**B_ASCII_TYPE**
B_SHORT_TYPE	**B_RTF_TYPE**
B_LONG_TYPE	**B_STRING_TYPE**
B_UCHAR_TYPE	**B_MONOCHROME_1_BIT_TYPE**
B_USHORT_TYPE	**B_GRAYSCALE_8_BIT_TYPE**
B_BOOL_TYPE	**B_COLOR_8_BIT_TYPE**
B_ULONG_TYPE	**B_RGB_24_BIT_TYPE**
B_FLOAT_TYPE	**B_TIFF_TYPE**
B_DOUBLE_TYPE	**B_REF_TYPE**
B_POINTER_TYPE	**B_RECORD_TYPE**
B_OBJECT_TYPE	**B_TIME_TYPE**
B_POINT_TYPE	**B_MONEY_TYPE**
B_RECT_TYPE	**B_RAW_TYPE**
B_RGB_COLOR_TYPE	
B_PATTERN_TYPE	**B_ANY_TYPE**

These constants are used in a BMessage object to describe the type of data the message holds. See the BMessage class for more information on what they mean.

See also: "Type Codes" on page 63 of the BMessage class overview

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: GetAppInfo()** in the BRoster class, "Application Flags" above

Message Constants

<app/AppDefs .h>

Enumerated constant

B_NO_REPLY

B_CUT
B_COPY
B_PASTE

These constants mark messages that the system puts together, but that aren't dispatched like system messages.

- **B_NO_REPLY** initializes the **what** data member of a BMessage that'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 reply is expected and none has been sent.
- **B_CUT**, **B_COPY**, and **B_PASTE** initialize the **what** data members of BMessages that the Command-*x*, Command-*c*, and Command-*v* shortcuts generate and that BTextView objects (in the Interface Kit) respond to.

See also: SendReply() in the BMessage class, the BTextView class in the Interface Kit, "Application Messages" on page 98

Defined Types

app_info

<app/Roster.h>

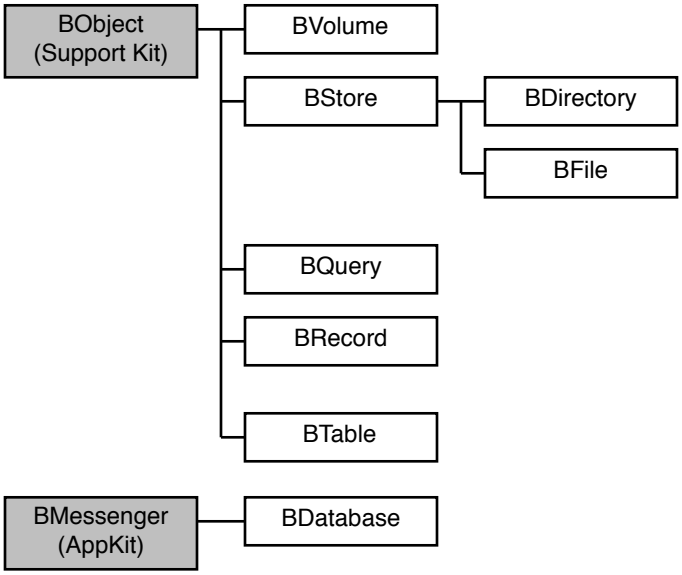
typedef struct {
ulong **signature**;
thread_id **thread**;
port_id **port**;
record_ref ref;
ulong **flags**;
} **app_info**

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: GetAppInfo() in the BRoster and BApplication classes

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. What this lacks in flexibility is made up for in retrieval power: Because the data is structured, you can locate a specific entry based on the values that are stored in the record.
- The file system classes (BStore, BDirectory, and BFile) provide a means for storing data in files. A file isn't (naturally) structured so its content is less restricted than that of a record.
- 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 by visiting the BVolume class description first, and then proceeding to the database or file system classes in the orders given above.

BDatabase

Derived from: public BMessenger

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. The database exists without the benefit of an accompanying object. In other words, databases are the real thing, BDatabases are merely representatives.

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*. Databases are contained within volumes. The relationship between databases and volumes is one-to-one: Each volume contains exactly one database; a single database is contained in exactly one volume.

  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. This subject (how to retrieve BVolume objects) is discussed in the BVolume class description. A cogent point, for the present topic, is that 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 *&List)
{
  BVolume this_vol;
  BDatabase *this_db;
  long counter = 0;

  while((this_vol = volume_at(counter++)))
  {
      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 system-wide 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. Database ID numbers appear, most commonly, as part of the **record_ref** structure. The **record_ref** structure is used to associate, or *refer*, a BFile or BDirectory object to an item in the file system.

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

Tables

When you "open" a database (by asking for the object 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()**. Examples are given in the function descriptions, below.

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.

Finding or creating tables through the BDatabase object are the only means by which you can obtain a BTable object. The implication of this is that a BTable object will always refer to a specific BDatabase object.

BDatabase as a BMessenger

BDatabase inherits from BMessenger to conveniently embody a connection to the Storage Server. The inheritance doesn't imply that you need to call any of the BMessenger functions yourself—not only do you not need to, you shouldn't even try. BDatabase's ancestry is an implementation detail that can be safely ignored.

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 object's table list.

See also: TableAt(), FindTable()

CreateTable()

BTable ***CreateTable(**char **table_name***)**
BTable ***CreateTable(**char **table_name*, char ***parent_name***)**
BTable ***CreateTable(**char **table_name*, BTable ***parent_table***)**

Creates a table in the database, names it *table_name*, 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 table designated by *parent_name* or *parent_table* argument.

The BDatabase doesn't check to make sure you aren't table name isn't unique: You can create a table with a given name even if that name identifies an existing table. To make sure that your name won't collide with that of an existing table, you should call **FindTable()** first:

```
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 table is created without a parent. Here, too, you could check first if you want to ensure that the table will be born of the desired progenitor:

```
if (a_db->FindTable("Phylum") == NULL &&
    a_db->FindTable("Kingdom") != NULL)
    a_table = a_db->CreateTable("Phylum", "Kingdom");
```

If the function can't create the table, it returns **NULL**.

You never explicitly delete a BTable object. Constructing and deleting BTables 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 invoke **Sync()** on the BDatabase.

See also: CreateTable(), Sync()

GetUnique()

long **GetUnique(**void**)**

Returns (directly, despite the **Get...()** component) an identifier that uniquely and persistently identifies the BDatabase's database.

Note: This function will be wedded with **ID()** and will take the latter's name in a future release.

ID()

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

Returns the database ID number that identifies the BDatabase's database. This number is unique among all databases that are *currently* available, and is only valid for as long as the database is mounted.

The value returned by this function is used, primarily, when you're communicating the identity of a database to some other application. It's also used as the **database** field of a **record_ref** structure; such structures are used to refer BFStore objects to items in the file system.

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**; don't use it.

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.

Sync()

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

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

- Reads the database's list of tables and re-installs it into the BDatabase's list.
- Makes sure that all committed data has been flushed from the Storage Server to the underlying storage media (in other words, write 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 flush committed data. Such data will (eventually) be written

to the disk as a matter of routine (within seconds, typically); **Sync()** simply anticipates the inevitable.

See also: BRecord::Commit()

TableAt()

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

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

See also: CountTables()

VolumeID()

inline 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 record ref is set.

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(), CountFSItems()

long **CountFiles(**void**)**
long **CountDirectories(**void**)**
long **CountFSItems(**void**)**

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

See also: GetFile(), GetDirectory()

CountFSItems() *see* CountFiles()

Create()

long **Create(**const char **newName*,
BStore **newItem*,
const char **tableName* = NULL,
file_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 *newItem* argument is modified (its ref 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 non-default tables. See the BStore class description on page 31, 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. You can test against the return value as the predicate to a loop that successively retrieves every file (or directory) contained in a directory:

```
/* Print the name and type of every item in a given
 * directory.
 */
```

```
void ShowNames(BDirectory *dir)
{
    long ktr = 0;
    BFile aFile;

    while (dir->GetFile(ktr++, &aFile) == B_NO_ERROR)
        printf("%s is a file\n", aFile.Name());

    ktr = 0;
    BDirectory aDir;

    while (dir->GetDirectory(ktr++, &aDir) == B_NO_ERROR)
        printf("%s is a directory\n", aDir.Name());
}
```

See also: Contains(), IsDirectory()

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 (data and resources) 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 information that live in directories. A file can live in only one directory at a time.

Locating and Creating Files

With one exception, the functions that BFile defines let you examine and manipulate a file's contents (the exception is **CopyTo()**, which copies a file to a specified directory). The functions that you use to locate, create, remove (and so on) files, and assign BFiles to refer to them are defined by the BStore class (from which BFile derives), and the BDirectory class. Listed below are the file-locating and -creating functions from these other classes that you should be aware of:

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.

Data and Resources

Every file has, potentially, two parts: A *data portion*, and a *resources portion*. There aren't any rules governing the content of these two parts; the distinction, from the BFile's perspective, is in how information is stored in either portion:

- Data (or "data in the data portion") is "flat." You add to the data portion by passing a buffer of data to the **Write()** function, but your addition isn't "marked" to distinguish it from the existing data—once you've added data to the data portion, it becomes part of a single, unstructured, vector of bytes. When you read from the data portion (through **Read()**), you wade into this vector and retrieve some amount of data (the amount is specified in the **Read()** call). There's no way to tell whether the data you've read was added in a single **Write()** call, or in a succession of calls.

- The resources portion contains data "entries" that are structured and identifiable—it's like a private database for the file. A file can contain any number of distinct resource entries. You add a resource through the **AddResource()** function. When you call the function, you supply a name by which the resource will be known. To retrieve the resource, you ask for it by name through the **FindResource()** function.

Theoretically, a file can have a data portion, a resources portion, both, or neither. In practice, however, every file has a data portion: When a file is created, a zero-length data portion is automatically created as a means for advertising the file's existence.

Opening and Closing Files

Before examining or manipulating either portion of a BFile, you must open that portion. Specifically, if you want to read or write data from a BFile, you first have to call its **OpenData()** function. To add, remove, or find resources, you first have to call **OpenResources()**. The portion remains open until the analogous "close" function (**CloseData()** or **CloseResources()**) is called.

As explained in the BStore class description, any number of BFile objects can refer to the same file. In addition, any number of BFile objects can read from or write to the data portion of the same file at the same time. Currently, there's no way to "lock" a file such that a particular object has exclusive access to the data portion.

By default, the resources portion of a file is read-only. Any number of BFile objects can read the resources portion of the same file. If you want to write the resources, you must declare as much in your **OpenResources()** call. Only one BFile may write the resources portion at a time.

If you destroy a BFile while it has a portion open, that portion is automatically closed.

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 calls to **CloseData()** and **CloseResources()**) before the object is destroyed.

See also: CloseData(), CloseResources()

Member Functions

AddResource()

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

Adds a resource to the BFile. For this function to have an effect, you must first open the resources portion for writing by calling **OpenResources()** with an argument of **TRUE**. The resource data is copied from the *data* buffer; the *dataLength* argument tells the function how many bytes of data to copy from the buffer.

The values that you supply for the *name* and *type* arguments are used to identify the resource after its been copied into the BFile. You would use these values to locate the resource in a subsequent **FindResource()** call (for example).

- The name is arbitrary and mustn't be longer than **B_OS_NAME_LENGTH** (32 characters).
- The type should be one of the data type constants—**B_STRING_TYPE**, **B_LONG_TYPE**, **B_OBJECT_TYPE**, and so on—defined in **app/AppDefs.h**. Note that the *type* argument isn't used to type-cast the data in die resource; it's simply a tag by which the resource is identified. (However, you may want to use the type value as a hint if you have to cast the data when you retrieve it.)

The combination of *name* and *type* needn't be unique within the BFile's resources: You can add (to the same BFile) any number of resources that have the same name and type. But retrieving identically-named and -typed resources is a bit messy—you have to use the

full-blown, indexing version of **FindResource()**. And distinguishing between them, once you've gotten them, is your own little hell.

If the BFile's resources portion isn't open—or if, for any other reason, the resource couldn't be added—the function fails and returns **B_ERROR**.

See also: OpenResources(), FindResource(), HasResource()

CloseData(), CloseResources()

long **CloseData(**void**)**
long **CloseResources(**void**)**

Closes the data or resources portion of the BFile. The object's BRecord is automatically committed to the database when you call either of these functions.

You should be aware that **CloseData()** and **CloseResources()** are (both) called automatically by the BFile destructor, and by BDirectory's **Remove()** function.

The BFile must previously have been opened through the analogous **OpenData()** or **OpenResources()** call. If the appropriate portion isn't open (or, more broadly, if the BFile's ref hasn't been set), these functions return **B_ERROR**; otherwise, **B_NO_ERROR** is returned.

See also: OpenData(), OpenResources()

CloseResources() *see* CloseData()

CopyTo()

long **CopyTo(**BDirectory **toDir*,
const char **newName*,
BFile **newFile*,
store_creation_hook **hookFunc* = NULL,
void **hookData* = 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 new 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.

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

- *hookFunc* is a pointer to a "store creation hook" function. This 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 *hookFunc*. The value returned by *hookFunc* is significant: If it returns **B_ERROR**, the copy operation is aborted; **B_NO_ERROR** lets it continue.
- *hookData* is a buffer of data that's passed as the second (and final) argument to *hookFunc*.

For more information on the use of the store creation hook mechanism, see the **BDirectory::Create()** function.

The **CopyTo()** function automatically commits the original object's BRecord.

The rules governing the ability to add the new file to the specified directory are the same as those that apply to creating a file in that directory. Again, see BDirectory's **Create()** function for more information.

The target BFile must be closed (both data and resources) 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: BDirectory::Create(), BStore::MoveTo()

CountResources()

long **CountResources(**const char **name*, ulong *type***)**

Returns the number of separate resource items the object contains. The resources portion of the file must be open (through a previous **OpenResources()** call) for this function to succeed. If the portion isn't open, the function returns **B_ERROR**.

See also: FindResource(), GetResourceInfo(), OpenResources()

DataSize(), ResourcesSize(), Size()

long **DataSize(void)**
long **ResourcesSize(**void**)**
long **Size(**void**)**

Returns the size of the file's data portion, resources portion, or their combination, in bytes. You don't have to open the data and/or resources prior to calling these functions.

The functions return **B_ERROR** if the BFile's ref hasn't been set, or if the BFile's record has disappeared. (The latter can happen if the file has been removed.)

See also: SetDataSize()

FindResource()

void ***FindResource(**const char **name*,
ulong *type*,
long **length***)*
void ***FindResource(**const char **name*,
ulong *type*,
long *index*,
long **length***)*

Returns a pointer to a specific resource from the BFile. The target resource is identified by *name* and *type*. The first version of the function always returns the first resource that matches these arguments. The second version of the function lets you ask for the *index*'th matching resource (the index is zero-based). This is necessary if you have more than one resource with the same name and type (an ambiguity that isn't disallowed), or if you're searching on the name only (by supplying **B_ALL_TYPES** as the type argument). The final argument, *length*, returns the size of the resource in bytes.

The pointer that the function returns points to data that's owned by the file; if you need to cache the resource, you should make your own copy of the pointed-to data.

These functions return **NULL** and set the **Error()** variable to **B_ERROR** if the resources portion of the BFile isn't open or if the designated resource wasn't found.

See also: AddResource(), GetResourceInfo(), HasResource()

GetResourceInfo()

bool **GetResourceInfo(**char **byName*,
long *andIndex*,
ulong **type*,
long **count* = NULL**)**
bool **GetResourceInfo(**ulong *byType*,
long *andIndex*,
char **name*,
long **count* = NULL**)**
bool **GetResourceInfo(**long *byIndex*,
char **name*,
ulong **type*,
long **count* = NULL**)**

These functions return information about a particular resource. The first one or two argument(s) locate the resource; the final arguments return the information by reference.

- The first version locates the *andIndex*'th resource that has the name *byName*. The resource's type is returned in *type*, and the number of resources that share this resource's name and type is returned in *count*.
- The second version locates the *andIndex*'th resource of type *byType*. The resource's name is returned in *name*, and the shared-count in *count*.
- The final version gets information for the *byIndex*'th resource—in other words, *all* resources in the BFile are considered. The name, type, and shared-count are returned in the final arguments.

The functions return **TRUE** if the designated resource was found; otherwise it returns **FALSE**.

See also: FindResource(), HasResource()

GetTypeAndApp() *see* SetTypeAndApp()

HasResource()

bool **HasResource(**const char **name*,
ulong *type*,
long *index* = 0**)**

Looks for the resource identified by *name*, *type*, and (optionally) *index*. Returns **TRUE** if the resource is found, otherwise returns **FALSE**. The resources portion of the BFile must already be open for this function to work properly (the function returns **FALSE** if the portion isn't open).

See also: FindResource(), GetResourceInfo()

OpenData(), CloseData()

long **OpenData(**void**)**
long **CloseData(**void**)**

These functions open and close the data portion of the BFile. **OpenData()** gives the BFile object access to the "normal," or non-resource, data in the underlying file, allowing the object to read and write this data (through the **Read()** and **Write()** functions). The data portion remains open until **CloseData()** is called.

Unsurprisingly, the **Read()**, **Write()**, and **Seek()** functions require that the BFile's data portion be open (**Seek()** sets the "data pointer" position). Conversely, **CopyTo()** and **SetTypeAndApp()** fail if the data portion isn't closed. The BFile destructor, the **SetRef()** function, and BDirectory's **Remove()** function also expect the data portion to be closed, but they don't fail if it's open—they close the data portion automatically.

If the BFile's ref hasn't been set, if its record has disappeared, or if, for any other reason, the data portion couldn't be opened, **OpenData()** returns **B_ERROR**. **CloseData()** returns **B_ERROR** if the data portion isn't open (by that BFile). Upon success, both functions return **B_NO_ERROR**.

Note that access to the data portion isn't affect by the state of the resources portion of the same file. For example, a given BFile object can open the data portion while a separate BFile object (that points to the same file) holds the resources portion open.

See also: Read(), Write(), Seek(), OpenResources()

OpenResources(), CloseResources()

long **OpenResources(**bool *forWriting* = FALSE**)**
long **CloseResources(**void**)**

These functions open and close the resources portion of the BFile. If *forWriting* is **FALSE** (the default), **OpenResources()** grants read-only access to the resources. Any number of BFile objects can read the same file's resources at the same time. If *forWriting* is **TRUE**, exclusive, read/write access is granted to the BFile. The resources portion remains open until **CloseResources()** is called.

Most of BFile's resource-accessing functions (**AddResource()**, **GetResourceInfo()**, **RemoveResource()**, and the others) require that the BFile's resources portion be open. Conversely, **CopyTo()** and **SetTypeAndApp()** fail if the resources portion is open. The BFile destructor, the **SetRef()** function, and BDirectory's **Remove()** function also expect the resources portion to be closed, but they don't fail if it's open—they close the resources portion automatically.

If the BFile's ref hasn't been set, if its record has disappeared or if, for any other reason, the resources portion couldn't be opened, **OpenResources()** returns **B_ERROR**. **CloseResources()** returns **B_ERROR** if the resources portion isn't open (by that BFile). Upon success, both functions return **B_NO_ERROR**.

Access to the resources portion isn't affect by the state of the data portion of the same file. For example, a given BFile object can open the resources portion while a separate BFile object (that points to the same file) holds the data portion open.

See also: AddResource(), FindResource(), HasResource(), OpenData()

Read()

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

Copies (at most) *dataLength* bytes of data from the data portion of the BFile 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 data portion; 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 data portion of the BFile must already be open. If the data isn't open, or if, for any other reason, the portion couldn't be read, the function returns **B_ERROR**.

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

RemoveResource()

void **RemoveResource(**const char ***name*,
ulong *type*,
long *index* = 0**)**

Removes the resource identified by the arguments. The resources portion must be open for this function to work. If the function fails (for whatever reason), it sets the **Error()** code to **B_ERROR**.

See also: AddResource(), HasResource(), GetResourceInfo()

ReplaceResource()

void **ReplaceResource(**const char ***name*,
ulong *type*,
void ***data*,
long *dataLength***)**
void **ReplaceResource(**const char ***name*,
ulong *type*,
long *index*,
void ***data*,
long *dataLength***)**

Finds the resource identified by *name* and *type* (and, in the second version, *index*), throws away the existing data for that resource and installs, in its place, data that's copied from the *data* buffer. The final argument gives the number of bytes to copy from the buffer. The name, type, and index of the resource aren't changed.

The resources portion of this BFile must be open for this function to work. If the resource data couldn't be replaced—for example, if the resource wasn't found, or (for another) the resources portion wasn't open—**ReplaceResource()** sets the **Error()** code to **B_ERROR** and, silently and unconsummated, expires.

See also: AddResource(), RemoveResource(), HasResource()

ResourcesSize() *see* DataSize()

Seek()

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

Relocates the BFile's data pointer (its pointer into the data portion of the file). 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 values (which you pass as the value of *relativeTo*):

- 0 represents the beginning of the file.
- 1 represents the pointer's current location.
- 2 represents the end of the file.

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

```
aFile->Seek(5, 1)
```

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, 2)
```

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->DataSize() * 2, 0)
```

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, 1) ;
```

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's data portion must already be open (**B_ERROR** is returned if the portion isn't open). Moving the data pointer doesn't affect access to the resources portion of the BFile.

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()

SetDataSize()

long **SetDataSize(**long *sizeInBytes***)**

Sets the length, in bytes, of the BFile's data portion to *sizeInBytes*. The data portion must be open for this function to succeed. The function returns **B_ERROR** if the data portion isn't open, or if, for any other reason, the file couldn't be set to the given size. Otherwise, it returns **B_NO_ERROR**.

See also: DataSize()

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 data and resource portions of the BFile 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* constant 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 look at the 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.

Both **SetTypeAndApp()** and **GetTypeAndApp()** expect the BFile's resources portion to be closed. They 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() *see* DataSize()

Write()

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

Copies *length* bytes from the *data* buffer into the data portion of the BFile. 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 data portion is extended, 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's data portion 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 a database to identify records whose values fall within specified ranges. 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. You supply tables as BTable objects through the **AddTable()** or **AddTree()** function.
- The predicate is a logical test that (typically) compares the value for a particular field (in a record) to a constant value. You can also compare a 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.

For example, 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->PushArg(12);
teenOrMore->PushOp(B_GT);
```

Details of the table and predicate specifications are examined in later sections.

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 record ID) are returned to the BQuery and placed in the BQuery's record 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:

```
/* Make BRecord objects for the BQuery's record IDs and place
 * them (the BRecords) in a BList.
 */
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);
}
```

The Tables

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.

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 arguments 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->PushArg(12);
teenOrMore->PushOp(B_GT);
```

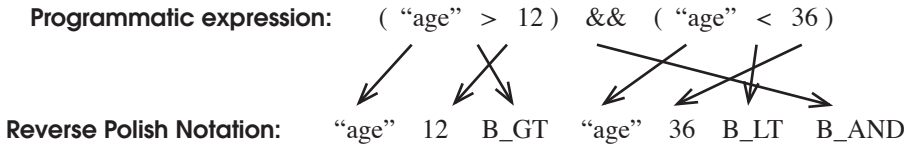
The query operators that you can use are represented by constants defined by the Storage Kit:

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)

Complex Predicates

You can create more complex predicates by using the conjunction operators **B_AND** and **B_OR**. As with comparison operators, a conjunction operator is pushed after its arguments; but with the conjunctions, the two arguments 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, mapped to its reverse Polish notation, looks like this:



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

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

```
teenOrMore->PushOp(B_GT);

teenOrMore->PushField("age");
teenOrMore->PushArg(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.

Fields and Constants

As implied by the examples, the **PushField()** function pushes fields, and **PushArg()** pushes constant values onto the predicate stack. Constants are straightforward; fields need a bit more explanation.

A field in a predicate acts as a variable: It represents the value that a record holds for that field. However, keep in mind that a BQuery can have more than one table in its table list, and any of these tables can have identically named fields. This can be a problem. For example, if you use a field named “size” in a predicate, and then search through two different tables that both have “size” fields, there’s no way to tell which table will supply the “size” value.

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 in sync 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:

```
/* Implementation of MessageReceived() for a subclass
 * of BApplication (called MyApp). The implementation
 * recognizes Storage Server query-updating messages.
 * To be polite, you should include Query.h to get the
 * update_query() declaration
 */

#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 subclass BQuery 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;
}
```

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***)**

Adds the given record (identified by **record_id**) to the BQuery's record list. 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.

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.

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()

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 the class description, above. You never call this function directly, but you can override it in a BQuery 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.

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.

PushArg(), PushDate(), PushField(), PushOp()

void **PushArg(**long *num_value***)**

void **PushArg(**char **string***)**

void **PushDate(**double *time***)**

void **PushField(**char **field_name***)**

void **PushOp(**query_op *operator***)**

These functions push elements onto the predicate stack:

- The **PushArg()** functions take constant values.
- **PushField()** takes the name of a table field.
- **PushDate()** takes a double value that specifies a time measured in microseconds since January 1, 1970.
- **PushOp()** takes one of the query operators listed below.

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 class description, above. Briefly, it's based on the "reverse Polish notation" convention in which the two arguments to an operation are pushed first, followed by the operator. The result of an operation can be used as one of the arguments in a subsequent operation.

See also: PrintToStream()

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 (single) 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()

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

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 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** (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 **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 ape record example, we begin to put data in the new BRecord by setting data for the name and cage number 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 field key.
 * Note that the SetString() function takes the length of
 * the string as its third argument.  The function copies the
 * string, so it needs to know how much data to copy.
 */
field_key name_key = employee_table->FieldKey("name");
employee_record->SetString(name_key, "Mingo, Lon",
                    strlen("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 that case, the fields will only be distinguishable be field key (which, remember, is based on the field's name and its type).

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 data 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 have the same record ID value (they can return the same **record_id** value from the **ID()** function), 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. In other words, 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 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 the employee record we created, 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.

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 five data types that a field can take (*ergo* **FindLong()**, **FindRaw()**, **FindRecordID()**, **FindString()**, and **FindTime()**). Each typed function has two version so you can designate the field by field key or by name.

Keep in mind that when you examine a BRecord's data, you're looking at a copy of the data that exists in the actual record. Changes to the record aren't automatically reflected in your BRecord object ("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 will be lost.

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()**.

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(), FindRef(), 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()...

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).

SetDouble(), SetLong(), SetRaw(), SetRef(), 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.

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",6);
a_record->SetString("name", "Decca")
...
a_record->Commit();
a_record->Unlock();
```

See also: FindLong()

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).

BStore

Derived from: public BObject

Declared in: <storage/Store.h>

Overview

BStore is an abstract class that defines common functionality for its two subclasses, BDirectory and BFile. You never construct direct instances of BStore, nor does the Storage Kit "deliver" such objects to your application (as it does BFiles and BDirectories). Furthermore, it's useless to derive your own BStore class: The Kit won't recognize your class, and so won't be able to deliver, to your application, objects constructed from it (this proscription applies to BFile and BDirectory derivations as well).

Note: Throughout this class description, the term "file" is 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 practically 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 currently available 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 idea 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 most 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 79.

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:

- BVolume's **GetRootDirectory()** sets the ref for the BDirectory argument that you pass to the function. The function causes the BDirectory to refer to the BVolume's *root directory*; this is the "starting-point" directory in the volume's file system.
- 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 immediately contains the file that the object refers to.

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()**.

Altering the File System

The Storage Kit provides a set of functions that alter the file system by creating, moving, and removing files. These functions, listed below, also set the refs of the target objects, although, in this context, setting the ref is a lesser concern:

- 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.

Unfortunately, you can't retrieve a BStore's ref directly. 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

Although you can't create BStore-derived classes, it is possible to "customize" your files by, instead, 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."

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 "Sound File" table is defined and used to create a new file:

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

SoundTable->AddLongField("Duration");
SoundTable->AddLongField("Format");
SoundTable->AddStringField("Description");

/* Create a new "sound file."  The BDirectory object aDir
 * is assumed to exist.
 */
BFile mySoundFile;
aDir->Create("Bug.snd", &mySoundFile, "Sound Table");
```

Tables, remember, are defined for specific databases; the **SoundTable** 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 *mySoundRec = mySoundFile->Record();

mySoundRec->SetLong("Duration", 2565);
mySoundRec->SetLong("Format", 1);
mySoundRec->SetString("Description", "Bug squish");
mySoundRec->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 55 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—**Name()**, 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 as many of the fields as 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. For instance, this would be the natural order of things given 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 file creation hook can't be declared as part of a class. Also, although **store_creation_hook** is declared (in **storage/Store.h**) as a **typed**, the declaration is intended to be seen for its protocol only: You can't declare a function as a **store_creation_hook** type.

The file creation hook is called just after the file's record has been 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 file creation hook thus:

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

     mySoundRec->SetLong("Duration", 2565);
     mySoundRec->SetLong("Format", 1);
```

```
        mySoundRec->SetString("Description", "Bug squish");
        return B_NO_ERROR;
}
```

Note that you *don't* commit record changes that you make in a file 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 file creation is aborted (by the **Create()** function).

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

```
aDir->Create("Bug.snd", &mySoundFile, "Sound Table",
             soundFileHook);
```

Other Hook Providers

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

- BFile's **CopyTo()** function.
- BDirectory's **Create()**.
- BStore's **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 a sound description string as hook data:

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

        mySoundRec->SetLong("Duration", 2565);
        mySoundRec->SetLong("Format", 1);
        mySoundRec->SetString("Description", (char *)hookData);
        return TRUE;
}
```

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

```
aDir->Create("Bug.snd", &mySoundFile, "Sound Table",
             soundFileHook, (void *)"Bug squish");
```

Hook Function Rules

- 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 an implementation of a 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

CreationDate(), ModificationDate()

long **CreationDate(**void**)**
long **ModificationDate(**void**)**

Returns the time the item was created or last modified, measured in seconds since January 1, 1970. If the object is invalid, this function returns **B_ERROR**. To convert the time value to a string, you can use standard-C function **strftime()** or **ctime()** (as declared in **time.h**).

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.

GetParent()

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

Sets the argument's ref to represent the directory that contains this item (you must allocate the argument before you pass it). 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**.

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 55 of the introduction to this class.

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

Name()

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

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

See also: SetName()

Record()

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

Returns a BRecord object that represents the record in the database that holds information for this file system item. 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.

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 53 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 an item 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: Name(), 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 object'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 51 of the introduction to this class.

Volume()

BVolume ***Volume(**void**)**

Returns the BVolume object that represents the volume in which this item is stored.

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 a field that represents **time_t** 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.

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—a file icon, for example—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 63 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 time (double) data.

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

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:

RAW_TYPE
LONG_TYPE
RECORD_TYPE
STRING_TYPE
TIME_TYPE

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:

LONG_TYPE
RAW_TYPE
RECORD_TYPE
STRING_TYPE
TIME_TYPE

The *flags* value will either be **B_INDEXED_FIELD** or 0. (See "Field Keys" on page 63 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 declared 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. The function returns **NULL** if the index is out-of-bounds. 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++)))
   {
      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 (but see the note marked "Important" on page 73). 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];

    old_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 Pathnames

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

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

/volumeName/directoryName/directoryName/...

The volume name itself *doesn't* include the surrounding slashes.

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()

int **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**.

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 set 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 returns **NULL**.

volume_for_database()

BVolume **volume_for_database(**BDatabase **db***db***)**

Returns the BVolume that corresponds to the volume that contains the database identified by the argument.

If *db* is invalid, the function returns **NULL**

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 executables for the kernel, app server, net server, and so on, that are currently running.

See also: the BVolume class description

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 across all available databases. They're not, however, persistently unique—you can't cache a database ID to use again tomorrow.

If *databaseID* is invalid—if it doesn't identify an available database—the function returns **NULL**.

See also: the BDatabase class description

does_ref_conform ()

<storage/Record.h>

bool **does_ref_conform(**record_ref *ref*, 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 within an implementation of

BApplication's **RefsReceived()** hook function. 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 "Files" table is used for files.
- The "Folders" 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 **is_ref_of_type()** 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 85.

Here we create a **RefsReceived()** function that looks for file-representing refs (only) and creates a BFile for each:

```
void MyApp::RefsReceived(BMessage *msg)
{
    record_ref *theRef;
    BFile *theFile;
    long counter;
    long countFound;
    ulong typeFound;

    /* First we count the refs in the message. */
    if (!msg->GetInfo("refs", &typeFound, &countFound))
        return;
    if (countFound < 1)
        return;

    /* Loop over the refs. */
    for (counter = 0; counter < countFound; counter++)
    {
        theRef = a_message->FindRef("refs", counter)

        /* Find the refs that represent files. */
        if (does_ref_conform(theRef, "File"))
        {
            theFile = new BFile();
            theFile->SetRef(*theRef);
            /* Do something with the BFile here */
        }
    }
}
```

If you've been paying attention, you'll probably have conjectured that you can perform the "does conform" test through clever manipulation of the BRecord constructor and BTable's **HasAncestor()** function. Indeed; but this function conveniently abstracts all that database nonsense, to the approbation of a database-leery public.

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: the BQuery class description

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 returns **NULL**.

See also: the BVolume class description

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 returns **NULL**.

Constants

Live Query Messages

<storage/Query.h>

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: MessageReceived() in the BQuery class

query_op Constants

<storage/Query.h>

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.

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.

"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"	LONG_TYPE
"openOnMount"	LONG_TYPE
"inited"	LONG_TYPE
"invisible"	LONG_TYPE
"dockInited"	LONG_TYPE
"dockX"	LONG_TYPE
"dockY"	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
“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

"Person"

Parent table: "BrowserItem"

Field Name	Field Type
"Company"	STRING_TYPE
"Address"	STRING_TYPE
"Phone"	STRING_TYPE
"City"	STRING_TYPE
"State"	STRING_TYPE
"Zip"	LONG_TYPE
"Account #"	STRING_TYPE
"Portfolio Value"	LONG_TYPE
"Position"	RECORD_TYPE
"Fax"	STRING_TYPE
"Comments"	STRING TYPE

"Position"

Parent table: "BrowserItem"

Field Name	Field Type
"Division"	STRING_TYPE
"Salary"	STRING_TYPE

"Quote"

Parent table: "BrowserItem"

Field Name	Field Type
"Stock Price"	LONG_TYPE
"52 Week High"	LONG_TYPE
"52 Week Low"	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

"Tool"

Parent table: "BrowserItem"

Field Name	Field Type
"Description"	STRING_TYPE

"Icon"

Parent table: (none)

Field Name	Field Type
"creator"	LONG_TYPE
"type"	LONG_TYPE
"largeBits"	RAW_TYPE
"smallBits"	RAW_TYPE

"DisplayTemplate"

Parent table: (none)

Field Name	Field Type
"dbType"	LONG_TYPE
"windowType"	STRING_TYPE
"numFields"	LONG_TYPE
"template Array"	RAW_TYPE

"Dock"

Parent table: (none)

Field Name	Field Type
"dbType"	LONG_TYPE
"width"	LONG_TYPE
"mode"	LONG_TYPE
"bigMode"	LONG_TYPE
"miniMode"	LONG_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:

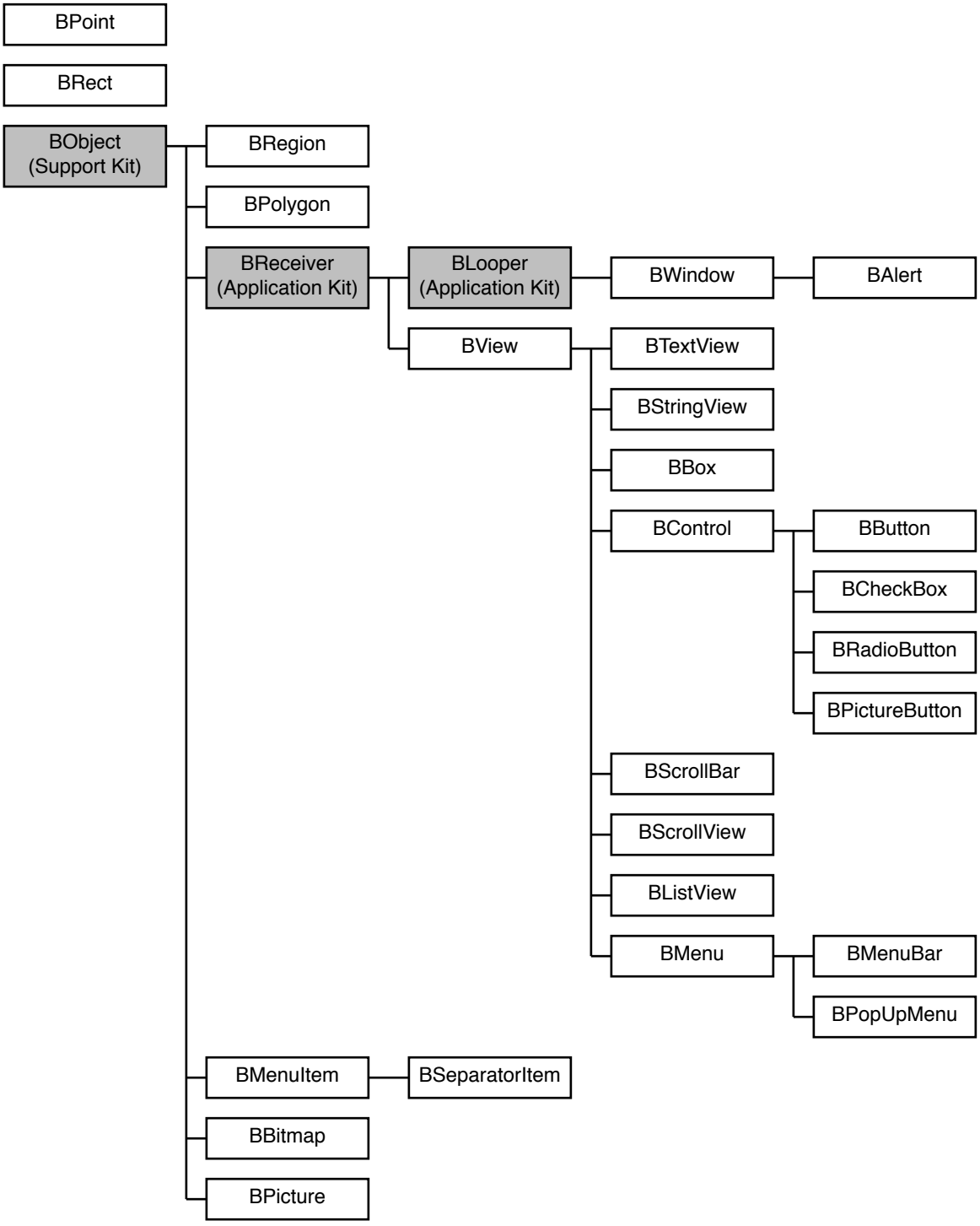
- The ‘ICON’ resource that’s named “BAPP” holds the icon that’s displayed for the application.
- The ‘ICON’ that takes, as a name, the application’s signature converted to a string holds the data that’s displayed for documents created by the application.

‘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 Receiver

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 BReceiver, so BView objects are eligible to handle messages dispatched by the BWindow.

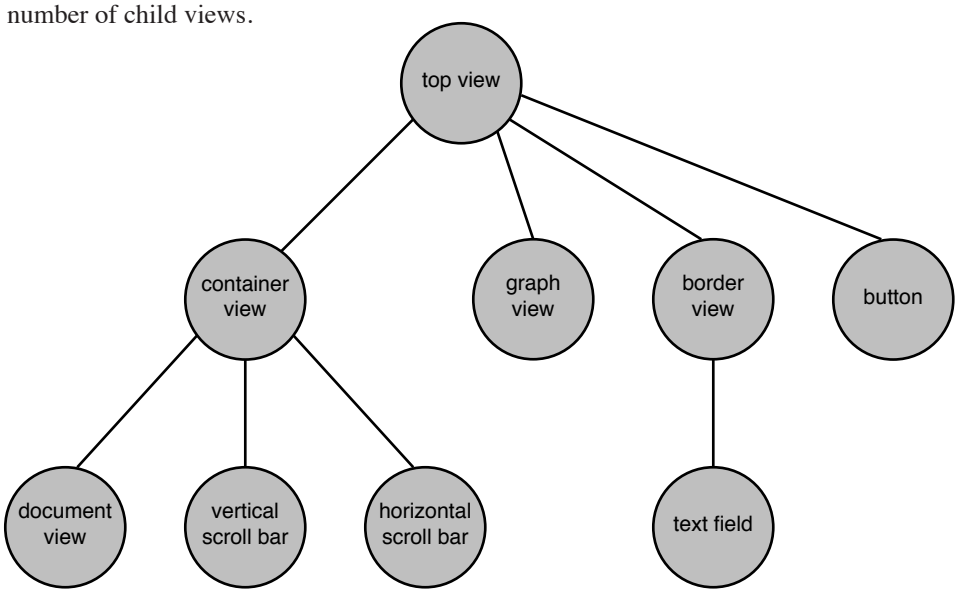
Just as classes derived from BView implement **Draw()** functions to draw within the view rectangle, they also implement the hook functions that respond to interface messages. These functions are discussed later, under "Hook Functions for Interface Messages" on page 43.

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.

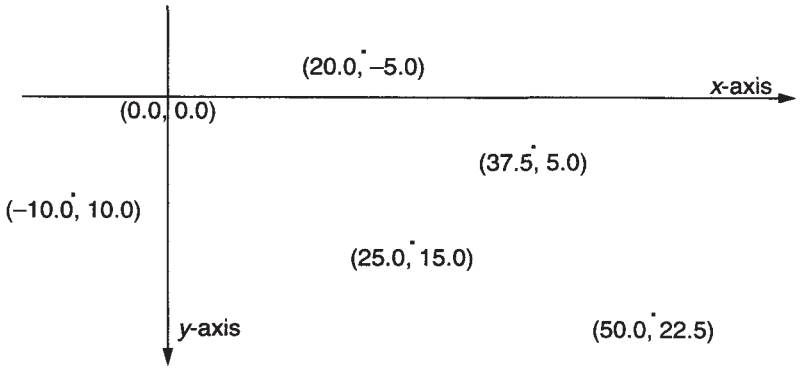
Similarly, it's indeterminate which view will be associated with mouse events in the area the siblings share. It may be one view or it may be the other, and it won't necessarily be the one that drew the image the user sees.

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:

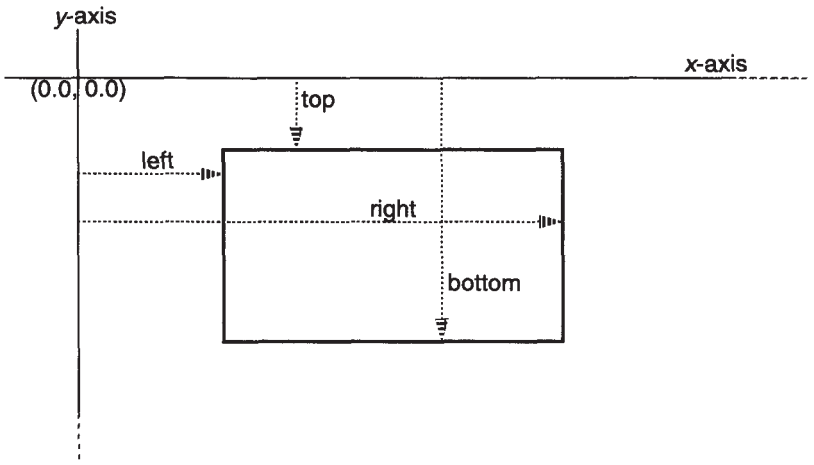
- The global or *screen coordinate system* has its origin, (0.0, 0.0), at the left top corner of the screen. It's used for positioning windows on-screen, < for arranging multiple screens connected to the same machine, > and for comparing coordinate values that weren't originally stated in a common coordinate 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

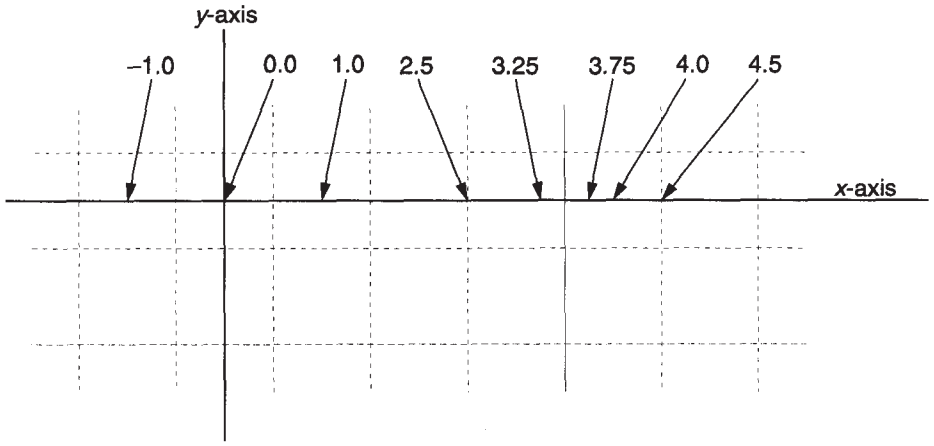
To map coordinate locations to device-specific pixels, you need to know only two things:

- 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.

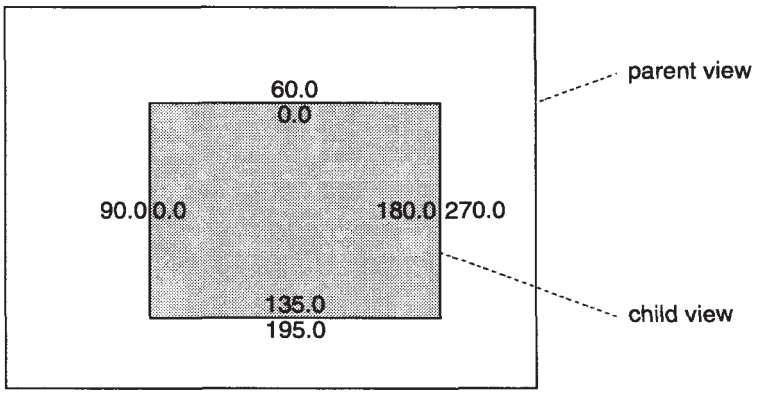
Since each view retains its own internal coordinate system no matter who its parent is, where it's located within the parent, or where the parent is located, a BView's drawing and message-handling code doesn't need to be concerned about anything exterior to itself. To do its work, a BView need look no 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:



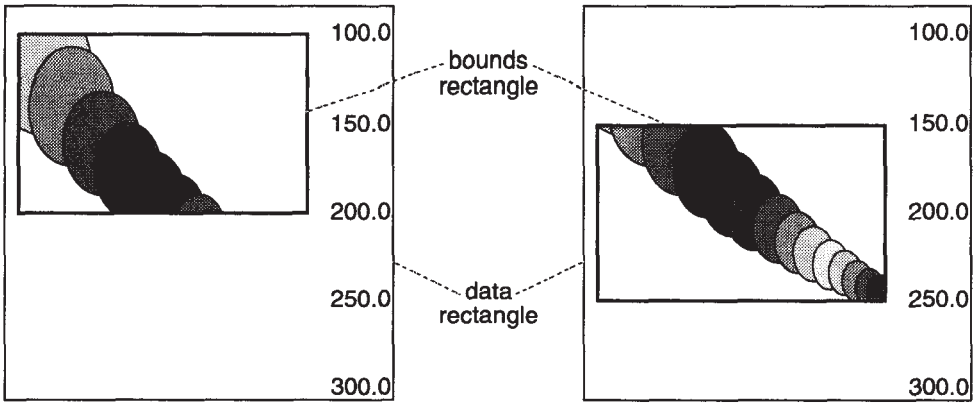
When a view moves to a new location in its parent, its frame rectangle changes but not its bounds rectangle. When a view scrolls its contents, its bounds rectangle changes, but not its frame. The frame rectangle positions the view in the world outside; the bounds rectangle positions the contents inside 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.

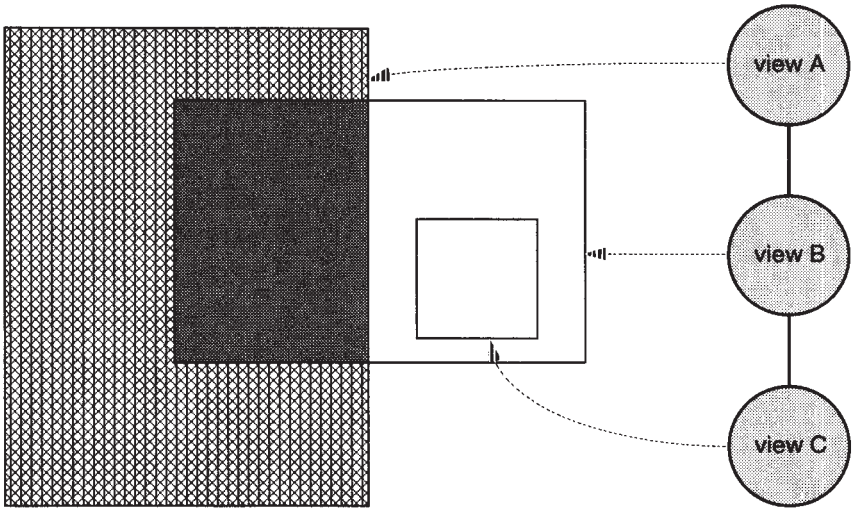
Clipping Region

The Application Server clips the images that a BView produces to the region where it's permitted to draw.

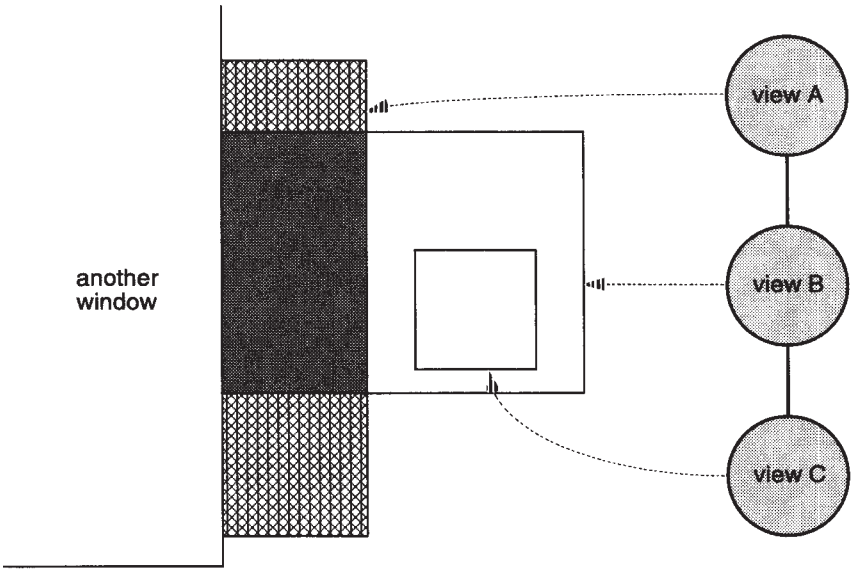
This region is never any larger than the view's bounds rectangle; a view cannot draw outside its bounds. Furthermore, since a child is considered part of its parent, a view can't draw outside the bounds rectangle of its parent either—or, for that matter, outside the bounds rectangle of any ancestor view. 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; it's not one of the BView's graphics parameters.

By default, the view color is white. You can assign a different color to a view by calling BView's **SetViewColor()** function. Every view can have its own background color.

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 coordinate system and the clipping region are two 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.)
- 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" (9-point bitmap font, no rotation, 90° shear)
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**)
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 30, 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 and held perpendicular to it. 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 24-bit values with separate red, green, and blue components. Although there may be limitations on the colors that can be rendered on-screen, there are none 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, and so on). Four 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. <This color space is currently not implemented. >
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_24_BIT	Four components of data per pixel—red, green, blue, and alpha, arranged in that order—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 of an **B_RGB_24_BIT** color 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.

The format of a **rgb_color** value exactly matches that of the **B_RGB_24_BIT** color space—in other words, the high and low colors are specified as **B_RGB_24_BIT** colors. However, onscreen, all colors are rendered in the **B_COLOR_8_BIT** color space. Specified 24-bit colors are converted to 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.

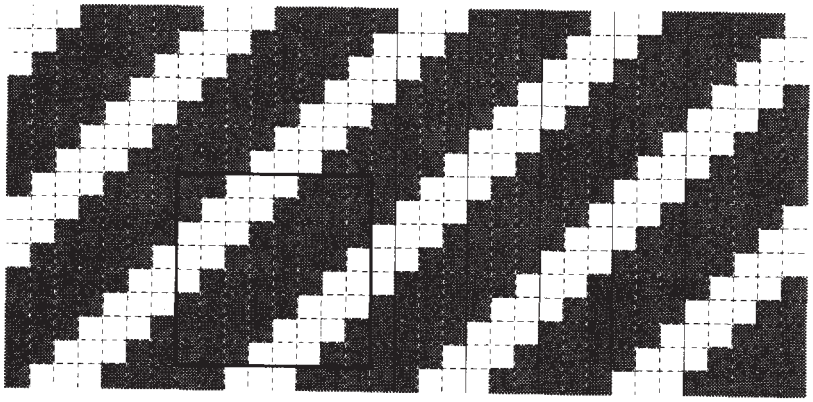
By combining the low color with the high color, patterns can produce dithered colors that lie somewhere between two hues in the **B_COLOR_8_BIT** color space. Patterns also permit

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:

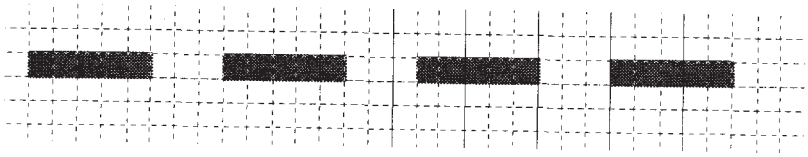
```
pattern stripes = { 0xc7, 0x8f, 0x1f, 0x3e,
                    0x7c, 0xf8, 0xf1, 0xe3 };
```

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_24_BIT** color space, a pixel assigned the **B_TRANSPARENT_24_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_24_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_SUBSTRACT**, 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_COLOR_24_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 **B_OP_ADD**, **B_OP_SUBTRACT**, and **B_OP_BLEND**, **B_OP_MIN** and **B_OP_MAX** work only for color images.

Views and the Server

Just as windows lead a dual life—as on-screen entities provided by the Application Server and as BWindow objects in the application—so too do views. 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 means that some functions can't operate on unattached BViews. Three kinds of functions are included in this group:

- Most obvious among them are the drawing functions—**DrawBitmap()**, **FillRect()**, **StrokeLine()**, and so on. A BView can't draw unless it's in a window.
- Also included are functions that set and return graphics parameters—such as **DrawingMode()**, **SetFontSize()**, **ScrollTo()**, and **SetHighColor()**. 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. It won't work, for example, to create a BView, set its low color, and then attach it to a window. The low color can be set only after the BView belongs to the window.
- The group similarly includes 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. These functions require information that an unattached BView can't provide.

Because of these restrictions, you may find it impossible 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. **AttachedToWindow()** can be implemented to set graphics parameters and to take care of any other final initialization that's required.

When a BView is removed from a window, it loses its graphics environment. Thus if a BView is moved to a different window or it changes its position in the view hierarchy of the same window, its graphics parameters must be reset. **AttachedToWindow()** is called to reset them.

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 added or 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.

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. The device-specific value measures 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 × ( 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.

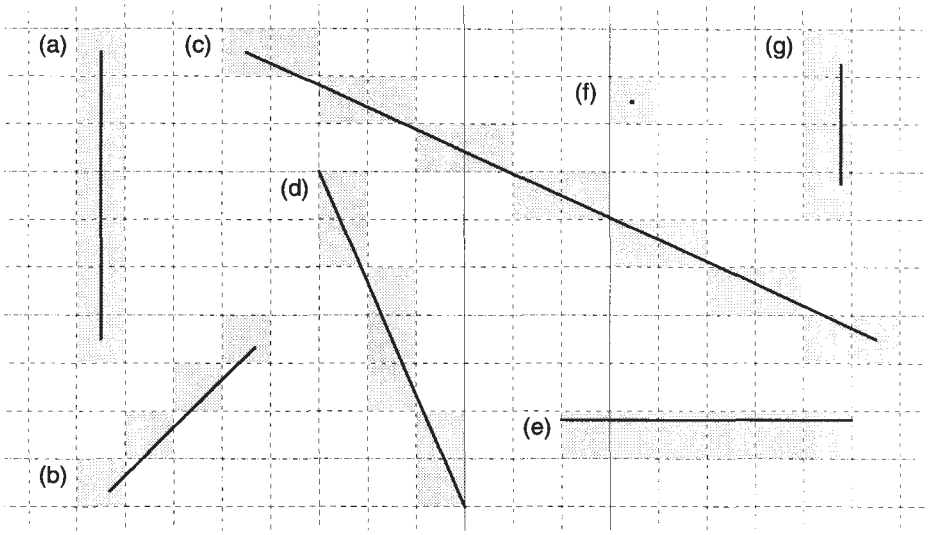
To describe where lines and shapes fall on the pixel grid, this section mostly talks about pixel units rather than coordinate units. The accompanying illustrations magnify the grid so that pixel boundaries are clear. As a consequence, they can show only very short lines and small shapes. By blowing up 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 that the pen will be a one-pixel square on all devices.

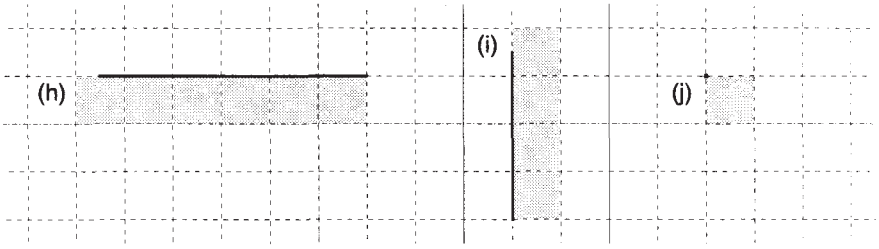
A one-pixel pen follows the path of the line it strokes and makes the line exactly one pixel thick. If the line is more vertical than horizontal, only one pixel in each row is used to render the line. If the line is more horizontal than vertical, only one pixel in each column is used.

Some illustrations of one-pixel thick lines are given below. The broken lines show the separation of the display surface into pixels:



The first thing to notice about this illustration is that 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 lines (d) and (e), for example, the pixels at the boundary aren't colored unless the path crosses into the pixel. The pen touches the fewest possible number of pixels.

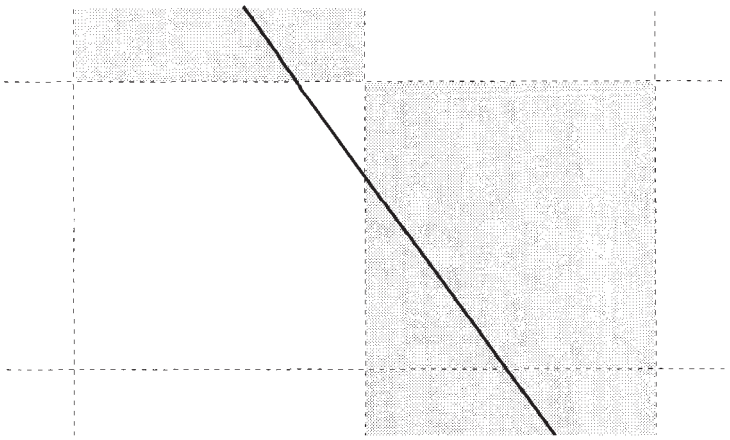
It's possible for a line path not to enter any pixels, but to lie entirely on the boundaries between pixels. Such a line is not invisible. A horizontal path between pixels colors the pixel row beneath it. A vertical path between pixels colors the pixel column to its right. A line path that reduces to a single point lying on the corner of four pixels colors the pixel at its lower right. The orientation of the pen is always toward the bottom and the right.



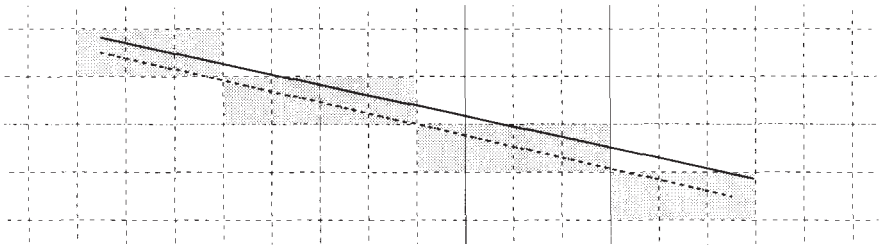
< However, currently, it's indeterminate which column or row of adjacent pixels would be used to display vertical and horizontal lines like (h) and (i) above. Point (j) would not be visible. >

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:



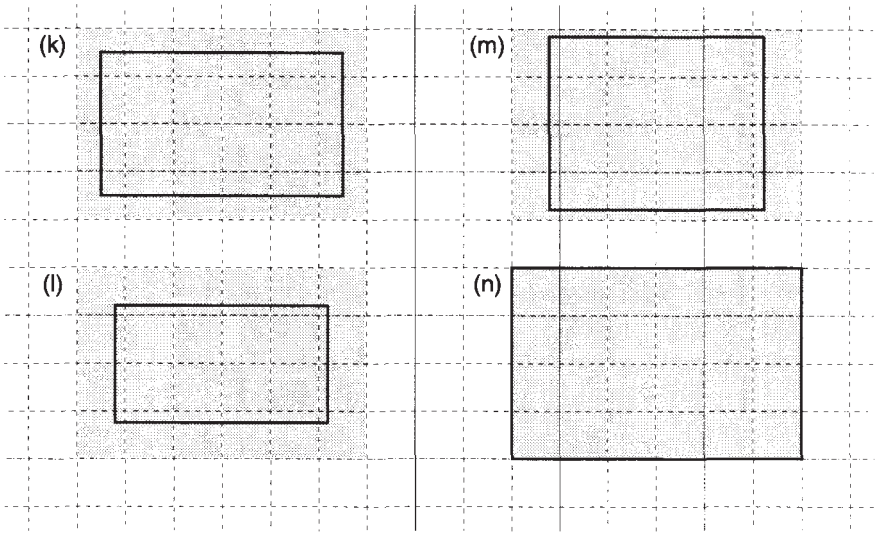
This normalization is nothing more than the natural consequence of the rounding that occurs when coordinate values are translated to device-specific pixel values.

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.



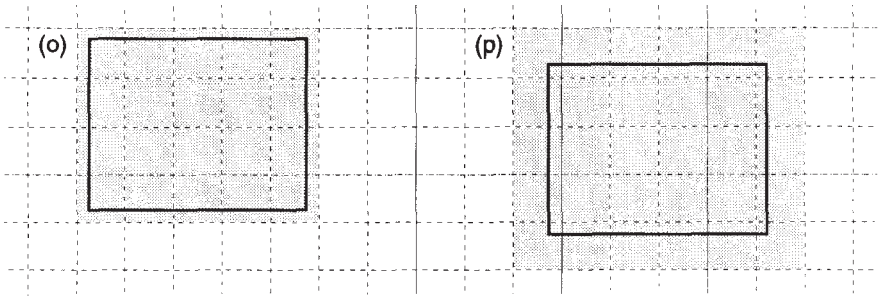
A rectangle includes every pixel that it encloses and every pixel that its sides pass through. However, as rectangle (n) illustrates, it doesn't include pixels that its sides merely touch at the boundary.

If the pixel grid in this illustration represents the screen, rectangle (n) would have left, top, right, and bottom coordinates with fractional values of .5. Rectangle (k), on the other hand, would have coordinates without any fractional parts. Nonfractional coordinates lie at the center of screen pixels.

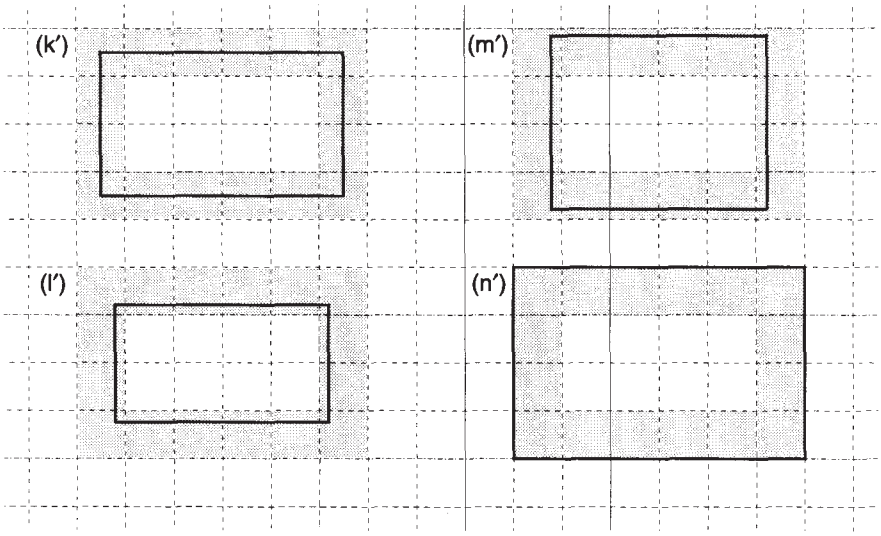
Rectangle (k), in fact, is the normalized version of all four of the illustrated rectangles. It shows how the sides of the four rectangles would be translated to pixel values. Note that for a rectangle like (n), with edges that fall on pixel boundaries, normalization means rounding the left and top sides 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 6x4 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:

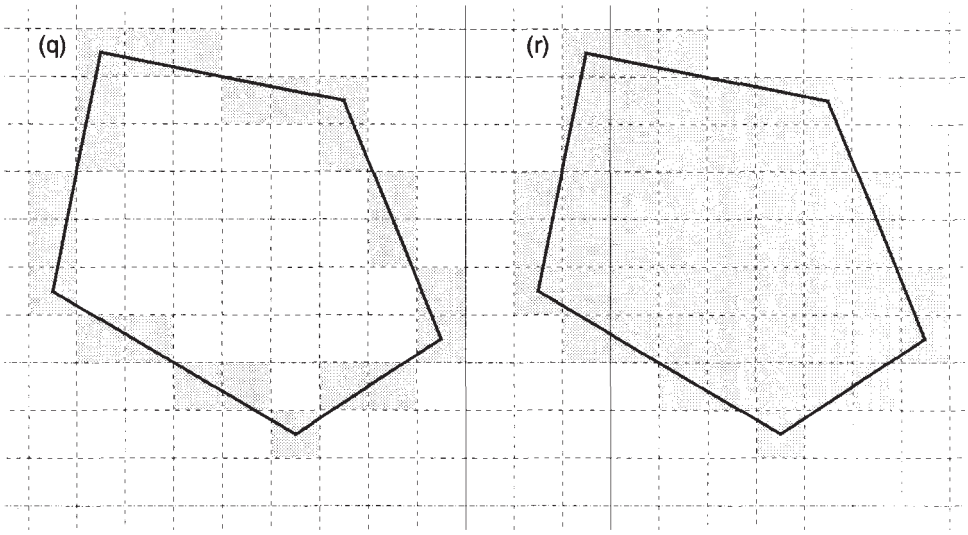


Each of the rectangles still covers a 6 × 4 pixel area. Note that even though the path of rectangle (n´) lies entirely on pixel boundaries, pixels below it and to its right are not touched by the pen. The pen touches only pixels that lie within the rectangle.

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.

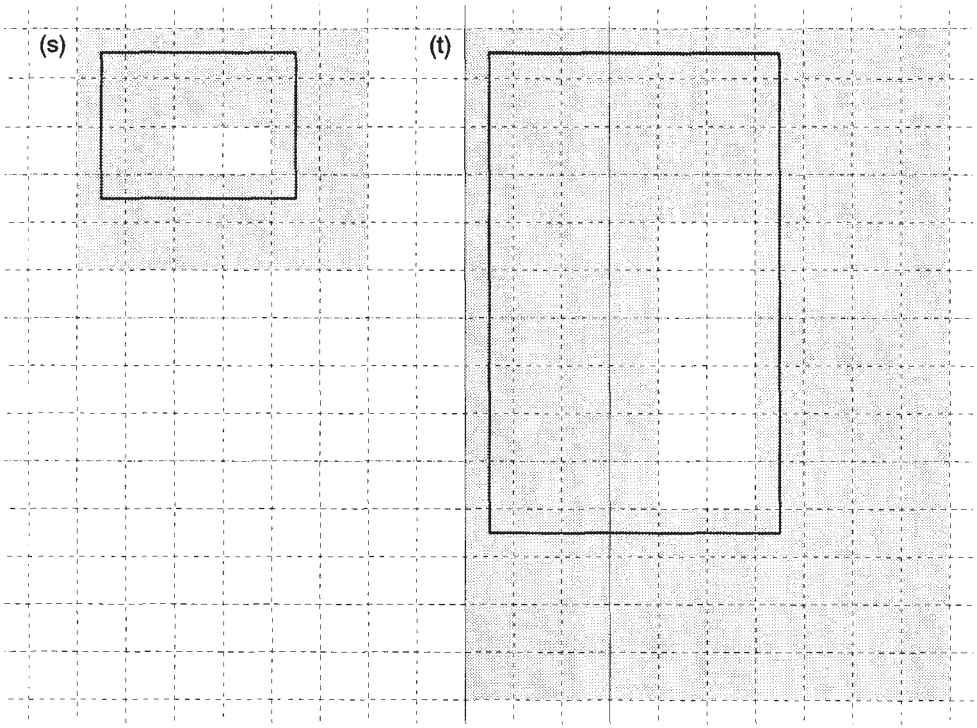
When the polygon is filled, no additional pixels around its border are colored. As is the case for a rectangle, the displayed shape of filled polygon is identical to the shape of the polygon when stroked with a one-pixel pen. The pen doesn't touch any pixels when stroking the polygon that aren't colored 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

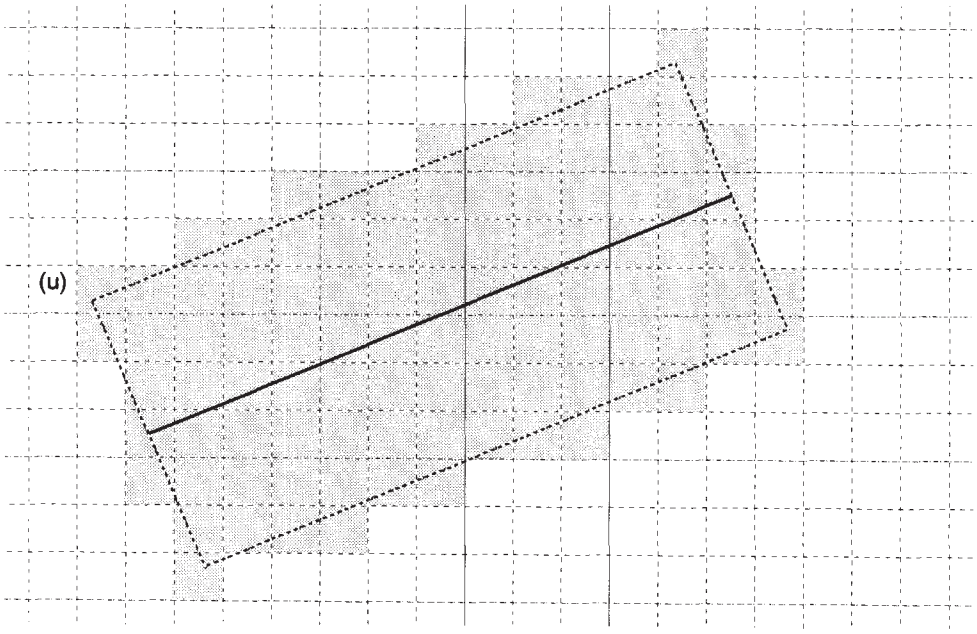
A pen that's thicker than one pixel touches the same pixels that a one-pixel pen does, but it adds extra columns and rows adjacent to the line path.

If the path is a rectangle, the pen tip is, in effect, a square. The left top corner of the tip follows the line path, so additional columns and rows of pixels are colored to the right and

below the rectangle. The following illustration shows rectangles stroked with two- and four-pixel pens:



A square tip is used only for rectangular paths. If the path is anything but a rectangle, the pen tip is, in effect, a linear brush that's kept centered on the line and perpendicular to the line path. The edges of the brush mark out a polygon around the path. Stroking the line is equivalent to filling the polygon, following the rules outlined above. The illustration below shows a short line that's six pixels thick:



In this way, lines retain their shape even when rotated.

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

Nineteen interface messages are currently defined. Two of them command the window to do something in particular:

- A *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 *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 reflects 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 events capture atomic user actions on the keyboard and mouse:

- A *key-down* event occurs 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 *key-up* event occurs when the user releases the character key. < Currently, this event isn't reported in an interface message. >

- 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 occurs (and 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 *mouse-up* event occurs when the user releases the mouse button. The event occurs only for the last button the user releases—that is, only if no other mouse button remains down.

- A *mouse-moved* event 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.

A closely related event announces the arrival of a package of information:

- A *message-dropped* event occurs when the user releases the mouse button after dragging an image from one view to another. The image represents information bundled in a BMessage object. The message is "dropped" on the view where the cursor is located when the mouse button goes up.

The six events above are all directed at particular views. Three others also concern views:

- A *view-moved* event occurs 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 recorded.

- A *view-resized* event occurs 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 *value-changed* event occurs when the Application Server changes a value associated with an object. Currently, the event occurs only for BScrollBar objects. Repeated events are reported as the user manipulates a scroll bar.

A few events affect the window itself:

- An *activation* event happens when a window becomes the active window, and 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 *quit-requested* event occurs when the user clicks a window's close box, or when the system perceives some other reason to request the window to quit.

- A *window-moved* event records the new location of a window that has been moved, either programmatically or by the user. When the user drags a window, repeated

events occur, each one capturing a small portion of the window's continuous movement. Only one event occurs when the program moves a window.

- A *window-resized* event occurs when the window is resized, again either programmatically or by the user. The event is reported repeatedly as the user resizes the window, but only once each time the application resizes it.
- A *screen-changed* event occurs when the configuration of the screen—the size of the pixel grid it displays < or the color space of the frame buffer >—changes. Such changes may require the window to take compensatory measures.

Two events are produced by the save panel:

- A *save-requested* event occurs when the user operates the panel to request that a document be saved.
- A *panel-closed* event occurs when the application or the user closes the panel. Finally, there's one event that doesn't derive from a user action:
- Periodic *pulse* events occur 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

As the user acts, interface messages are generated and reported to the application. 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 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. 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.

The charts 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.

Instruction type	Virtual function	Class
Zoom	**Zoom()**	BWindow
Minimize	**Minimize()**	BWindow
Event type	**Virtual function**	**Class**
Key-down	**KeyDown()**	BView
Key-up	*none*	
Mouse-down	**MouseDown()**	BView
Mouse-up	*none*	
Mouse-moved	**MouseMoved()**	BView
Message-dropped	**MessageDropped()**	BView
View-moved	**FrameMoved()**	BView
View-resized	**FrameResized()**	BView
Value-changed	**ValueChanged()**	BScrollBar
Window-activated	**WindowActivated()**	BWindow and BView
Quit-requested	**QuitRequested()**	BLooper, inherited by BWindow
Window-moved	**FrameMoved()**	BWindow
Window-resized	**FrameResized()**	BWindow
Screen-changed	**ScreenChanged()**	BWindow
Save-requested	**SaveRequested()**	BWindow
Panel-closed()	**SavePanelClosed()**	BWindow
Pulse	**Pulse()**	BView

< Key-up events are currently not reported. > Mouse-up events are reported, but the messages 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-moved and mouse-up messages from the BWindow's message queue, so the same information won't be reported twice.

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, most are handled by BViews. When the BWindow receives a 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. When the user drags a message to a window and drops it there, it calls the **MessageDropped()** function of the view the cursor points to.

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 depends on which kind of key-down event it is:

- If the user holds a Command key down while pressing a character key, the event 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 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*.

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 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. If it displays editable data, it's also expected to handle commands that target the current selection. When the user presses a key on the keyboard, the BWindow calls the focus view's **KeyDown()** virtual function. When the user pastes material from the clipboard, the application should arrange for the focus view to respond.

The focus doesn't have to stay on one view all the time; it can shift 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. 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()** or **MessageDropped()** 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 so that the user can operate it from the keyboard,
- It has a **KeyDown()** function to display typed characters, or
- It can show the current selection, whether or not it displays what the user types.

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).

Filtering Events

A BWindow can scrutinize the messages that report mouse and keyboard events before it gives the target BView a chance to respond. The BWindow class declares four hook functions that preview events before a BView is notified:

FilterKeyDown(),
FilterMouseDown(),
FilterMouseMoved(), and
FilterMessageDropped()

These functions give BWindows an opportunity to modify aspects of the report or even change the BView that will be expected to respond. Unless the BWindow completely intercepts the message, the responsible BView is notified through its **KeyDown()**, **MouseDown()**, **MouseMoved()**, or **MessageDropped()** function.

The filter functions are rarely implemented to prevent the BView functions from being called. Since the response to an message depends on what prompted it—for example, a click would mean one thing to a button and quite another to a text field—the principal message-handling code must be located within BViews, not at the BWindow level.

Message Protocols

Interface messages are delivered to the window thread as BMessage objects. The object's **what** data member is a constant that always names the event it reports or the instruction it gives. The constants for interface messages are:

B_KEY_DOWN
B_KEY_UP
B_MOUSE_DOWN
B_MOUSE_UP
B_MOUSE_MOVED

B_MESSAGE_DROPPED

B_VIEW_MOVED
B_VIEW_RESIZED
B_VALUE_CHANGED

B_ZOOM
B_MINIMIZE

B_WINDOW_ACTIVATED
B_QUIT_REQUESTED
B_WINDOW_MOVED
B_WINDOW_RESIZED
B_SCREEN_CHANGED

B_SAVE_REQUESTED
B_PANEL_CLOSED

B_PULSE

Typically, the BMessage object also carries various kinds of data describing the event or clarifying the instruction. In some cases, it may contain 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 of a doubleclick, 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 **CurrentMessage()**, 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 following sections list the data that's available from the BMessage objects that carry interface messages.

Zoom Instructions

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.

Minimize Instructions

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.

Key-Down Events

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 bit field 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

later section, "Keyboard Information" on page 55, 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. It's described under "Modifier Keys" on page 59 below.

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 56.

The "states" bit field has one bit assigned to each key. For most keys, the bit is set to 1 if the key is down, and to 0 if the key is up. However, the bits corresponding to keys that toggle keyboard locks (the Caps Lock, Num Lock, and Scroll Lock keys) are set to 1 if the lock is on, and to 0 if the lock is off. See "Key States" on page 64 for details on how to read information from the "states" array.

Key-Up Events

< Key-up events are not currently reported. >

Mouse-Down Events

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 59.

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:

PRIMARY_MOUSE_BUTTON
SECONDARY_MOUSE_BUTTON
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.

Mouse-Up Events

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 59.

Mouse-Moved Events

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.
"dragging"	**B_OBJECT_TYPE**	A pointer to a BMessage object that the user is dragging, or **NULL** if nothing is being dragged.

The "area" constant records which part of the window the cursor is over. It can be:

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 button.
B_RESIZE_AREA	The cursor is over the area where the window can be resized.
B_UNKNOWN_AREA	It's not known where the cursor is.

If the location of the cursor is unknown, it's probably because it just left the window.

The "buttons" mask is formed from one or more of the following constants:

PRIMARY_MOUSE_BUTTON
SECONDARY_MOUSE_BUTTON
TERTIARY_MOUSE_BUTTON

If no buttons are down, the mask is 0.

Message-Dropped Events

Data name	Type code	Description
"when"	**B_DOUBLE_TYPE**	When the message was dropped, 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 to drop the

		dragged message. The point is expressed in window coordinates.
"offset"	**B_POINT_TYPE**	Where the cursor was located inside the rectangle or image being dragged. The point is expressed in coordinates relative to an origin at the left top corner of the rectangle or image.

A **B_MESSAGE_DROPPED** BMessage simply informs the window that another BMessage has been dragged to it and dropped on one of its views. The dropped BMessage is passed to the BView as an argument in a **MessageDropped()** function call; it's not recorded as part of the message-dropped event.

View-Moved Events

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.

View-Resized Events

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.)

A **B_VIEW_RESIZED** BMessage 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.

Value-Changed Events

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.

Window-Activated Events

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.

Quit-Requested Events

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.

This data entry is added by the Application Server whenever it posts a **B_QUIT_REQUESTED** message—for example, when the user clicks the window's close box. However, it's not crucial to the interpretation of the event. You don't need to add it to messages that are posted in application code.

Window-Moved Events

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.

Window-Resized Events

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.

Screen-Changed Events

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. < Given the current configuration, this will always be **B_COLOR_8_BIT**. >

Serve-Requested Events

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. See **RunSavePanel()** in the BWindow class.

Panel-Closed Events

Data name	Type code	Description
"frame"	**B_RECT_TYPE**	The frame rectangle of the save panel in screen coordinates at the time the panel was closed. (The user may have resized it and relocated it on-screen before it was closed.)
'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.

Pulse Events

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.

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 "Key-Down Events" on page 48 above, 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 BWindow, BView, and BApplication classes have **Modifiers()** functions that return 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

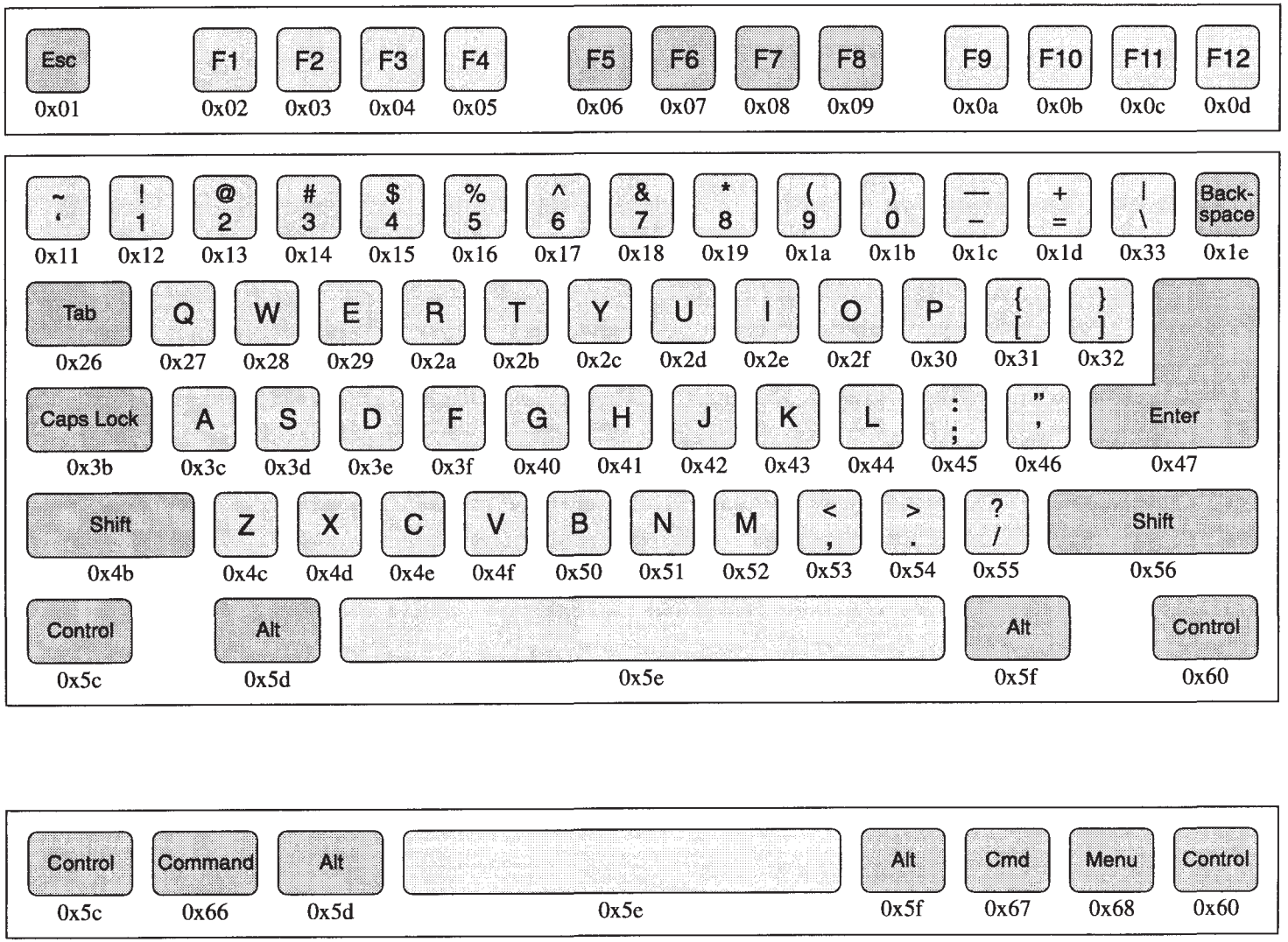
To talk about the keys on the keyboard, it's necessary first to have a standard way of identifying them. For this purpose, each key is arbitrarily assigned a numerical code.

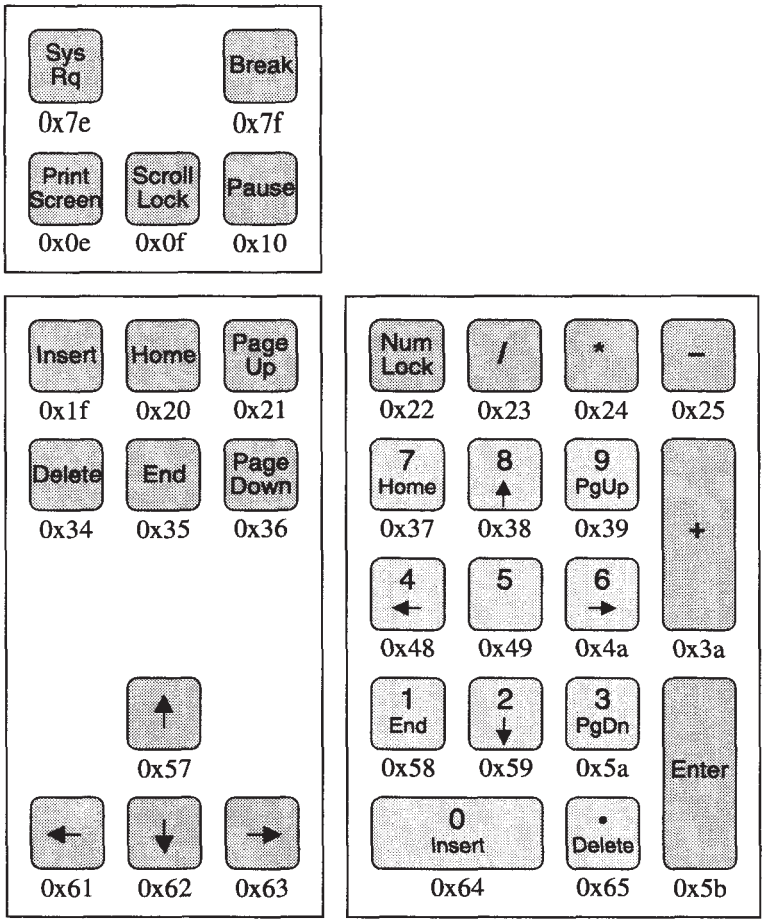
The illustrations on the next two pages show the key identifiers for a typical keyboard. The codes for the main keyboard are shown on page 57. This diagram shows a standard 101-key keyboard and an alternate version of the bottom row of keys—one that adds a Menu key and left and right Command keys.

The codes for the numerical keypad and for the keys between it and the main keyboard are shown on page 58.

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
Caps Lock 0x3b
A 0x3c
S 0x3d
D 0x3e
F 0x3f
G 0x40
H 0x41
J 0x42
K 0x43
L 0x44
: ; 0x45
” ’ 0x46
Enter 0x47
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 various **Modifiers()** functions). 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 298 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.

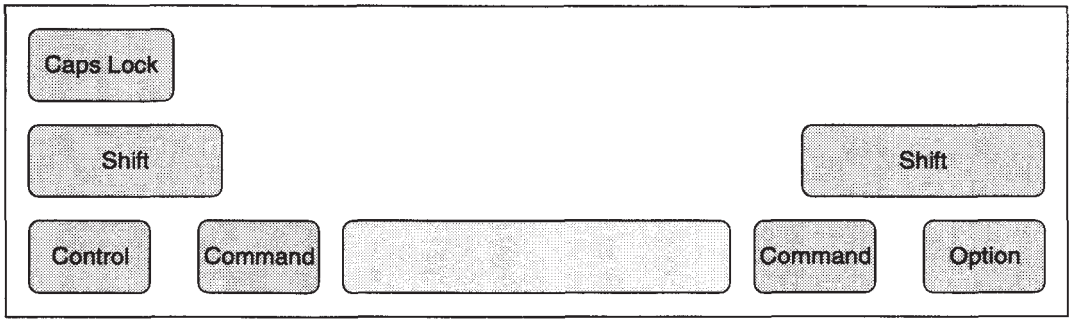
Second, the set of modifier states listed above doesn't quite match the keys that are marked on a typical keyboard. A standard 101-key keyboard has left and right "Alt(ernate)" keys, but lacks those labeled "Command," "Option," or "Menu."

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 various **Modifiers()** functions (defined by the BApplication class in the Application Kit and by BWindow and BView in the Interface Kit). It's returned, along with other information, by BView's **GetKeys()** function. And 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.

Responding to the User

A few examples are given in the table below:

Key Code	Without Modifiers	With Shift	With Option	With Shift & Option	With 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'	'©'		0x1a
0x44	'l'	'L'	'æ'	'Æ'	**B_PAGE_DOWN**
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.

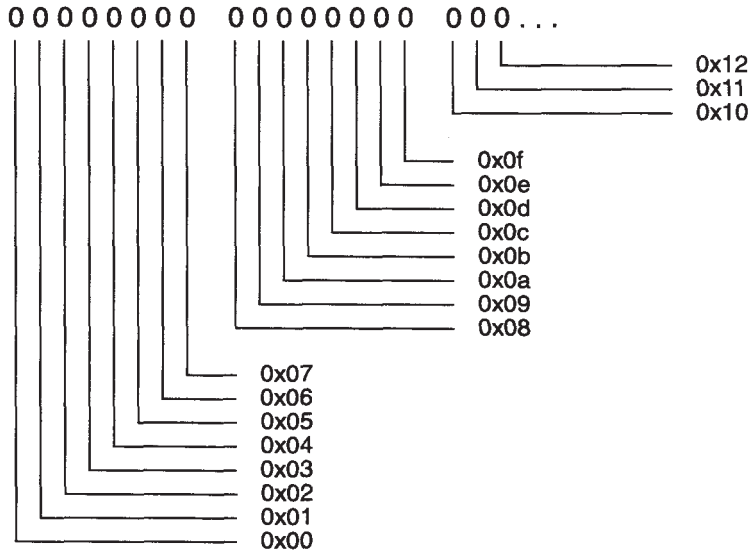
Key States

The "states" bit field that's reported in a key-down message captures the state of all keys and keyboard locks at the time of the event. At other times, you can obtain the same information through BView's **GetKeys()** function.

Although the "states" bit field is declared as **B_UCHAR_TYPE**, it's not just a single **uchar**. It's really 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 "states" mask 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.

Guide to the Classes

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 each application must also invent some BViews of its own, to do the drawing and message handling that's unique to it.

To learn about the Interface Kit for the first time, it's recommended that you first read this introduction, then look at the class descriptions in roughly the following order:

1	BWindow	Windows are at the center of the user interface. They're where applications present themselves to the user and where users do their work. All other Interface Kit objects are associated with BWindows in one way or another.
2	BView	BView objects draw within windows and handle most user actions on the keyboard and mouse. Each object corresponds to a particular *view*, one part of the window's display. Several of the other classes in the Interface Kit inherit from BView and implement particular kinds of views—such as buttons, text displays, and scroll bars. In

		conjunction with the BWindow class, BView defines the Kit's mechanisms for drawing and message-handling.
3	BPoint and BRect	These two classes define the basic data types for coordinate geometry. They're ubiquitous throughout the kit.
4	BRegion and BPolygon	Like BRect, these two classes define objects that describe areas and shapes within a coordinate system. They're used by functions in the BView class.
5	BBitmap	This class defines objects that store bitmap data. BBitmaps are passed to BView functions, which place the bitmap images on-screen.
6	BPicture	BPicture objects record a set of drawing instructions that can be replayed at will.
7	BScrollBar and BScrollView	BScrollBar objects provide scroll bars for an application, and a BScrollView sets up the scroll bars for a target view. Scrolling is explained in the BView and BScrollBar class descriptions.
8	BMenu, BMenuItem, BMenuBar, and BPopUpMenu	These classes implement the Be menu system. A BMenu object represents a menu list, and a BMenuItem represents a single item in the list. An item can control a submenu—another BMenu object—so menus can be hierarchically arranged. A BMenuBar is the visible menu at the root of the hierarchy.
9	BTextView	A BTextView object displays text on-screen and implements the user interface for editing and selecting text.
10	BControl, BButton, BRadioButton, BCheckBox, and BPictureButton	The BControl class is the base class for objects that implement control devices. The other four classes are derived from BControl.
11	BListView	A BListView is similar to the control classes. It displays a list of items that the user can select and invoke. This class is based on the BList class of the Storage Kit.
12	BAlert	A BAlert runs a modal window that alerts the user to something and asks for a response. It's a convenience for putting warnings and dialogs on-screen.

13	BStringView and BBox	These are simple views that don't respond to interface messages. A BString View draws a string (such as a label). A BBox draws a labeled box around other views.

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 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 *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()

FilterKeyDown()

virtual bool **FilterKeyDown(**ulong **aChar*, BView ***target***)**

Permits keyboard shortcuts to operate the buttons and dismiss the window. There's no need for your application to call or override this function. Call **SetShortcut()** to assign shortcut characters to buttons.

See also: SetShortcut()

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: FrameResized() in the BWindow class

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 mat 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 167 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 four color spaces are currently defined:

B_MONOCHROME_1_BIT
B_GRAYSCALE_8_BIT
B_COLOR_8_BIT
B_RGB_24_BIT

In the **B_RGB_24_BIT** color space, the color of each pixel is specified as an **rgb_color** value. In the **B_COLOR_8_BIT** color space, colors are specified as 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 four color spaces can be found under "Colors" on page 25 of the introduction to this chapter.)

< Currently, bitmap data is stored only in the **B_COLOR_8_BIT** and **B_MONOCHROME_1_BIT** color spaces, though it can also be specified in the **B_RGB_24_BIT** format. The **B_GRAYSCALE_8_BIT** color space is not used at the present time. >

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_24_BIT** bitmap, a pixel can be assigned the special value of **B_TRANSPARENT_24_BIT**. (Or **B_TRANSPARENT_24_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() global function

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_COLOR_8_BIT** and **B_MONOCHROME_1_BIT** are acceptable as the color_space *mode*. **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 offscreen 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: AddChild() in the BWindow class, **AttachedToWindow()** in the BView class, **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.

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.)

< Do not rely on these functions as they may not remain in the API. >

These functions fail if the BBitmap wasn't constructed to accept views.

See also: ChildAt() in the BWindow class

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 and in the overview above.

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: FindView() in the BView class

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: Lock() in the BLooper class of 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. The new *data* is copied into the bitmap beginning *offset* bytes from the start of allocated memory. To set data beginning with the first (left top) pixel in the image, the *offset* should be 0.

The 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, a conversion is automatically made. < Currently, only **B_RGB_24_BIT** data is converted, to **B_COLOR_8_BIT** data. In the conversion, colors are dithered, so that the resulting image will match the original as closely as possible, despite the lost information. **SetBits()** rejects data in **B_GRAYSCALE_8_BIT** mode. >

This function works for all BBitmaps, whether or not BViews are also enlisted to produce the image.

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. The label is drawn in the current font, which **AttachedToWindow()** sets to a 9-point "Erich." 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_TOP,
ulong *flags* = B_WILL_DRAW**)**

Initializes the BBox by passing all arguments to the BView constructor. 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

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets the default font for drawing the label to the 9-point "Erich" bitmap font.

This function is called by the Interface Kit; you shouldn't call it yourself. However, you can reimplement it to set a different font and other graphics parameters—such as the high color and pen size that will be used to draw the box.

See also: AttachedToWindow() in the BView class

Draw()

virtual void **Draw(**BRect *updateRect***)**

Draws the box and its label. This function is called automatically in response to update messages.

See also: Draw() in the BView class

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, like BCheckBox and BRadioButton, 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 receive 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 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_TOP,
ulong *flags* = B_WILL_DRAW**)**

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.

See also: the BControl and BView constructors, **Invoke()** in the BControl class

Member Functions

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Augments the BControl version of this function to make 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.

This version of **AttachedToWindow()** incorporates the BControl version.

See also: AttachedToWindow() in the BControl and BView classes, **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: Draw() in the BView class

IsDefault() see MakeDefault

KeyDown()

virtual void **KeyDown(**ulong *aChar***)**

Responds to a message reporting that the user pressed the Enter key by:

- Momentarily highlighting the button and changing its value, and
- Posting a copy of the model BMessage to the target receiver.

This function is called if:

- The BButton is the default button for the window,
- The window the button is in is the active window, and
- *aChar* is **B_ENTER**.

It might also be called if the BButton object is the focus view for the active window, but BButtons normally don't make themselves the focus for keyboard events.

See also: Invoke() in the BControl class, **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: SetDefault() in the BWindow class

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. 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 receiver.

See also: MessageReceived() in the BReceiver class, **Invoke()** and **SetTarget()** in the BControl class

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 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_TOP,
ulong *flags* = B_WILL_DRAW)

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.

The frame rectangle of a BCheckBox should be at least 11.0 units high to accommodate the check box and the label in the default font. The object draws at the bottom of its frame rectangle beginning at the left side; it doesn't use any extra space there may happen to be at the top or on the right. (However, the user can click anywhere within the frame rectangle to operate the check box).

See also: the BControl and BView constructors

Member Functions

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: Draw() in the BView class

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 receiver. See BControl's **Invoke()** and **SetTarget()** functions for more information. The message is dispatched by calling the target's **MessageReceived()** virtual function.

The receiver can get a pointer to the BCheckBox from the message, and use it to discover the object's new value. For example:

```
void MyReceiver::MessageReceived(BMessage *msg)
{
    . . .
    BCheckBox *box = (BCheckBox *}msg->FindObject("source");
    if ( message->Error() == B_NO_ERROR ) {
        long value = box->Value();
        . . .
    }
    . . .
}
```

See also: Invoke(), **SetTarget()**, and **SetValue()** in the BControl class

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 turn the messages that report generic mouse and keyboard events into other messages with more specific instructions for the application. Just as a switch that you might buy in a hardware store can be hooked up to do various kinds of work, a BControl object can be customized by setting the message it posts when invoked and the target receiver that should handle the message.

The Interface Kit currently includes three classes derived from BControl—BButton, BRadioButton, and BCheckBox. 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, text fields, 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*, which can be **NULL**, 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* 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 **Draw()**, **MouseDown()**, or **KeyDown()** functions. 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 received by the designated target. By default, the target is the window where the control is located, but **SetTarget()** can designate another receiver.

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.

See also: the BView constructor, **PostMessage()** in the BLooper class of 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 set the default font for all control devices to 9-point "Emily". It also makes 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 make the font change, it calls BView's **SetFontName()**; to designate the target, it calls **SetTarget()**. Both are virtual functions.

AttachedToWindow() is called for you when the BControl becomes a child of a view already associated with the window.

See also: AttachedToWindow() and **SetFontName()** in the BView class, **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_LONG_TYPE**	When the control was invoked, as measured in milliseconds from the time the machine was last booted.
"source"	**B_OBJECT_TYPE**	A pointer to the BControl object. This permits the message receiver 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 designated target, but it does have a designated BLooper where it can post the message, it will ask the BLooper for its preferred receiver and name it as the target. Since the preferred receiver 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 BReceiver 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()

Label() *see* SetLabel()

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BControl if *flag* is **TRUE**, and disables it if *flag* is **FALSE**. **IsEnabled()** returns whether or not the object is currently enabled. BControls are enabled by default.

While disabled, a BControl typically won't post messages and won't 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 flag)
{
    if ( flag == IsEnabled() )
        return;
    BControl::SetEnabled(flag);
    /* 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. 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, **AttachedToWindow()**, **SetFontName()** in the BView class

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(**BReceiver **target*, BLooper **looper*= NULL**)**

BReceiver ***Target(**BLooper ***looper* = NULL**)** const

These functions set and return the object that's targeted to receive the messages the BControl posts (through its **Invoke()** function).

SetTarget() sets the *target* BReceiver, but is successful only if it can also discern a BLooper object where **Invoke()** can post messages to that target. **Invoke()** calls the BLooper's **PostMessage()** function and names the *target* as the **Invoke()** that should receive the message:

```
looper->PostMessage(theMessage, target);
```

If the target receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the target (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and the *target* can't supply one, the function fails and returns **B_BAD_VALUE** to indicate that the *target* alone is inadequate.

Moreover, **SetTarget()** also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. In this case, **B_MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to name a specific *looper*, but a **NULL** *target*. In this case, messages will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** target and a BWindow *looper* to **SetTarget()**,

```
myControl->SetTarget(NULL, myControl->Window());
```

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 **PreferredReceiver()** is **NULL**, the *looper* itself becomes the target.)

When successful, **SetTarget()** returns **B_NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where **Invoke()** will post messages. By default (established by **AttachedToWindow()**), both roles are filled by the BWindow where the control device is located.

See also: Looper() in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes, **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.

Each class that's derived from BControl should call **SetValue()** in its **MouseDown()** and **KeyDown()** functions 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 receiver 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_TOP,
ulong *flags* = B_WILL_DRAW | 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, **AddItem()** in the BList class

~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: AttachedToWindow() in the BView class, **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: Draw() in the BView class, **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: FrameResized() in the BView class

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: Invalidate() in the BView class

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 received by the designated target. The default target (established by **AttachedToWindow()**) is the BWindow where the BListView is located. If **SetTarget()** was called to name a particular BLooper where the message should be posted, but to set a **NULL** target, the target will be the BLooper's preferred receiver.

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	Invokes the currently selected item.

This function is called to notify the BListView of key-down events whenever it's the focus view in the active window; you shouldn't call it yourself.

See also: KeyDown() in the BView class, **Select()**, **Invoke()**

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: MouseDown() in the BView class, **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 BReceiver. 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: SetFontName() in the BView class

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: SetSymbolSet() in the BView class

SetTarget(), Target()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper*= NULL**)**

BReceiver ***Target(**BLooper ***looper*= NULL**)** const

SetTarget() sets the *target* BReceiver that's expected to handle messages the BListView posts (through its **Select()** and **Invoke()** functions). It's successful only if it can also learn about a BLooper object where messages can be posted to the 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:

```
looper->PostMessage(theMessage, target);
```

If the *target* receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the *target* (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and can't be discovered from the *target*, the function fails and **B_BAD_VALUE** is returned to indicate that the target alone is insufficient.

Moreover, **SetTarget()** also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. In this case, **B_MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to specify a **NULL** *target*. In this case, the message will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a

BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** *target* and a BWindow *looper* to **SetTarget()**,

```
myList->SetTarget(NULL, myList->Window());
```

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* doesn't have a preferred receiver (as a BLooper doesn't by default, and a BWindow won't if none of its views are currently in focus), the message will be targeted to the *looper* itself.

If both *target* and *looper* are **NULL**, the function fails and **B_BAD_VALUE** is returned. When successful, **SetTarget()** returns **B_NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where **Invoke()** will post messages. By default (established by **AttachedToWindow()**), both roles are filled by the BWindow where the list is displayed. If the BListView isn't attached to a window and a target hasn't been set, **Target()** returns **NULL**.

See also: Looper() in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes, **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 BReceiver. What the item does depends on the content of the BMessage and the BReceiver'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 view 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: AttachedToWindow() in the BView class

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: Draw() in the BView class

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 with that particular *label* or associated with that particular *command*, this function returns the first one it finds (the one with the lowest index). If none of the items in the menu meet the criterion, it 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: SetMarked() in the BMenuItem class, **SetRadioMode()**

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: Show() in the BView class, **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: KeyDown() in the BView class

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 *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenu if *flag* is **TRUE**, and disables it if *flag* 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: SetEnabled() in the BMenuItem class

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 on radio mode.

See also: SetMarked() in the BMenuItem class, **SetLabelFromMarked()**

SetTarget()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = NULL**)**

This function is a convenience for assigning the same *target* and *looper* to all items in the menu. It works through the list of items in order, calling BMenuItem's **SetTarget()** virtual function for each one. However, 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* (and *looper*) of all items, it returns **B_NO_ERROR**. See BMenuItem's **SetTarget()** for information on acceptable *target* and *looper* values.

This function doesn't work recursively; it acts only on items added to the BMenu, not on items added to submenus of the BMenu.

See also: SetTarget() in the BMenuItem class

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: SetTrigger() in the BMenuItem class

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(**void**)**

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.

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 "Main" 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 a menu bar. 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 there's only one BMenuBar in the window's view hierarchy, the Menu key (or Command) will put it in focus. But if there's more than one BMenuBar object, the Menu key must choose one of them. By default, it selects the last one added to the window. However, the **SetMainMenuBar()** function defined in the BWindow class can be called to designate a different BMenuBar object as the "main" 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_TOP_RIGHT,
menu_layout *layout* = B_ITEMS_IN_ROW,
bool *resizeToFit* = FALSE**)**

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 "real" menu bar in a window should have a frame rectangle just high enough to accommodate a single row of items and a border. Given the default font currently used for menu items, the *frame* height should be about 14.0 coordinate units.

The layout of the menu determines how items are arranged. By default, they're arranged in a row as befits a true menu bar. If an instance of this class isn't being used to implement an actual menu bar, 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 **TRUE**, the frame rectangle of the BMenuBar will be resized to exactly fit the items that are added to the object. This usually is not what's desired. For a true menu bar, the frame rectangle should stretch all the way across the window, from the left side to the right, no matter how many items it contains. The default resizing mode of **B_FOLLOW_LEFT_TOP_RIGHT** permits the menu bar to adjust itself to changes in the window's width, while keeping it glued to the top of the window.

Change the *resizingMode*, the *layout*, and the *resizeToFit* flag for BMenuBars that are used for a purpose other than to implement a true menu bar.

See also: the BMenu constructor

~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 receiver 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 "main 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 "main" menu bar should always be the real menu bar at the top of the window. It can be explicitly set with BWindow's **SetMainMenuBar()** function.

See also: SetMainMenuBar() in the BWindow class

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: Draw() in the BView class

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: MouseDown() in the BView class

SetBorder()

void **SetBorder(**ulong *border***)**

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.

The default is **B_BORDER_FRAME**.

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 to a target BReceiver, usually the window where the menu at the root of the hierarchy (a BMenuBar object) is displayed. The action mat the item initiates, or the state that it sets, depends entirely on the message and the receiver’s response to it.

The target receiver 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 that the user can type in combination with the 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 drawn in a distinctive color. Like shortcuts, triggers are case-insensitive.

For an item to have a keyboard shortcut, the application must explicitly assign one when constructing the object. 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***)**

Initializes the BMenu 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*.

Whenever the user invokes the item, the model message is copied and the copy is posted to the target receiver. Three pieces of information are added to the copy before it's posted:

Data name	Type code	Description
"when"	**B_LONG_TYPE**	The time the item was invoked, as measured in milliseconds 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 doesn't post messages—its role is to bring up the submenu—and it can't take a shortcut. 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 want to draw a textual string, it should do so by assigning the string as 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: AddItem() in the BMenu class

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: AddItem() in the BMenu class

Message() *see* SetMessage()

SetEnabled(), IsEnabled()

virtual void **SetEnabled(**bool *flag***)**

bool **IsEnabled(**void**)** const

SetEnabled() enables the BMenuItem if *flag* is **TRUE**, disables it if *flag* 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: SetEnabled() in the BMenu class

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: SetLabelFromMarked() in the BMenu class, 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: SetLabelFromMarked() and **FindMarked()** in the BMenu class

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.

When a menu item is invoked, its model message is copied, relevant information is added to the copy, and the copy is posted to the target BReceiver. (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—if, for example, it controls a submenu or is a separator item—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 or an item that controls a submenu. However, the message will never be used.

See also: the BMenuItem constructor, **SetTarget()**

SetTarget(), Target()

virtual long **SetTarget(**BReceiver **target*, BLooper **looper* = NULL**)**

BReceiver ***Target(**BLooper ***looper*= NULL**)** const

These functions set and return the object that's targeted to receive messages posted by the BMenuItem.

SetTarget() sets the *target* BReceiver, but is successful only if it can also discern a BLooper object where the BMenuItem can post messages to that target. The BMenuItem calls the BLooper's **PostMessage()** function and names the *target* as the object that should receive the message:

```
looper->PostMessage(theMessage, target);
```

If the *target* receiver passed to **SetTarget()** is itself a BLooper object (such as a BWindow) or if it's associated with a BLooper object (as BViews are associated with BWindows), the *looper* argument can be **NULL**. **SetTarget()** can discover the BLooper from the *target* (by calling the *target*'s **Looper()** function).

However, if the *target* can't supply a BLooper object, a specific *looper* must be named as an argument. If a *looper* isn't named and the *target* can't supply one, the function fails and returns **B_BAD_VALUE** to indicate that the *target* alone is insufficient.

Moreover, it also fails if a specific *looper* is named but the *target* is associated with some other BLooper object. **B_MISMATCHED_VALUES** is returned to indicate that there's a conflict between the two arguments.

It's also possible to name a specific *looper*, but a **NULL** *target*. In this case, messages will be targeted to the *looper*'s preferred receiver (the object returned by its **PreferredReceiver()** function). For a BWindow, the preferred receiver is the current focus view. Therefore, by passing a **NULL** *target* and a BWindow *looper* to **SetTarget()**,

```
myItem->SetTarget(NULL, myItem->Window());
```

the BMenuItem can be targeted to whatever BView happens to be in focus at the time it's invoked. This is useful for items like "Cut" and "Copy" that act on the current selection. (Note, however, that if the **PreferredReceiver()** is **NULL**—if there's no current focus view—the BWindow itself will be the target.)

At least one of the two arguments must point to a real object. If both *target* and *looper* are **NULL**, **SetTarget()** fails and returns **B_BAD_VALUE**. When successful, it returns **B_NO_ERROR**.

Target() returns the current target and, if a pointer to a *looper* is provided, fills in the BLooper where the item will post messages. By default, both roles are 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: Looper() in the BReceiver and BView classes, **PreferredReceiver()** in the BLooper and BWindow classes

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 drawn in an eye-catching color.

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: SetTriggersEnabled() in the BMenu class

Shortcut()

char **Shortcut(**ulong **modifiers* = NULL**)** const

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 set by the BMenuItem constructor.

See also: the BMenuItem constructor

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: BeginPicture() and **DrawPicture()** in the BView class, 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(kbuttonBitmap);
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_TOP,
ulong *flags* = B_WILL_DRAW**)**

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 receiver 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()**

~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: Draw() in the BView class

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 receiver. 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: MouseDown() in the BView class, **Invoke()** in the BControl class, **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()

virtual void **SetEnabledOff(**BPicture **picture***)**

virtual void **SetEnabledOn(**BPicture **picture***)**

virtual void **SetDisabledOff(**BPicture **picture*)

virtual void **SetDisabledOn(**BPicture **picture***)**

These functions set 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** (unless it's being operated), so it needs only a single BPicture to indicate disabling; it can be set by calling **SetDisabledOff()**.

All four of these 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.

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: Contains() in the BRect class

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, 3 3.0);
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.

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: PrintToStream() in the BPoint class

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 comer 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 mat are added to it.

The new BPopUpMenu is empty; you add items to it by calling BMenu's **AddItem()** function.

See also: SetRadioMode() and **SetLabelFromMarked()** in the BMenu class

~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**)**

Places the pop-up menu on-screen and keeps it there as long as the user holds a mouse button down. The menu appears on-screen so that its left top corner is located at *screenPoint* in the screen coordinate system. When the user releases the mouse button, the menu is hidden again and **Go()** returns. 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;
    BReceiver *target = selected->Target(&looper);
    if ( target == NULL )
        target = looper->PreferredReceiver();
    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 receiver.
- 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.

See also: SetMessage() in the BMenuItem class

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: SetLabelFromMarked() and **ScreenLocation()** in the BMenu class, the BPopUpMenu constructor

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_TOP,
ulong *flags* = B_WILL_DRAW**)**

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 to the target receiver.

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 frame rectangle of a BRadioButton must be at least 12 units high (a difference of 11 between the bottom and the top) to accommodate the icon and the label in the default font. Anything over a height of 12 is superfluous; the BRadioButton draws at the bottom of the 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 *frame* to turn on the radio button).

See also: the BControl and BView constructors

Member Functions

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: Draw() in the BView class

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 BReceiver. Unlike a BCheckBox, a BRadioButton posts the message—it's "invoked"—only when it's turned on, not when it's turned off.

To set the value of each radio button in the group, this function calls **SetValue()** (a hook function defined in the BControl class).

See also: Invoke() and **SetTarget()** in the BControl class

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: SetValue() in the BControl class

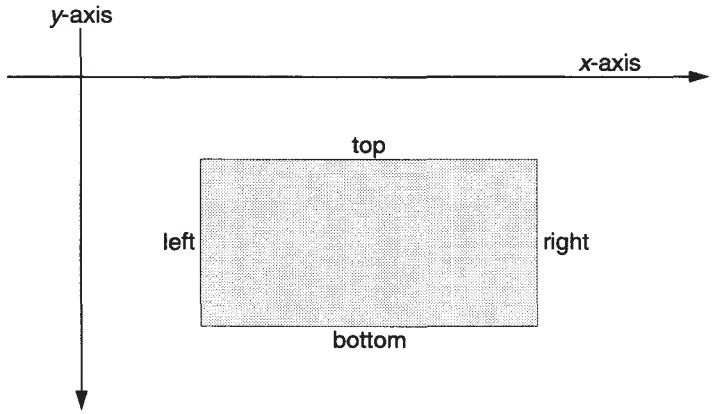
BRect

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:



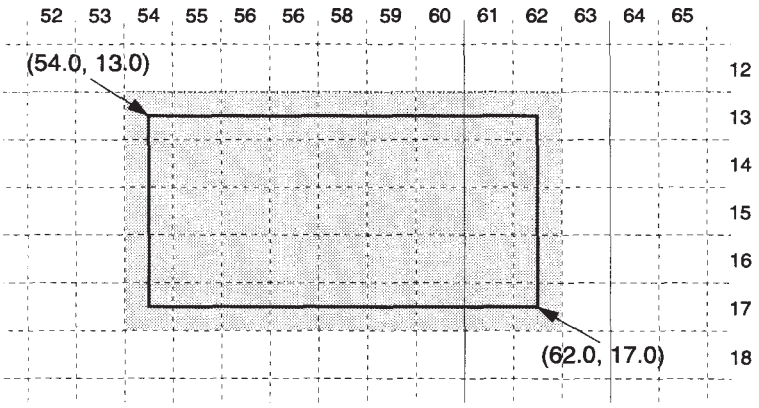
In a valid rectangle, the top *y* coordinate value is never greater than the bottom *y* coordinate, and the left *x* coordinate value is never greater than the right.

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, **ConstrainTo()** in the BPoint class

Height() seeWidth()

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()

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 **LeftBoftom(**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 its bottom and top coordinates) and its height (the difference between its right and left sides). 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.

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)

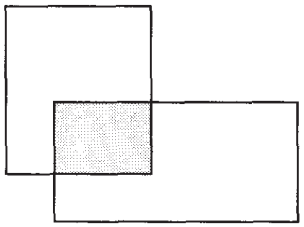
char operator **!=(**BRect**)** const

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);
```

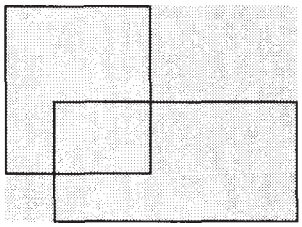
If the two rectangles don't actually intersect, the result will be invalid. You can test for this by calling the **Intersects()** function on the original rectangles, or by calling **IsValid()** on the result.

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: IsValid() in the BRect class

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: PrintToStream() in the BRect class, **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 button, 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.

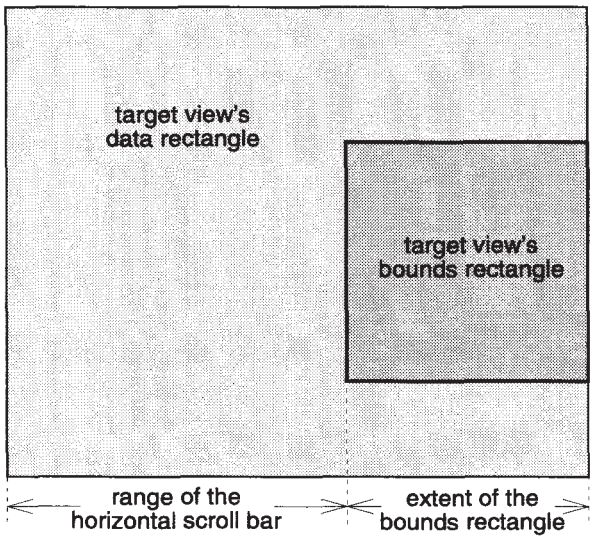
Value and Range

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.

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. A horizontal scroll bar should be exactly 12.0 units high, and a vertical scroll bar should be exactly 12.0 pixels wide. < These values may change as the user interface changes. >
- 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 of **B_FOLLOW_LEFT_RIGHT_BOTTOM**) and vertical scroll bars resize themselves vertically (as if their resizing mode was **B_FOLLOW_TOP_RIGHT_BOTTOM**).

~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

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 buttons 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(**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()**

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_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**, 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

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, **Draw()** in the BView class

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

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: AddSeparatorItem() in the BMenu class

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_TOP,
ulong *flags* = B_WILL_DRAW**)**

Initializes the BStringView by assigning it a *text* string. 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()**.

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.

~BStringView()

virtual **~BStringView(**void**)**

Frees the text string.

Member Functions

Alignment() *see* SetAlignment()

AttachedToWindow()

virtual void **AttachedToWindow(**void**)**

Sets the default font for drawing the label to the 9-point "Erich" bitmap font. This function is called by the Interface Kit; you shouldn't call it yourself. However, you can reimplement it to set the high color or a different font for drawing the string—or simply to take notice when the BStringView becomes part of a window's view hierarchy.

See also: AttachedToWindow() in the BView class

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: Draw() in the BView class

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.

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, tab widths, and interline spacing—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 **MakeResizeable()** 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 “Double Space” 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 **SetSpacing()**. For example:

```
void myText::MessageReceived(BMessage *message)
{
    switch ( message->what ) {
    case SELECT_ALL:
        SelectAll();
        break;
    case SINGLE_SPACE:
        SetSpacing(1);
        break;
    case DOUBLE_SPACE:
        SetSpacing(2);
        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.

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 and single-spaced.
- 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 allocates memory for the text and 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-point "Erich" bitmap font (no rotation, 90° shear).

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: AttachedToWindow() in the BView class, **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: Draw() in the BView class

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: KeyDown() in the BView class, **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 in 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: GetFontInfo() in the BView class

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: MakeFocus() in the BView class, **MouseDown()**

MakeResizeable()

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.

MakeResizeable() 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()

MessageDropped()

virtual bool **MessageDropped(**BMessage **message*, BPoint *point*, BPoint *offset***)**

Takes textual data from the dropped *message* and pastes it into the text. The text replaces the current selection, or is placed at the site of the current insertion point.

This function first looks in the BMessage for an entry named "text" registered as **B_ASCII_TYPE**. Failing that, it looks for a single character named "char" registered as **B_LONG_TYPE**. If successful in finding either entry, it adds the data to the text, updates the display on-screen, and returns **TRUE**. If unsuccessful, it returns **FALSE**.

See also: AcceptsChar()

MessageReceived()

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

Overrides the BReceiver function to handle **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: SetMessage() and **SetTarget()** in the BMenuItem class

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: MouseDown() and **MakeFocus()** in the BView class

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: Pulse() in the BView class

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: ScrollBy() in the BView class

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-point "Erich".

See also: SetFontName() in the BView class

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.

SetSpacing(), Spacing()

void **SetSpacing(**long *spacing***)**

long **Spacing(**void**)** const

These functions set and return the spacing between lines of text. A value of 1 indicates single spacing, 2 double spacing, 3 triple spacing, and so on.

Single spacing is the default.

SetSymbolSet()

virtual void **SetSymbolSet(**const char **name***)**

Overrides its BView counterpart to recalculate the text layout when the symbol set changes.

See also: SetSymbolSet() in the BView class

SefTabWidth()

void **SetTabWidth(**float *width***)**

Sets the distance between tab stops to *width* coordinate units. All tabs have a uniform width.

The default tab width is 44.0.

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()

Spacing() *see* SetSpacing()

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).

Get**Text()** 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()

Window Activated()

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: WindowActivated() in the BView class, **MakeFocus()**

BView

Derived from: public BReceiver

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:

BMenu	BControl	BScrollBar
BMenuBar	BButton	BScrollView
BPopUpMenu	BCheckBox	BBox
BListView	BRadioButton	BStringView
BTextView	BPictureButton	

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 BReceiver 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 receive 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.

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 destination BView's **MessageDropped()** virtual function is called. The dragged BMessage is passed to the BView as a **MessageDropped()** argument.

Aside from creating a BMessage object and passing it to **DragMessage()**, or implementing **MouseMoved()** and **MessageDropped()** 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 30 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—**MouseDown()**, **KeyDown()**, **MessageDropped()**, 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. BText view 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 your own application-specific BViews.

Hook Functions

AttachedToWindow()	Can be implemented to finish initializing the BView once it becomes part of 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.
MessageDropped()	Can be implemented to accept or reject a BMessage dropped on the view.

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
B_FOLLOW_NONE

B_FOLLOW_ALL 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.)

< In additional to the *resizingMode* constants listed above, some obsoleted constants still work:

B_FOLLOW_LEFT_TOP
B_FOLLOW_LEFT_BOTTOM
B_FOLLOW_TOP_RIGHT
B_FOLLOW_RIGHT_BOTTOM
B_FOLLOW_LEFT_TOP_RIGHT
B_FOLLOW_LEFT_TOP_BOTTOM
B_FOLLOW_LEFT_RIGHT_BOTTOM
B_FOLLOW_TOP_RIGHT_BOTTOM

A few of these constants are still used to set default parameters in classes derived from BView. >

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 has a **Draw()** function that needs to be called on updates—the view isn't simply a container for other views; it does some drawing on its own. If this flag isn't set, the BView won't receive update notifications.
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.

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()

~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. If *aView* already has a parent, it's removed from that view and added to this one. A view can have only one parent.

If the BView is attached to a window, *aView* and all of its descendants become attached to the same window. Each of them is notified of this change through an **AttachedToWindow()** function call.

See also: AddChild() in the BWindow class, **AttachedToWindow()**, **RemoveChild()**

AddLine() *see* BeginLineArray()

AttachedToWindow()

virtual void **AttachedToWindow(**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. 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. However, the BView receives the notification before any of its children do and before they are recognized as part of the window's view hierarchy. This function should therefore do nothing that depends on descendent views being attached to the window. However, it can depend on ancestor views being attached.

AttachedToWindow() is often implemented to set up a view's graphics environment, something that can't be done before the view belongs to a window. For example:

```
void MyView::AttachedToWindow()
{
    MyBaseClass::AttachedToWindow();

    SetFontName("Emily");
    SetFontSize(14);
    SetLowColor(192, 192, 192);
}
```

The default (BView) version of **AttachedToWindow()** is 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(20 * 1000);
        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 assigns the BView a default coordinate system and returns a bounds rectangle that's the same size and shape as the frame rectangle, but with the left and top sides at 0.

See also: Frame()

ChildAt(), CountChildren()

BView ***ChildAt(**long *index***)** const

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

< The first of these functions returns the child BView at *index*, or **NULL** if the BView has no such child. The second returns the number of children the BView has. Indices begin at 0 and the children are not arranged in any particular order. Don't rely on these functions as they may not remain in the API in this form. >

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 *region* that was passed in the previous call.

See also: GetClippingRegion(), **Draw()**

ConvertToParent(), ConvertFromParent()

void **ConvertToParent(**BPoint **localPoint*) const
void **ConvertToParent(**BRect **localRect*) const

void **ConvertFromParent(**BPoint **parentPoint*) 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 the point referred to by *localPoint*, or the rectangle referred to by *localRect*, from the BView's coordinate system to the coordinate system of its parent. **ConvertFromParent()** does the opposite; it converts the point referred to by *parentPoint*, or the rectangle referred to by *parentRect*, from the coordinate system of the BView's parent to the BView's own coordinate system.

Both functions fail if the BView isn't attached to a window.

See also: ConvertToScreen()

ConvertToScreen(), ConvertFromScreen()

void **ConvertToScreen(**BPoint **localPoint*) const
void **ConvertToScreen(**BRect **localRect*) const

void **ConvertFromScreen(**BPoint **screenPoint*) const
void **ConvertFromScreen(**BRect **screenRect*) const

These functions convert points and rectangles to and from the global screen coordinate system. **ConvertToScreen()** converts the point referred to by *localPoint*, or the rectangle referred to by *localRect*, from the BView's coordinate system to the screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts the point referred to by *screenPoint*, or the rectangle referred to by *screenRect*, from the screen coordinate system to the BView's local coordinate system.

Neither function will work if the BView isn't attached to a window.

See also: ConvertToScreen() in the BWindow class, **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.

CountChildren() *see* ChildAt()

DragMessage()

void **DragMessage(**BMessage **message*, BBitmap **image*, BPoint *point*)
void **DragMessage(**BMessage **message*, BRect *rect*)

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 final 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.)

This function works only for BViews that are attached to a window.

See also: the BMessage class in the Application Kit, **MessageDropped()**, 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: UpdateIfNeeded() in the BWindow class, **Invalidate()**, **GetClippingRegion()**

DrawBitmap()

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***)**

Places 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.

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");
```

See also: MovePenBy(), **SetFontName()**

EndLineArray() *see* BeginLineArray()

EndPicture() *see* BeginPicture()

EndRectTracking() *see* BeginRectTracking()

FillArc() *see* StrokeArc()

FillEllipse() *see* StrokeEllipse()

FillPolygon() *see* Stroke Polygon ()

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.

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: FindView() in the BWindow class, **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 instructions happen to be in it 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: Flush() in the BWindow class, 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.

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 **B_FOLLOW_TOP_RIGHT_BOTTOM** 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(), **FrameMoved()** in the BWindow class, **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(), **FrameResized()** in the BWindow class, **SetFlags()**

GetCharEscapements(), GetCharEdges()

void **GetCharEscapements(**char *charArray*[], long *numChars*,
short *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.)

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. 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 32 characters, plus a null terminator. The name can be set by BView's **SetFontName()** function.
short **size**	The size of the font in points. It can be set by **SetFontSize()**.

short **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()**.
short **rotation**	The angle of rotation, which is 0.0° by default. It's set by **SetFontRotation()**.
short **ascent**	How far characters ascend above the baseline.
short **descent**	How far characters descend below the baseline.
short **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. < The **font_info** structure will be converted to floating-point values in a future release. >

See also: SetFontName()

GetKeys()

void **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 the following fields:

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 field that records the state of all the keys on the keyboard, and all keyboard locks.

These fields match the BMessage entries that record information about a key-down event.

If the *checkQueue* flag is **FALSE**, **GetKeys()** provides information about the current state of the keyboard.

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 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 the current message, 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.

When the **modifiers** field reflects the current keyboard state, it contains the same information that the **Modifiers()** function returns.

The **key_states** array works identically to the "states" array passed in a key-down message. See "Key States" on page 64 for information on how to read the array.

See also: Modifiers(), **KeyDown()**, "Keyboard Information" on page 55 of the chapter introduction

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 B View'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:

PRIMARY_MOUSE_BUTTON
SECONDARY_MOUSE_BUTTON
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()

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: Hide() in the BWindow class, **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()**, **UpdateIfNeeded()** in the BWindow class

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: CurrentFocus() in the BWindow class, **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()

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.
- 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. 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_SPACE	**B_UP_ARROW**	**B_HOME**
B_TAB	**B_DOWN_ARROW**	**B_END**
B_ESCAPE		**B_PAGE_UP**
	B_FUNCTION_KEY	**B_PAGE_DOWN**

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()** is empty.

See also: "Key-Down Events" on page 48 and "Keyboard Information" on page 55) of the chapter introduction, **FilterKeyDown()** and **SetDefaultButton()** in the BWindow class, **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: LeftTop() in the BRect class, **Bounds()**

Looper() *see* Window()

LowColor() *see* SetHighColor()

MakeFocus()

virtual void **MakeFocus(**bool *flag* = TRUE**)**

Makes the BView the current focus view for its window (if *flag* is **TRUE**), or causes it to give up that status (if *flag* 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.

If the BView isn't attached to a window, this function has no effect.

See also: CurrentFocus() in the BWindow class, **IsFocus()**

MessageDropped()

virtual bool **MessageDropped(**BMessage **message*, BPoint *point*, BPoint *offset***)**

Implemented by derived classes to read data from a *message* that the user dragged and dropped on the view and to initiate whatever course of action this new information entails. The BMessage object is freed after **MessageDropped()** returns, so you must copy any of its data you want to keep.

When the message was dropped, the cursor was located at *point* within the BView's coordinate system and at *offset* within the rectangle or image 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. >

If the BView accepts the message, it should return **TRUE**. A return of **FALSE** rejects the message and causes **MessageDropped()** to be called for the BView's parent. The notification works its way up the view hierarchy until it finds a BView that will return **TRUE**, or it reaches the top view.

The BView version of this function always returns **FALSE**; by default, views don't accept dropped messages.

Often the messages that can be successfully dropped on a view hold data that could also be pasted from the clipboard. To handle this data in common code, **MessageDropped()** and the **Paste()** function you define for the view can pass the data to a third function implemented for this purpose. **MessageDropped()** would extract the data from the *message* and **Paste()** would get it from the clipboard.

If a BView displays any of the data it takes from the message, it should generally make itself the focus view:

```
bool MyView::MessageDropped(BMessage *message,
                                BPoint point, BPoint offset)
{
    MakeFocus(TRUE);
    . . .
    return TRUE;
}
```

The messages that a user drags and drops on a view might have their source in any application. The Browser will probably be a common source, since it permits user, 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.

You can choose whether your version of **MessageDropped()** should handle these messages or not. If it does, it might simply pass them to the **RefsReceived()** function you implemented in a class derived from BApplication.

See also: "Message-Dropped Events" on page 51, **FilterMessageDropped()** in the BWindow class, **RefsReceived()** in the BApplication class of the Application Kit, **MouseMoved()**, the BMessage class

Modifiers()

ulong **Modifiers(**void**)** const

Returns a mask that has a bit set for each keyboard lock that's on and for each modifier state that's set because the user is holding down a modifier key. 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 Keyboard utility.

See also: "Modifier Keys" on page 59 of the introduction to the chapter, **GetKeys()**

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;
    . . .
    do {
        snooze(20 * 1000);
        GetMouse(&point, &buttons, TRUE);
        . . .
    } while ( buttons );
    . . .
}
```

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 one in a series constituting a multiple click.

The BView version of **MouseDown()** is empty.

See also: "Mouse-Down Events" on page 49, **FilterMouseDown()** in the BWindow class, **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: "Mouse-Moved Events" on page 51, **FilterMouseMoved()** in the BWindow class, **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()

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.

Like other functions that set graphics parameters, **MovePenBy()**, **MovePenTo()**, and **PenLocation()** work only for BViews that are attached to a window.

See also: SetPenSize()

MoveTo() *see* MoveBy()

Name() *see* SetName()

Parent()

BView ***Parent(**void) const

Returns the BView's parent, or **NULL** if the BView doesn't have one.

See also: AddChild()

PenLocation() *see* MovePenBy()

PenSize() *see* SetPenSize()

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, **SetPulseRate()** in the BWindow class

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.

See also: AddChild(), **RemoveSelf()**

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()**, **Width()** and **Height()** in the BRect class

ResizingMode() see SetResizingMode()

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. They work only for BViews that are attached to a window.

The mode can be set to 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 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, Baskerville MT, Courier New, Times New Roman, Symbol >, 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

They're available only in one size—9.0 points. The default font is "Kate". 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. >

These four font functions work only for BViews that are attached to a window. < 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. They only work for BViews that are attached to a window.

The high and low colors combine to form a pattern that's passed as an argument to most **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 augment either function in some way, you need override only the **rgb_color** version.

See also: "Patterns" on page 25 of the chapter introduction, **SetViewColor()**

SetName(), Name()

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

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

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

See also: the BView constructor, **FindView()**

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 translated from coordinate units to a device-specific number of pixels.

For stroking rectangles, the pen is a square and the size measures the number of pixels on one of its sides. When it strokes a rectangle, the left top pixel in the square follows the path of the line from pixel to pixel. Therefore, if the pen square has more than one pixel on a side, it extends to the right and hangs below the path being stroked. As it moves along the path, the pen paints all the pixels that it touches.

For stroking all other lines, the pen is 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 output devices. This guarantees that it will always draw the thinnest possible line no matter what 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.

These functions can set and return the pen size only if the BView is attached to a window.

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:

"IS() 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.

This function works only for BViews that are attached to a window. Derived classes can override it 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.

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 that it doesn't draw in, the new areas are displayed in the view color.

The view color can be set only after the view is attached to a window. It's best to set it before the window is shown on-screen. The default color is white.

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.

< The *alpha* color component is currently ignored. >

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()

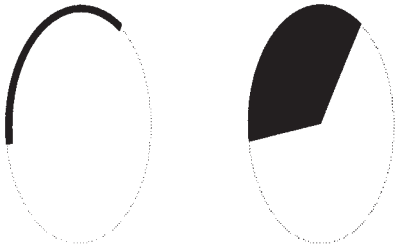
void **StrokeArc(**BRect *rect*, float angle, float span,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **StrokeArc(**BPoint *center*, float *xRadius*, float *yRadius*,
float *angle*, float *span*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **FillArc(**BRect *rect*, float *angle*, float *span*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

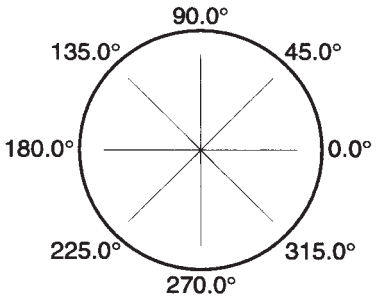
void **FillArc(**BPoint *center*, float *xRadius*, float *yRadius*,
float *angle*, float *span*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

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*, const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokeEllipse(**BPoint *center*, float *xRadius*, float *yRadius*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **FillEllipse(**BRect *rect*, const pattern **aPattern* = &B_SOLID_HIGH**)**
void **FillEllipse(**BPoint *center*, float *xRadius*, float *yRadius*,
const 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*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokeLine(**BPoint *end*, const 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*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **StrokePolygon(**BPolygon **polygon*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokePolygon(**BPoint **pointList*, long *numPoints*, BRect *rect*,
const 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. **FillPolygon()** 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.

< Currently, **StrokePolygon()** doesn't accept pen sizes other than 1 or patterns other than the default. >

See also: SetPenSize(), the BPolygon class

StrokeRect(), FillRect()

void **StrokeRect(**BRect *rect*, const pattern **aPattern* = &B_SOLID_HIGH**)**

void **FillRect(**BRect *rect*, const 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*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **FillRoundRect(**BRect *rect*, float *xRadius*, float *yRadius*,
const 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*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **StrokeTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
BRect *rect*,
const pattern **aPattern* = &B_SOLID_HIGH**)**

void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
const pattern **aPattern* = &B_SOLID_HIGH**)**
void **FillTriangle(**BPoint *firstPoint*, BPoint *secondPoint*, BPoint *thirdPoint*,
BRect *rect*,
const 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.

< Currently, **StrokeTriangle()** doesn't accept pen sizes other than 1 or patterns other than the default. >

See also: SetPenSize()

Sync() *see* Flush()

Window(), Looper()

BWindow ***Window(**void**)** const

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

Both these functions return the BWindow to which the BView belongs, or **NULL** if the BView hasn't yet been attached to a window. **Looper()** overrides the virtual function first declared in the BReceiver class to return the BWindow as a pointer to a BLooper object. **Window()** returns it more directly as a pointer to a BWindow.

See also: Looper() in the BReceiver class of the Application Kit, **AddChild()** in both this and the BWindow class, **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: WindowActivated() in the BWindow class

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

FilterKeyDown() Can be implemented to filter reports of key-down events before they're dispatched by calling the focus view's **KeyDown()** function.

FilterMessageDropped() Can be implemented to filter reports of message-dropped events before they're dispatched by calling a BView's **MessageDropped()** function.

FilterMouseDown() Can be implemented to filter reports of mouse-down events before they're dispatched by calling a BView's **MouseDown()** function.

FilterMouseMoved() Can be implemented to filter reports of mouse-moved events before they're dispatched by calling a BView's **MouseMoved()** function.

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.

WindowActivated() Can be implemented to take whatever action is necessary when the window becomes the active window, or when it loses that status.

Zoom() Zooms the window to a larger size, or from the large size to its previous state; can be reimplemented to the modify target window size or make other adjustments.

Constructor and Destructor

BWindow()

BWindow(BRect *frame*, const char **title*, window_type *type*, ulong *flags***)**

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*, sets the title the window will display if it has a tab and also 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 tab.

B_TITLED_WINDOW	A window with a border and a tab. Most windows are of this type. The title is displayed in the tab.
B_SHADOWED_WINDOW	A window with a border and tab, and a drop shadow on its right and bottom sides.

The tab, border, and drop shadow are drawn around the window's frame rectangle.

The final 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 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()**, **MessageDropped()**, **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() in this and the BView class

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. If *aView* already has a parent, it's 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.

See also: AddChild() and **AttachedToWindow()** in the BView class, **RemoveChild()**

AddShortcut(), RemoveShortcut()

void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message***)**
void **AddShortcut(**ulong *aChar*, ulong *modifiers*, BMessage **message*,
BReceiver **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* BReceiver object is specified, it will be named as the message receiver. If a *target* isn't specified, the current focus view will be named as the receiver. If there is no focus view, the BWindow will act as the receiver.

The message is dispatched by calling the receiver'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, 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. Do not rely on these functions as they may not remain in the API. >

Close() *see* Quit()

CloseSavePanel() *see* RunSavePanel()

ConvertToScreen(), ConvertFromScreen()

void **ConvertToScreen(**BPoint **windowPoint***)** const
void **ConvertToScreen(**BRect **windowRect***)** const

void **ConvertFromScreen(**BPoint **screenPoint***)** const
void **ConvertFromScreen(**BRect **screenRect***)** const

These functions convert points and rectangles to and from the global screen coordinate system. **ConvertToScreen()** converts the point referred to by *windowPoint*, or the rectangle referred to by *windowRect*, from the window coordinate system to the screen coordinate system. **ConvertFromScreen()** makes the opposite conversion; it converts the point referred to by *screenPoint*, or the rectangle referred to by *screenRect*, from the screen coordinate system to the window coordinate system.

The window coordinate system has its origin, (0.0, 0.0), at the left top corner of the window's content area.

See also: ConvertToScreen() in the BView class

CurrentFocus(), PreferredReceiver()

BView ***CurrentFocus(**void**)** const

virtual BReceiver ***PreferredReceiver(**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 **PreferredReceiver()** overrides the BLooper function to return it as a BReceiver.

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 **PreferredReceiver()** to discover where they should target messages posted to the BWindow when a specific receiver hasn't been designated. This mechanism permits these objects to target 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: MakeFocus() and **IsFocus()** in the BView class, **SetTarget()** in the BControl, BListView, and BMenuItem classes, **PreferredReceiver()** in the BLooper class

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: Invalidate() in the BView class, **UpdateIfNeeded()**

DispatchMessage()

virtual void **DispatchMessage(**BMessage **message*, BReceiver **receiver***)**

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 43 of the introduction to this chapter for a list of these functions.
- It dispatches other messages by calling the targeted *receiver*'s **MessageReceived()** function.

Derived classes can override **DispatchMessage()** to make it dispatch specific kinds of messages in other ways. For example:

```
void MyWindow::DispatchMessage(BMessage *message)
{
    if ( message->what == MAKE_PREDICTIONS )
        predictor->GuessAbout(message);
    else
        BWindow::DispatchMessage(message);
}
```

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, **DispatchMessage()** and **CurrentMessage()** in the BLooper class

EnableUpdates() *see* DisableUpdates()

FilterKeyDown()

virtual bool **FilterKeyDown(**ulong **aChar*, BView ***target***)**

Implemented by derived classes to interpret a key-down message before the window's focus view (or its default button) is notified with a **KeyDown()** function call. The first argument, *aChar*, points to the character recorded in the message. The second argument, *target*, points to the focus BView that's slated to receive the **KeyDown()** notification.

FilterKeyDown() is called for every key-down event that's reported to the window, except for those that might correspond to keyboard shortcuts. If it returns **TRUE**, the **KeyDown()** virtual function implemented for the target view will be called. If it returns **FALSE**, **KeyDown()** isn't called and the key-down message isn't handled (except to the extent that **FilterKeyDown()** itself might be implemented to handle it).

Before returning **TRUE**, this function can change the *aChar* value that will be passed to **KeyDown()**. (This, however, won't change the "char" entry of the BMessage object that

reported the event and that **CurrentMessage()** returns). It can also change the *target* BView to another view located within the same window. For example:

```
bool MyView::FilerKeyDown(ulong *aChar, BView **target)
{
    if ( *target->IsVeryMuchDisabled() )
        *target = *target->Parent();
    . . .
    if ( *aChar == B_ENTER )
        *aChar = B_TAB;
    . . .
}
```

Neither **FilterKeyDown()** nor **KeyDown()** is called for key-down events that are potential keyboard shortcuts—that is, for key-down events that are produced while holding down a Command key.

The BWindow version of **FilterKeyDown()** makes no changes to either the character or the target BView and simply returns **TRUE**.

See also: KeyDown() in the BView class, **AddShortcut()**, **Modifiers()**, "Key-Down Events" on page 48 of the introduction

FilterMessageDropped()

virtual bool **FilterMessageDropped(**BMessage **message*, BPoint *point*, BView ***target***)**

Implemented by derived classes to preview a message-dropped event before **MessageDropped()** is called for any of the window's BViews. The first argument, *message*, is the dropped message (not the message that reports the message-dropped event, but the message that the user dragged and dropped). The second argument, *point*, is the location of the cursor when the message was dropped; it's stated in the window's coordinate system.

The third argument, *target*, points to the BView that's scheduled to receive the **MessageDropped()** notification. It's the view located at *point*. However, **FilterMessageDropped()** can be implemented to replace the *target* BView with another view located within the same window. The replacement BView will then be notified instead.

FilterMessageDropped() is called whenever the user drops a dragged message within the window. By returning **TRUE**, it permits the target's **MessageDropped()** function to be called. By returning **FALSE**, it prevents any BView from notified of the message-dropped event.

The default version of **FilterMessageDropped()** simply returns **TRUE**.

See also: MessageDropped() in the BView class, **CurrentMessage()** in the BLooper class, "Message-Dropped Events" on page 51 of the introduction

FilterMouseDown()

virtual bool **FilterMouseDown(**BPoint *point*, BView ***target***)**

Implemented by derived classes to return **TRUE** if the mouse-down event located at *point* should be handled by a subsequent call to the *target* view's **MouseDown()** function, and **FALSE** if **MouseDown()** should not be called. The point is stated in the target view's coordinate system.

Before returning **TRUE**, this function can alter the BView that will receive the **MouseDown()** notification—simply by changing the object that *target* points to. The replacement target must be located in view hierarchy of the same window.

FilterMouseDown() is called for every mouse-down message the window receives. BWindow's default version of the function never alters *point* and always returns **TRUE**.

See also: MouseDown() in the BView class, **CurrentMessage()** in the BLooper class, "Mouse-Down Events" on page 49 in the chapter introduction

FilterMouseMoved()

virtual bool **FilterMouseMoved(**BPoint *point*, ulong *area*, BMessage **message*, BView ***target***)**

Implemented by derived classes to preview a mouse-moved event before **MouseMoved()** is called for any of the window's BViews. **FilterMouseMoved()** is called once for every mouse-moved message the window receives. It receives messages only while the cursor is moving over the window.

The message reports that the user has moved the cursor to a new *point* in the window's coordinate system. Normally, the BView that the cursor is over is notified by calling its **MouseMoved()** virtual function. If the cursor has moved out of one view and into another, both BViews are notified. However, by returning **FALSE**, **FilterMouseMoved()** prevents any BViews from being notified of the event. A return of **TRUE** permits **MouseMoved()** to be called.

The first argument, *point*, is the current location of the cursor, stated in the window's coordinate system. The second argument, *area*, conveys 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_ZOOM_AREA	The cursor is over the zoom button in the title tab.
B_MINIMIZE_AREA	< *Currently unused.* >
B_UNKNOWN_AREA	It's unknown where the cursor is, probably because it just left the window.

If the cursor is over a BView in the window's content area, a pointer to the view is passed as the final argument, *target*. If the cursor isn't over a BView, *target* points to a **NULL** value.

In the normal course of events, the target view will receive a **MouseMoved()** notification, provided **FilterMouseMoved()** returns **TRUE**. However, before returning **TRUE**, **FilterMouseMoved()** can alter the target view. Depending on which BView is chosen, the replacement BView will receive a **MouseMoved()** notification informing it either that the cursor has just entered it—even though the cursor is really inside another view—or that the cursor has moved somewhere else inside it, having previously entered—even though the cursor is actually no longer inside the view. If the cursor had previously entered the *target* view passed to this function, that view will be notified that the cursor has left it, even though it really hasn't.

If the user is moving the cursor to drag a BMessage object, the third argument, *message*, points to the dragged BMessage. If nothing is being dragged, *message* is **NULL**.

The BWindow version of this function simply returns **TRUE**.

See also: MouseMoved() and **DragMessage()** in the BView class, "Mouse-Moved Events" on page 51 of the chapter introduction

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: FindView() in the BView class

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: Flush in the BView class

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(), "Window-Moved Events" on page 53 of the chapter introduction

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(), "Window-Resized Events" on page 54 of the chapter introduction

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()

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

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: "Minimize Instructions" on page 48 of the chapter introduction, **Zoom()**

Modifiers()

ulong **Modifiers(**void**)** const

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**	

If a Shift, Command, Control, or Option key is down, the mask can be further tested against the following constants to reveal which key it is, the one on the left or the one on the right:

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**

No bits are set (the mask is 0) if none of the modifiers keys are down and no locks are on.

See also: Modifiers() and **GetKeys()** in the BView class, **CurrentMessage()** in the BLooper class

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()

PreferredReceiver() *see* CurrentFocus()

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: QuitRequested() and **Quit()** in the BLooper class, **QuitRequested()** in the BApplication class

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()

RemoveMouseMessages()

void **RemoveMouseMessages(**void**)**

< Removes messages reporting mouse-down and mouse-up events from the window's message queue. Don't rely on this function; it's likely to be removed from the API. Instead, get the BMessageQueue and call its **RemoveMessage()** function, as follows:

```
myWindow->MessageQueue()->RemoveMessage(MOUSE_DOWN);
myWindow->MessageQueue()->RemoveMessage(MOUSE_UP)
```

>

See also: MessageQueue() in the BLooper class of the Application Kit

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 **buttonLabel* = NULL,
BMessage **message* = NULL**)**

void **CloseSavePanel(**void**)**

bool **IsSavePanelRunning(**void**)** const

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 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 *buttonLabel* isn't provided, the principal button in the panel (the default button) will be labeled "Save". (The panel also has a "Cancel" button.)

- 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 *buttonLabel*) 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 doesn't automatically disappear when the user operates the "Save" (or *buttonLabel*) button; it remains on-screen 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" 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: “Panel-Closed Events” on page 55 of the chapter introduction, **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 its located in the screen coordinate system. Moreover, there is no way to change the screen color space. Only the pixel grid can change. >

See also: set_screen_size(), "Screen-Changed Events" on page 54 of the chapter introduction

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. **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: MakeDefault() in the BButton class

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

SetMainMenuBar()

void **SetMainMenuBar(**BMenuBar ***menuBar*)**

Makes the specified BMenuBar object the "main" menu bar for the window—the object that's at the root of the menu hierarchy that users can navigate using the keyboard.

If a window contains only one BMenuBar view, it's automatically designated the main 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 main 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 using the keyboard. This function can be called to make sure that the BMenuBar object at the root of that hierarchy is the "main" menu bar.

See also: the BMenuBar class

SetPulseRate()

void **SetPulseRate(**long *milliseconds*)**

Sets how often **Pulse()** is called for the BWindow's views.

By turning on the **B_PULSE_NEEDED** flag, a BView can request periodic **Pulse()** notifications. By default, pulse messages are posted every 500 milliseconds, as long as no other messages are pending. Each message causes **Pulse()** to be called for every BView that requested the notification.

SetPulseRate() permits you to set a different interval. The interval set should not be less than 100 milliseconds; differences less than 50 milliseconds may not be noticeable. A finer granularity can't be guaranteed.

All BViews attached to the same window share the same pulse rate.

See also: Pulse() in the BView class

SetSizeLimits(), SetZoomLimits()

void **SetSizeLimits(**float *minWidth*, float *maxWidth*,
float *minHeight*, float *maxHeight***)**

void **SetZoomLimits(**float *maxWidth*, float *maxHeight***)**

These functions set 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.

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 functions 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

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: Draw() in the BView class, **Invalidate()** in the BView class, **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: WindowActivated() in the BView class

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, the screen, installed fonts and symbol sets, and the list of possible colors.

The Application Server maintains this environment. Therefore, for any of these functions to work, your application needs a connection to the Server. The connection they all depend on is the one established when the BApplication object is constructed. Consequently, none of them should be called before a BApplication object is present in your application.

count_fonts() *see* get_font_name()

count_screens() *see* get_screen_info()

count_symbol_sets() *see* get_symbol_set_name()

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***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: GetFontInfo() and **SetFontName()** in the BView class

get_keyboard_id()

<interface/InterfaceDefs.h>

long **get_keyboard_id(**ushort **theId***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 56 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_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**)**

These functions provide information about the monitors (screens) that are currently hooked up to the BeBox.

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 screen 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. >

get_screen_info() writes information about the screen at *index* into the **screen_info** structure referred to by *theInfo*. If no index is mentioned, this function 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. (See the BBitmap class description 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.
void ***bits**	A pointer to the frame buffer.
long **bytes_per_row**	The number of bytes used to specify one row of pixel data in the frame buffer.

count_screens() returns the number of screens (monitors) that are attached to the computer. < Currently, no more than one screen can be attached, so this function always returns 1. >

See also: the BBitmap class

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: SetSymbolSet() in the BView class, **get_font_name()**

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 **B_RGB_24_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

lock_screen(), unlock_screen()

<interface/InterfaceDefs.h>

void **lock_screen(**long *index* = 0**)**

void **unlock_screen(**long *index* = 0**)**

These functions lock and unlock the screen at *index*. Indices begin at 0. The screen at index 0 is the one that has the origin of the screen coordinate system at its left top corner. < Currently, only one monitor can be attached to the BeBox, so the *index* should always be0.>

While a screen is locked, its frame buffer—and consequently the on-screen display—won't change. Updates are held until the screen is unlocked again.

The screen should be locked before reading pixel data directly from the frame buffer. The locking thread should not communicate with the Application Server until the screen is unlocked.

See also: get_screen_info()

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 supply 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. **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 Desktop application found in **/preferences**.

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() in the BView 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.

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:

PRIMARY_MOUSE_BUTTON
SECONDARY_MOUSE_BUTTON
TERTIARY_MOUSE_BUTTON

If both the **left** and **right** fields are set to **PRIMARY_MOUSE_BUTTON**, they both function as the primary button; if either is set to **PRIMARY_MOUSE_BUTTON**, it functions 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_size()

<interface/InterfaceDefs.h>

void **set_screen_size(**long *index*, float *width*, float *height*,
bool *makeDefault* = TRUE**)**

Sets the size of the pixel grid displayed on the monitor at *index* in the screen list. < Since a BeBox currently can have only one monitor, *index* should always be 0. >

The grid is the size of the screen measured in pixels—the number of pixels that it displays horizontally and vertically. Only two screen sizes are currently supported—640 x 480 and 800 x 600—so the *width* and *height* passed to this function should match these values.

If the *makeDefault* flag is **TRUE**, the new screen size becomes the default and will be used the next time the machine reboots. If the flag is **FALSE**, the change is for the current session only; the machine will reboot in the previously determined default screen size.

When the size of the screen grid 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. >

It's usually left to the user to set the screen size, with the Desktop preferences application.

See also: ScreenChanged() in the BWindow class, **get_screen_info()**

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's key map—the structure that describes the role of each key on the keyboard. The second function restores the default key 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 Keyboard utility, 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/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 five 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.

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 59 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 various **Modifiers()** functions return. 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/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. Any key that produces that character while an Option key is held down will be dead and will combine to produce the characters listed in the array.

The Option key is an essential ingredient; a key is dead only when an Option key is held down and only if it's mapped (in the four **option_..._map** tables) to the second character listed in one of the dead-key arrays.

See also: GetKeys() and **Modifiers()** in the BView class, "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. 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 member 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: SetAlignment() in the BTextView class

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
B_ENTER	0x0a
B_RETURN	0x0a (same as **B_ENTER** or '\n')
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

Enumerated constant	Character value
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_24_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 four different color spaces currently defined for this type.

See also: "Colors" on page 25, the BBitmap class

Control States

<interface/Control.h>

Enumerated constant	Value
B_CONTROL_ON	1
B_CONTROL_OFF	0

These constants define the possible states of a typical control device.

See also: SetValue() in the BControl class

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: MouseMoved() in the BView class

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, **SetDrawingMode()** in the BView class

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 61 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_MOUSE_DOWN
B_WINDOW_RESIZED	**B_MOUSE_UP**
B_WINDOW_MOVED	**B_MOUSE_MOVED**
B_WINDOW_ACTIVATED	
B_QUIT_REQUESTED	**B_MESSAGE_DROPPED**
B_SCREEN_CHANGED	
	B_VIEW_RESIZED
B_SAVE_REQUESTED	**B_VIEW_MOVED**
B_PANEL_CLOSED	**B_VALUE_CHANGED**
B_PULSE	

These constants identify interface messages—system messages that are delivered to BWindow objects. Each constant names an instruction to do something in particular (**B_ZOOM**) or a type of event (**B_KEY_DOWN**).

See also: "Interface Messages" on page 41 of the introduction to this chapter

Menu Bar Borders

<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: SetBorder() in the BMenuBar class

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_NUMLOCK	

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() in the BView and BWindow classes, **AddShortcut()** in the BWindow class, 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, but by their roles in the user interface.

See also: GetMouse() in the BView class, **set_mouse_map()**

orientation Constants

<interface/InterfaceDefs.h>

Enumerated constant

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, 00xaa, 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 25 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, **SetResizingMode()** in the BView 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: BeginRectTracking() in the BView class

Transparency Constants

<interface/InterfaceDefs.h>

const uchar **B_TRANSPARENT_8_BIT**
const rgb_color **B_TRANSPARENT_24_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_24_BIT** designates a transparent pixel in the **B_RGB_24_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

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, **SetFlags()** in the BView class

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 in messages that report mouse-moved events to designate the area where the cursor is located.

See also: FilterMouseMoved() in the BWindow class

Window Flags

<interface/Window.h>

const long **B_NOT_MOVABLE**
const long **B_NOT_H_RESIZABLE**
const long **B_NOT_V_RESIZABLE**
const long **B_NOT_RESIZABLE**
const long **B_WILL_ACCEPT_FIRST_CLICK**
const long **B_NOT_CLOSABLE**
const long **B_NOT_ZOOMABLE**
const long **B_NOT_MINIMIZABLE**
const long **B_WILL_FLOAT**

These constants set the behavior of a window. They can be combined to form a ma^k that's passed to the BWindow constructor.

See also: the BWindow constructor

window_type Constants

<interface/Window.h>

Enumerated constant	Meaning
B_SHADOWED_WINDOW	The window has a title bar and a shadowed bolder.
B_TITLED_WINDOW	The window has a title bar.
B_BORDERED_WINDOW	The window has a border but no title bar.
B_MODAL_WINDOW	The window is a modal window.
B_BACKDROP_WINDOW	The window is the backdrop for the whole screen.
B_QUERY_WINDOW	The window displays the results of a query.

These constants define the **window_type** data type. They describe the various kinds of windows that can be requested from the Application Server. Two of them, **B_BACKDROP_WINDOW** and **B_QUERY_WINDOW**, are used only by the Browser application. The others can be used by any application when constructing a window.

See also: the BWindow constructor

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" above and 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" above and **SetAlignment()** in the BTextView class

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" above and 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() global function

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" 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" 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: GetCharEscapements() and **GetFontInfo()** in the BView class

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: GetFontInfo() and **SetFontName()** in the BView class

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: SetFontName() in the BView class, **get_font_name()** global function

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: GetKeys() in the BView class, "Keyboard Information" on page 55 of 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];
} 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() global function

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 **PRIMARY_MOUSE_BUTTON**, **SECONDARY_MOUSE_BUTTON**, or **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 25 of the introduction to this chapter.

See also: "Pattern Constants" above, "Patterns" in the chapter introduction

rgb_color

<interface/InterfaceDefs.h>

typedef struct {
	uchar **red**;
	uchar **green**;
	uchar **blue**;
	uchar **alpha**;
} **rgb_color**

This type specifies a full color in the **B_RGB_24_BIT** color space. Each component, except **alpha**, 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_24_BIT**. >

See also: SetHighColor() in the BView class

screen_info

<interface/InterfaceDefs.h>

typedef struct {
	color_space **mode**;
	BRect **frame**;
	void ***bits**;
	long **bytes_per_row**;
	long **reserved**;
} **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() global function

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() global function

window_type

<interface/Window.h>

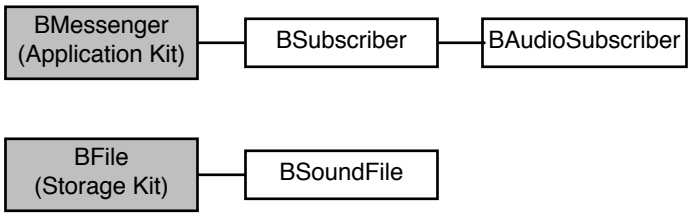
typedef enum {. . .} **window_type**

This type describes the various kinds of windows that can be requested from the Application Server.

See also: the BWindow constructor, "**window_type** Constants" 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.

Note: The Media Kit and the Midi Kit don't mix. The functionality that's currently provided by the Midi Kit will, in a future release, be subsumed by the Media Kit.

BAudioSubscriber

Derived from: public BSubscriber

Declared in: <media/AudioSubscriber.h>

Overview

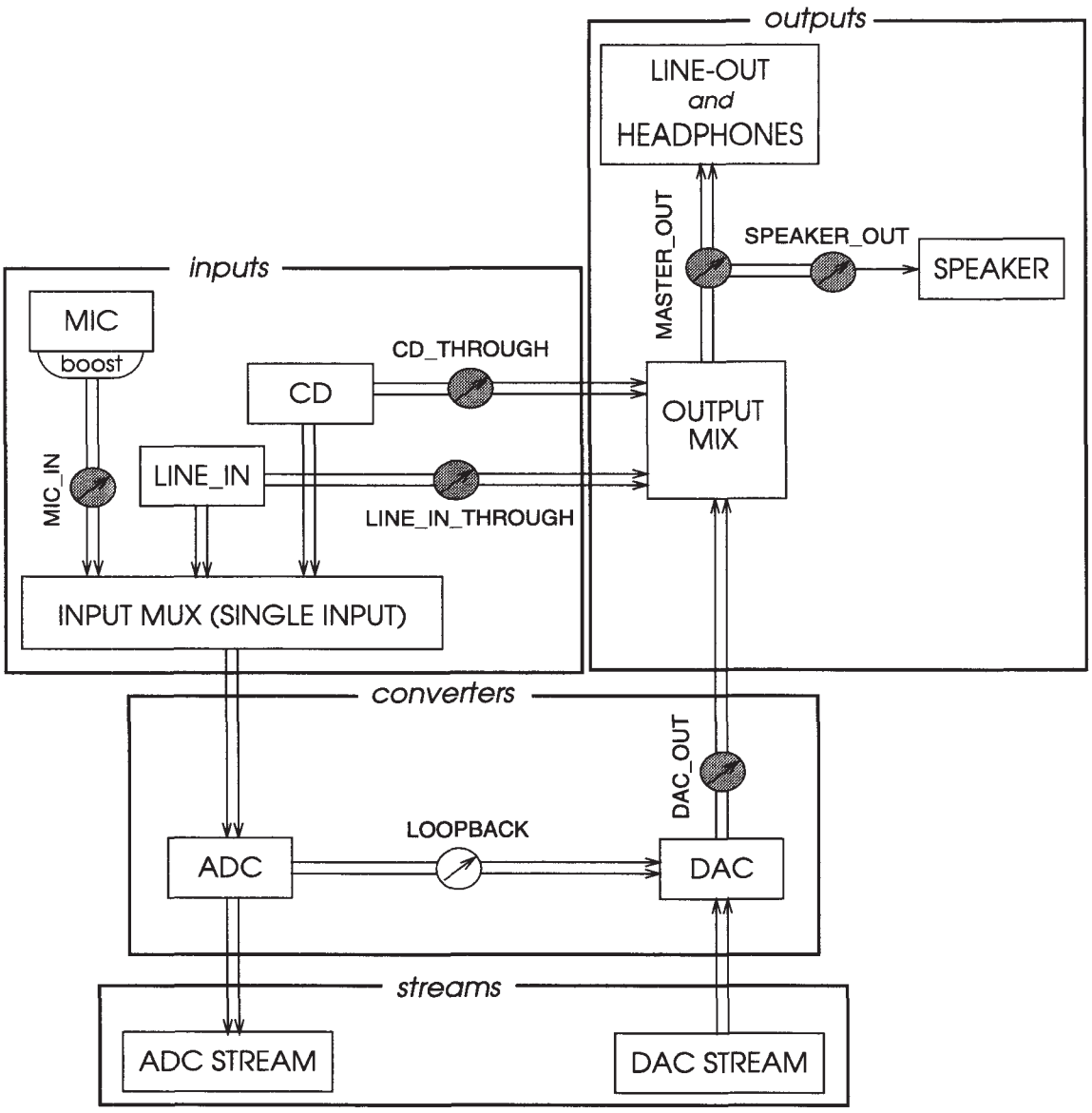
BAudioSubscriber objects perform two functions:

- They let your application receive, process, and broadcast sound data.
- They let you control certain aspects 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. A shaded circle means

the control point has a volume control *and* a mute. An unshaded circle signifies a mute but no volume control.

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 some number of candidate signals. 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 (which can be muted, but doesn't have a volume control) 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

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 (in other words, 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 microphone's 20 dB boost is toggled through the **BoostMic()** function. The state of the boost is retrieved by **IsMicBoosted()**.

Sound Data

Sound, as it appears—so to speak—in nature, is 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.

Note: Currently, one-byte linear sample data is assumed to be unsigned. The most important implication of this is that you can only play one source of 8-bit linear sound at a time. If you try to mix two 8-bit sound sources, you'll have to shift the data yourself. This will be fixed in the next release.

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 along 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 the data that was converted by the ADC.
- The buffers that flow along the DAC stream are ultimately dumped into the DAC.

Keep in mind that there's only one Audio Server-per-Be Machine; this means that there's only one sound-in and one sound-out stream, as well. So take care in receiving and manipulating the data that your BAudioSubscribers receive; any changes to the data that you make will affect all downstream subscribers.

Also note that the sound-in stream isn't automatically "connected" to the sound-out 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 stream object to the DAC stream object.

Constructor and Destructor

BAudioSubscriber()

BAudioSubscriber(char **name*)**

Creates and returns a new BAudioSubscriber object. The object is given the name that you pass as *name*; the length of the name shouldn't exceed 32 characters (this length is represented by the **B_OS_NAME_LENGTH** constant, as defined by the Operating System Kit). 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 buffer 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). 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. Your BAudioSubscriber must be subscribed to the DAC stream to successfully call these functions.

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 (directly) the sampling rate, which 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(), EnableDevice()

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 you can't set the volume of the **B_LOOPBACK** device (it doesn't have a volume control). Also, 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

A BSoundFile object can read sound data from a file. The association between a BSoundFile object and a sound file is established through the use of a record ref, as explained in the BFile documentation.

The BSoundFile functions let you examine the data in the sound file, read the data into a buffer (that you must allocate yourself), and position a "frame pointer" in the file. The frame pointer locates the first audio frame that's considered when the BSoundFile next reads the file.

Currently, BSoundFile can read AIFF and "standard" UNIX sound files. If it encounters a sound file that it doesn't understand, it assumes that the data in the file is 44.1 kHz, 16-bit stereo data, and that the file doesn't have a header (it reads from the very first byte).

Note: WAV sound file support will be added in the next release.

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()**.

Warning: Currently, only the first version of the constructor—the version that doesn't accept an argument—works. Don't try to set a BSoundFile's ref by passing the ref as an argument to the constructor.

~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 isn't associated with a file, this returns zero.

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_UNIX_FILE** and **B_UNKNOWN_FILE**. AIFF is the Apple-defined sound 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. (WAV sound file support will be added in the next release.)

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 shape this 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 also probably 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 reads of the file will retrieve the "correct" amount of data.

FrameIndex() *see* SetFrameIndex()

FramesRemaining()

long **FramesRemaining(**void**)**

Returns the number of unread frames in the file.

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, this function returns 0.

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**

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 zero.

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.

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 will only playback 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.

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*. Furthermore, each server can control more than one *buffer stream* (the Audio Server has a sound-in stream and a sound-out stream). Each BSubscriber can receive buffers from only one stream; however, more than one BSubscriber can "subscribe" to the same stream. The collection of a server's 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 stream that it wants to get buffers from.
- *Subscribing*, or applying for acceptance by the server.
- Entering the server's buffer stream. By entering the stream, the BSubscriber 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. Other (potential) single-stream servers may provide **Subscribe()** functions (in their corresponding BSubscriber-derived classes) that exclude the *stream* argument.

A BSubscriber may only subscribe to one stream at a time.

The Clique

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 the same server) must pass the same clique value (with one exception, as described later). The actual value that's used to represent the clique is irrelevant; matching is the only concern. A given clique value is enforced until all objects that subscribed with that value have *unsubscribed* (trough the **Unsubscribe()** function).

Note: The *clique* argument is type cat 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.

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.

- 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:

```
/* FirstSubscriber is assumed to be a valid BSubscriber
 * object (currently, it must be an instance of
 * BAudioSubscriber).
 */
subscriber_id firstID = FirstSubscriber->ID();
FirstSubscriber->Subscribe(firstID, ...);
```

- 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 = FirstSubscriber->ID();
FirstSubscriber->Subscribe(firstID, ...);
...

/* Notice that the second subscriber passes the
 * first subscriber's ID value as the clique argument.
 */
SecondSubscriber->Subscribe(firstID, ...);
```

  To share the stream with certain BSubscribers in other applications (or in certain other applications), the first subscriber's application would have to broadcast the first subscriber's ID value (through a BMessage, for example).

- 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*:

```
FirstSubscriber->Subscribe (B_SHARED_SUBSCRIBER_ID, ...).;
```

  Note, however, that the **B_SHARED_SUBSCRIBER_ID** clique doesn't guarantee that a BSubscriber will be allowed in the stream. If some other non-shared BSubscriber has already set the clique to some other value, a BSubscriber that passes **B_SHARED_SUBSCRIBER_ID** will be turned down.

- 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. For example, if the stream is empty and you subscribe with an **B_INVISIBLE_SUBSCRIBER_ID** clique, and then another subscriber subscribes with **B_SHARED_SUBSCRIBER_ID** while you're still in the stream, all subsequent subscribers will also have to specify **B_SHARED_SUBSCRIBER_ID**.

Waiting for Access

If a BSubscriber is denied access to a server because it didn't pass the correct clique value, it can either give up immediately, or wait for the current clique members to unsubscribe. This is expressed in the **Subscribe()**'s final argument, the boolean *willWait*:

- If *willWait* is **FALSE**, Subscriber,) 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 stream. By this, the object will begin receiving buffers of data.

Positioning your BSubscriber

The **EnterStream()** function, through which a BSubscriber enters a stream, takes a number of arguments. The first two arguments position the BSubscriber with respect to the other BSubscriber objects that are already in the stream (if any):

virtual long **EnterStream(**subscriber_id *neighbor*, bool *before*, ...**)**

The *neighbor* argument identifies the BSubscriber 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) to make room.

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 **userData*, StreamFn *streamFunction*, ..., bool *background***)**

The arguments, taken out of order, are:

- *streamFunction* is a pointer to a boolean-returning function (the complete protocol is given below) that will be invoked once for each buffer that's received.
- *userData* is a pointer-sized value that will be passed as an argument to *streamFunction*.
- The value of *background* is used to determine whether *streamFunction* 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 "stream function" that you must supply: This is global function that's invoked once for each buffer that the BSubscriber receives. The protocol for the function (which is **typedef**'d as **StreamFn**) is:

bool ***stream_function*(**void **userData*, char **buffer*, long *count***)**

- *userData* is taken from the userData argument 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 stream function yourself; the Media Kit doesn't supply any stream function 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 hard 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 stream function. 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.

Note: Although the stream function must be global, it's often convenient to use an object to manage the actual data-processing. See "Processing Data in a Member Function" on page 27 for details.

Exiting the Stream

There are two (unexceptional) ways to remove a BSubscriber from a buffer 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 want to pull the trigger from the main thread.

Whichever method is used, the BSubscriber's "completion function" 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(..., CompletionFn *completionFunction*, ...**)**

The protocol for the completion function is:

long ***completion_function(***void **userData*, long *error***)**

- The *userData* 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. If you're running in the background, this clean-up can include deleting the BSubscriber object itself (if the function knows the identity of the object). But don't try this trick if you're not running in the background; such a maneuver will void the warranty on your mattress.

Processing Data in a Member Function

As mentioned above, the stream function must be a global function. But it's easier to create a class that will perform the actual data-processing. This is where the *userData* argument comes in: The argument is provided, primarily, so you can identify the object that will perform the processing. The implementation of the stream function would, then, look like something this:

```
bool my_stream_function(void *userData,
                  char *buffer,
                  long count)
{
/* Cast userData to the proper class (we'll call it
 * SoundMaker for this example), and invoke the object's
 * data-processing function (we'll call it MakeSound()),
 * passing along the buffer and byte-count.
 */
return (((SoundMaker *)userData)->MakeSound(buffer,count));
}
```

This implies, of course, that you must pass an existing object as the *userData* argument when you call **EnterStream()**:

```
/* Create a SoundMaker and pass it to EnterStream(). */
SoundMaker *beeper = new SoundMaker();
...

/* MySubscriber is assumed to be an instance of a
 * BSubscriber-derived class that has successfully
 * subscribed to the server.
 */
MySubscriber->EnterStream(NULL, TRUE,
                  beeper, my_stream_function, ...);
```

This methodology works equally well for the completion function.

Constructor and Destructor

BSubscriber()

BSubscriber(char **name***name***)**

Creates and returns a new BSubscriber object. The object is given the name that you pass as *name*; the length of the name shouldn't exceed 32 characters (this length is represented by the **B_OS_NAME_LENGTH** constant, as defined by the Operating System Kit). The name is provided as a convenience and needn't be unique (across the set of all BSubscribers).

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 can delete a BSubscriber from within an implementation of the object's "completion function" (as defined by the **EnterStream()** function), but only if the function is executed in a background thread.

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 **userData*,
StreamFn *streamFunction*,
CompletionFn *completionFunction*,
bool *background***)**

Causes the BSubscriber to begin receiving buffers of data from the media server. 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**.
- *userData* 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 **userData*, char **buffer*, long *count***)**

  The *userData* argument, here, is taken literally as the *userData* 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 **userData*, long *error***)**

  userData, 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. The value returned by the completion function becomes the value that's returned by **EnterStream()** (if the function isn't running in the background, as described in the next argument.)

  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 **SUBSCRIBER_NOT_FOUND**. Otherwise, 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 (**B_NO_ERROR** or otherwise) that indicates the success of the communication with the server (unless something's gone wrong, the return, in this case, should be **B_NO_ERROR**). In all cases, the **Error()** value is set to the value returned here.

Calling **EnterStream()** while you're already in the stream isn't disallowed. But be aware of the consequences: If the object is in the stream, **ExitStream()** is invoked automatically before the second and all subsequent calls to **EnterStream()**.

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(**void**)**

Causes the BSubscriber to leave the buffer stream after it completes the processing of its current buffer. This function always returns **B_NO_ERROR** (and sets **Error()**) to do the same.

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 buffer 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.

Name()

long **Name(**subscriber_id *id*, char **name***name***)**

Returns, by reference in *name*, the name of the subscriber identified by *id*. If *id* is **NULL**, the name of the calling BSubscriber is returned.

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 has successfully 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_RESOURCE_UNAVAILABLE** 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 set these values to reasonable 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 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 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 is running (it's "streaming"). You should only need to call **StartStreaming()** if a previous invocation of **StopStreaming()** was made.

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.
- 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* SefTimeout()

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 the server's buffer stream, it's removed and the object's "completion function" (as set by the **EnterStream()** function) is called. When you delete a BSubscriber, it's automatically unsubscribed.

The function returns **B_NO_ERROR** if the object was successfully unsubscribed. Other error codes signify a problem in communicating with the server. The **Error()** value is set to the value returned directly here.

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 **sound_wait()**. This will cause your thread to wait until the sound has finished playing.

beep() will mix other sounds, but it never waits if it 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 file's data portion mustn't already be open.

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()**.

sound_stop()

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()**.

wait_for_sound()

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()**.

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.

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.

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

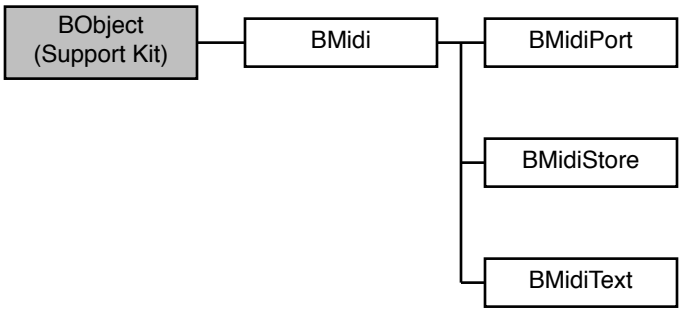
<media/MediaDefs.h>

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 Be Computer 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, unfortunately, 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 defines the fundamental concepts of the Kit by providing the mechanisms and 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 function allow instances of your class to behave in a fashion that the other objects will understand.

The functions that BMidi declares 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 type. 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 object responds depends on the object's class—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.

For example, let's say you want to record a MIDI performance—in other words, you want to connect a 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 (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) ;
```

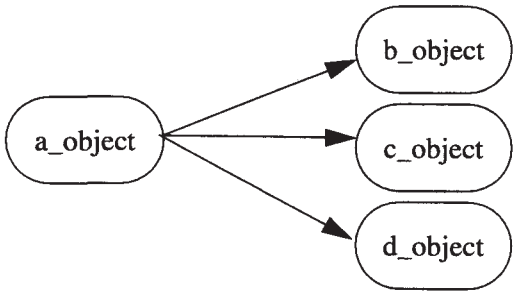
Simply connecting the objects isn't enough, however; you have to tell the BMidiPort to start listening to the MIDI port, by calling its **Start()** function. This is explained in a later section.

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:



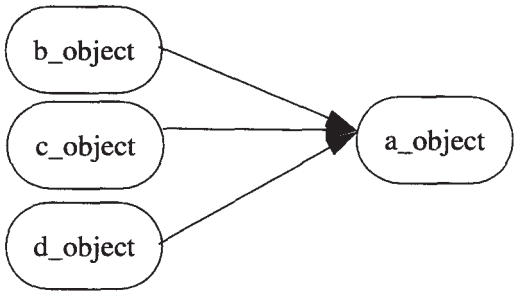
The configuration in the illustration is created thus:

```
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)
```



a_object *doesn't* know about the objects that are connected to its input. In other words, when you use a BMidi object as the argument to a **Connect()** method, the argument object *isn't* informed.

Message Generation and the Run() Function

To generate MIDI data, you implement, in a BMidi-derived class, the **Run()** function. An implementation of **Run()** should include a **while** loop that, on each circuit, produces the data for (ideally) a single MIDI message, 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 data and spray. */
    }
}
```

To tell an object to perform its **Run()** function, you call the object's **Start()** function—you never call **Run()** directly. This 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).

As with **Run()**, you never call spray functions directly; they should only be called within the implementation of a BMidi-derived class.

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 hook (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 automatically called 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 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, 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

As pointed out earlier, 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, operating system-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 (it calls the **snooze()** function) until the designated tick is at hand.

Spraying Time and the B_NOW Macro

If you're implementing the **Run()** function, then you have to generate a time value yourself (to pass as the final argument to a spray function). The value you generate depends on what your class is doing:

- 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()**/l000.0 (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.

- 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 would 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 runningTime so the next Note On
       * will be 100 ticks after this one.
       */
      when += 1;
  }
}
```

  When a MyTicker object is told to start running, it will busily churn out notes as fast as possible and rely on its connected objects to apply the time-stamps in their input functions.

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 15.

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* 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: the BMidi destructor, **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 "Message Generation and die **Run()** Function" 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 description 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***)**

BMidiPort

Derived from: public BObject

Declared in: <midi/MidiPort.h>

Overview

The BMidiPort class provides the mechanisms for reading MIDI data that appears at the MIDI-In port, and for writing MIDI data to the MIDI-Out port. Although the BeBox has two pairs of MIDI-In and MIDI-Out hardware ports, BMidiPort objects only read from the "A" set. These are the top two MIDI ports on the back of the computer: MIDI-In A is the top port, MIDI-Out A is immediately below.

You can create and use any number of BMidiPort objects in your application. The immutable number of hardware MIDI ports doesn't dictate the number of objects.

Opening the Ports

To obtain data from the MIDI-In port or send data to the MIDI-Out port, you must first open the ports. BMidiPort's **Open()** function opens both ports. The function's single argument is a string that names the in/out pair that you're opening. The two pairs of MIDI ports are named "midi1" and "midi2"; thus, currently, the argument must be "midi1":

```
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 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 construction of the object: 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. This function always opens a MIDI-In and a MIDI-Out port; currently, the only pair you can open are identified as "midi1". 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 **NOW**. Since **NOW** is a shorthand for "the current tick," and since time tends to move forward at a reasonably steady rate, 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 proceed 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.

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. If you want to edit a 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 **OpenData()** 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's data portion 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 Program Change 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 moire information on time).

SystemExclusive()

virtual void **SystemExclusive(**void **data*,
long *data_length*,
ulong *time* = B_NOW**)**

Responds to a Program Change 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 Program Change 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 Kernel Kit

7 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*. A thread is a synchronous computer process. By creating multiple threads, you can make your application perform different tasks at (virtually) the same time.
- *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, depending on its type, can be executed or 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.

The rest of this chapter describes these topics in detail. The final two sections ("Miscellaneous Functions" and "Constants, Defined Types, and Structures") tie up the loose ends and describe the associated API—constants, macros, data types, and so on—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.

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. 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. 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.

The following sections describe how to spawn, control, and examine threads and teams.

Spawning and Running 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;

if ((my_thread = spawn_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.

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.

Of these two functions, **resume_thread()** is the more common means for starting a thread. For more information on **wait_for_thread()**, see the function description on page 22; for the balance of the current topic, **wait_for_thread()** is ignored.

The **spawn_thread()** and **resume_thread()** function calls are often nested:

```
if (resume_thread(my_thread = spawn_thread(...)) < B_NO_ERROR)
    /* failure, in either spawning or running. */
else
    /* the thread was successfully spawned and is running. */
```

There are, however, situations in which you may need or want to separate the two calls. One such situation, in which data is sent to the thread before the thread is told to run, is described in "Passing Data to a Thread" on page 13. Separating the calls can also improve your application's response to the user: **spawn_thread()** involves some overhead that you may want to incur while your application is being set up (for example), rather than when the thread begins performing.

Loading an Executable

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. The program is loaded in a separate thread that's spawned (and returned) by the **iood_executable()** call. An important difference between **spawn_thread()** and **load_executable()** is that the latter creates a new team for the thread it spawns.

As with **spawn_thread()**, a call to **load_executable()** must be followed by **resume_thread()** or **wait_for_thread()**—the new thread isn't run until one of the latter functions is called.

For more information on the **load_executable()** function, see "Images" beginning on page 49.

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 would call **spawn_thread()** and **resume_thread()** thus:

```
/* Spawn the thread; the final argument--a BList object that's
 * assumed to exist--will be passed as the argument to the
 * lister() function when the thread starts running.
 */
if ((my_thread = spawn_thread(lister, ..., (void *)listObj))
        < B_NO_ERROR)
    /* failure */

/* Run the thread. */
if (resume_thread(my_thread) < B_NO_ERROR)
    /* failure */

. . .
```

The Entry Function's Argument

With regard to the entry function's data argument, there are three points worth noting:

- Although the *data* argument can point to any amount and type of data, the argument is almost always a pointer to an object (as shown above). Passing an object is particularly useful if you want to redirect the entry function to a member function of that object. This is explained in the next section.

- The **spawn_thread()** function *doesn't* copy the *data* that data points to. Changes to the pointed-to data that are made between the **spawn_thread()** and **resume_thread()** calls will be seen by the entry function.

- Because of the no-copy condition, you should never pass a pointer to data that's allocated locally (on the stack). Spoken emphatically and as a general rule,

  An entry function's argument should point to global data.

  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 that data vanishes. There are ways around this restriction—for example, you could use a semaphore to ensure that the entry function has copied the argument before the calling frame exits. A better solution, if you absolutely must pass locally-allocated data, is to use the inter-thread data-sending mechanism described in the section "Passing Data to a Thread" on page 13.

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:

```
/* Our MyList class derives from BList.  It contains two
 * functions--the static FakeLister(), and the non-static
 * Lister().  The latter will be invoked by the former.
 */
class MyList : BList {
public:
    static long FakeLister(void *arg);
    long Lister();
};
```

```
/* FakeLister() will be used as an entry function.  But it
 * doesn't really do anything--it simply casts its argument as
 * a MyList object, and then invokes the "real" entry
 * function, Lister(), upon that object.
 */
long MyList::FakeLister(void *arg)
{
    MyList *obj = MyList *arg;
    return (obj->Lister ( )) ;
}

/* Lister() performs the actual work.  Notice that it doesn't
 * have to adhere to the entry function protocol (since it
 * isn't going to spawn_thread()).
 */
long MyList::Lister()
{
    /* do something here */
    . . .
    return (whatever);
}
```

The **spawn_thread()** call for this set up would look like this:

```
spawn_thread(MyList::FakeLister, ..., (void *)MyListObj)
```

Again, the final argument (which will be forwarded as the *data* argument to the entry function) must already exist, and should be global. To fit in this example, the *MyListObj* object should be an instance of the MyList class.

Note: If you aren't familiar with static member functions, you should consult a 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 only 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 other_thread_return;

/* We'll dispense with error checks for this example. */
other_thread = spawn_thread(...);
resume_thread(other_thread);
```

```
. . .
wait_for_thread(other_thread, &other_thread_return);
```

The system doesn't cache a thread's return value in anticipation of a subsequent wait on that thread: If the target thread is already dead, **wait_for_thread()** will return 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: Currently, 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. Also, you mustn't wait for the thread that's calling **wait_for_thread()**.

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. (Currently, there's no way to retrieve the second and subsequent identically-named threads.)

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 function description for **get_thread_info()** on page 16.

Dissatisfied with a thread's name? Use the **rename_thread()** function to change it. Fool your friends.

Thread Priority

To provide 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 as the third argument to **spawn_thread()**. There are four priority constants that you can use for this assignation (listed here from least to most attention):

- **B_LOW_PRIORITY** represents the lowest priority. It's meant for threads that don't need much attention and that certainly don't want to interrupt other threads to get what little attention they deserve.
- **B_NORMAL_PRIORITY** is the most common priority—this is the priority used for all main threads, for example. If your thread isn't controlling a user interface object, and doesn't need real-time interaction or response:, you should set it to this priority.
- **B_DISPLAY_PRIORITY** is used for threads that control user interface objects. For example, the thread that's spawned when you create a BWindow object is given display priority. If your thread needs to compete with the active window (which should be rare given the objects that are provided by the Interface Kit), then use this priority.
- **B_REALTIME_PRIORITY** is the highest priority. It should only be used by threads that need as much attention as possible, even if this means degrading the responsiveness of the user interface. The only threads that should consider this priority are those that control real-time processes, such as music synthesis that's driven by user events. The Media Kit makes use of this priority when it spawns a thread that runs a "stream function."

Although higher priority threads get more attention than those with lower priority, the system tries to be fair about scheduling. If a thread is starving for attention, the system will occasionally throw it a few cycles even if there are higher-priority threads that are ready to run.

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 29.
- Synchronization is sometimes a side-effect of sending data between threads. This is explained in "Passing Data to a Thread" on page 13, 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 a thread—any thread—through the **kill_thread()** function. You should only need to do this in exceptional cases; keep in mind that a thread will die a natural death when it reaches the end of its entry function. If you're feeling particularly frustrated, try killing an entire team of threads: The **kill_team()** function is just such a purgative.

As mentioned earlier, the control that's visited upon a thread doesn't influence the "children" that have been spawned from that thread. For example, consider the following:

```
/* Here, the main thread spawns say_oh which spawns say_ah.
 */
long say_ah(void *data)
{
    while (1)
        printf("ah");
    return 0;
}

long say_oh(void *data)
{
    resume__thread (spawn_thread (say_ah, ...));
    while (1)
    printf("oh");
    return 0;
}

main()
{
    thread__id oh_thread;
    oh_thread = resume_thread(spawn_thread(say_oh,...))
    snooze(1000000);
    kill_thread(oh_thread);
}
```

The main thread spawns (and runs) **say_oh()**, which spawns and runs **say_ah()** and then starts spitting out the word "oh". **say_ah()**, in the meantime, is printing the word "ah". The main loop snoozes for one second before killing the "oh"-printing thread. The three important points, here, are:

- The main thread's **snooze()** call doesn't affect the other threads; they continue printing their "ah"s and "oh"s while the main thread sleeps.
- The assassination of the "oh" thread doesn't affect the "ah" thread, even though "ah" was spawned from "oh".
- Similarly, the death of the main thread (as the **main()** function exits) doesn't stop the "ah" thread.

The last point is worth keeping in mind: The death of the main thread doesn't cause the other threads in that team to die. All other threads continue until they reach the ends of their entry functions, or until some other force kills them (such as the **kill** *thread_id* command-line program). However, when the main thread goes down it takes the team's heap and all its statically allocated objects (among other team-wide resources) with it. Threads that linger beyond the death of the main thread may be seriously crippled.

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" beginning 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 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,
- and the size of the buffer of data, in bytes.

In the following example, the main thread spawns a thread and then sends it some data:

```
main(int argc, char *argv[])
{
    thread_id otherthread;

    /* Spawn the thread. */
    other_thread = spawn_thread(a_func, "Other thread",
```

```
            B_NORMAL_PRIORITY, NULL);

        /* Send some data. */
        if (send_data(other_thread, 63, "Hello", 5) < B_NO_ERROR)
            /* failure */
        else:
            /* success */
        ...

        /* Now start the other thread. */
        if (resume_thread(other_thread) < B_NO_ERROR)
            ...
    }
```

The **send_data()** call copies the code and the buffer (the second and third arguments) into the target thread's "data 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 contents of the data buffer into its second argument. It also returns, by reference in its first argument, the **thread_id** of the thread that sent the data (in other words, the thread in which **send_data()** was called). This is demonstrated in the example implementation of **my_func()**, below:

```
long my_func (void *data)
{
    thread_id sender;
    long code;
    char buf[512];

    /* The last argument to receive_data() gives the size
     * of the buffer into which the data is copied.
     */
    code = receive_data(&sender, (void *)buf, 512);
    . . .
}
```

A slightly annoying aspect of this mechanism is that there isn't any way for the data-receiving thread to determine how much data has been sent. If the buffer that the receiver allocates isn't big enough to accommodate all the data, the left-over portion is simply thrown away.

As shown in the examples, **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 in the implementation of **my_func()**). However, **send_data()** isn't limited to this use—you can also send data to a thread that's already running.

A thread's data buffer isn't a queue; it can only hold one data-transmission 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 never 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 any data waiting to be read:

```
if (has_data(find_thread(NULL)))
    /* The calling thread has some unread data, so the
     * receive_data() call won't block.
     */
    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))
    /* The target doesn't have any unread data, so the
     * send_data() call won't block.
     */
    send_data(...);
```

This usually works, but be aware that there's a race condition between the **has_data()** and **send_data()** calls. If some other thread sends data to your target in that time interval, your **send_data()** (might) block.

Functions

find_thread()

thread_id **find_thread(**const char **name***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, the **B_NAME_NOT_FOUND** error constant 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***info***)**
long **get_nth_team_info(**long *n*, team_info **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 object_count;
        long thread_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
} team_info
```

The structure's fields are described below:

Field	Meaning
team	The **team_id** of this team.
object_count	The number of "objects" or executable files (including libraries) that are loaded in the team.
thread_count	The number of threads that comprise the team.
area_count	The number of areas that the team can reference.
debugger_nub_thread	A thread that's used by the debugger to execute operations on this team.
debugger_nub_port	A port that's used by the debugger to communicate with this team.

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;
        int priority;
        sem_id sem;
        double time;
        char *stack_base;
        char *stack_end;
} thread_info
```

The fields in the structure are:

Field	Meaning
id	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. (The thread state constants are listed in the next table.)
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 time, in microseconds, the thread has spent being attended to by the CPUs.
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.

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 bounds—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 any unread data that's been sent to it through a previous **send_data()** call; otherwise returns **FALSE**.

See also: send_data(), receive_data()

kill_team()

long **kill_team(**team_id *team***)**

Kills all the threads in the given team. You should only call this function as a last resort; killing a team can leave your system in an ugly state. If the *team* argument isn't valid, **B_BAD_TEAM_ID** is returned. Otherwise, the function returns **B_NO_ERROR**.

kill_thread()

long **kill_thread(**thread_id *thread***)**

Kills the given thread. To kill the calling thread, use the construction

```
kill_thread(find_thread(NULL))
```

If the *thread* argument isn't valid, **B_BAD_THREAD_ID** is returned. Otherwise, the function returns **B_NO_ERROR**.

receive_data()

long **receive_data(**thread_id **sender*,
void **buf*,
long *buf_size***)**

Receives data that's been sent to the thread through a previous **send_data()** function call. Typically, and most usefully, the **send_data()** call will have been made from another thread.

The data that's been sent is copied into the buffer pointed to by *buf*. The *buf_size* argument tells the function how many bytes of data to copy. If you don't want to receive any data—if the value returned directly by the function, as described below, is sufficient—you set *buf* to **NULL** and *buf_size* to 0.

The **thread_id** of the thread that sent the data (in other words, the thread that called **send_data()**) is returned by reference in the *sender* argument. The value that's returned directly by the function is the value that was passed as the *code* argument to **send_data()**.

Each **receive_data()** call matches exactly one **send_data()** call: If **send_data()** sent more data than *buf* can accommodate, the unaccommodated portion is discarded—a second **receive_data()** call will not read the "rest" of the data. Conversely, if **receive_data()** asks for more data than was sent, the function returns with the excess portion of *buf* unmodified—**receive_data()** doesn't wait for another **send_data()** call to provide more data with which to fill up the buffer.

If there isn't any data to receive, **receive_data()** blocks until there is some. In some instances, you may want to call **has_data()** as a predicate for the call to **receive_data()**.

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 **THREAD_SUSPENDED** (newly spawned threads are born with this state). You can't use this function to resume a thread that's sleeping (**B_THREAD_ASLEEP** status), waiting to acquire a semaphore (**B_THREAD_WAITING**), or waiting for a message (**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. 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***)**

Sends information to the thread given in the *thread* argument (this is the "target" thread). There are two pants to the information that you send:

- You can send a single **long**-sized datum through the code argument.
- You can send a variable-length buffer of data through *buffer*. The length of the buffer, in bytes, is given by *buffer_size*.

If you only need to send the code datum, you should set *buffer* to **NULL** and *buffer_size* to 0. Note that **send_data()** copies the data that's pointed to by *buffer*. Changes that you make to buffer after **send_data()** returns won't affect the data that's received by the target thread.

The information that you send through **send_data()** is retrieved by the target thread when it (the target) calls the **receive_data()** function. Normally, **send_data()** returns immediately (after copying the data in *buffer*)—it doesn't wait for the target to call **receive_data()**. However, **send_data()** will block (and the calling thread will assume the **B_THREAD_WAITING** status) if the target has data that it hasn't yet read.

If sufficient memory in which to copy the data in *buffer* couldn't be allocated, this function fails and returns **NO_MEMORY**. If *thread* doesn't identify a valid thread, **BAD_THREAD_ID** is returned. Otherwise, the function succeeds and returns **NO_ERROR**.

See also: receive_data(), has_data()

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):

Priority Constant	Use
B_LOW_PRIORITY	Non-crucial computation
B_NORMAL_PRIORITY	The default priority
B_DISPLAY_PRIORITY	Threads that control interlace objects
B_REALTIME_PRIORITY	Threads that need real-time response

  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 **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.

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. When the target thread is dead, the value that was returned by its entry function is returned by reference in *exit_value*. If the target thread's entry function didn't exit naturally—if the target was assassinated through a **kill_thread()** call, for example—the value returned in *exit_value* will be unreliable (unfortunately, there's currently no way to tell how the target died).

If the target thread has already exited or is otherwise invalid, this function returns **B_BAD_THREAD_ID**, otherwise it returns **B_NO_ERROR**. If the thread is killed while you're waiting for it, the function returns **B_NO_ERROR** (but, as noted above, the value in *exit_value* will be unreliable).

Note: An invalid *thread* argument will, in certain circumstances, cause the function to return **B_ERROR**. This will be corrected in the next release (such that a bad thread will always return **B_BAD_THREAD_ID**).

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 (or, less usefully, the same thread) can then retrieve the buffer. This repository is implemented as a first-in/first-out 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.

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

Each 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 thread doesn't "own" the ports that it creates. Most significantly, the ports that a thread creates aren't freed when the thread is killed. The operating system provides a limited number of port identifiers, so it's important that you delete your ports (through the **delete_port()** function) when they're no longer needed.

The Message Queue

The functions **write_port()** and **read_port()** send information through a port by placing and removing messages in the port's message queue. Technically, **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.

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. **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. As a convenience, the **B_MAX_PORT_COUNT** constant provides a reasonable queue length (although see the warning, below).

Note: When a thread is waiting in a **write_port()** or **read_port()** call, its state is **B_THREAD_SEM_WAIT**—in other words, it's waiting to acquire a (system-defined) semaphore. For each port, there are two such semaphores: one for reading and another for writing. Both semaphores are given the same name as the port. See "Threads and Teams" on page 5 for more information about thread state.

Warning: 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.

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 integer) 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 code and message buffer are set and retrieved as separate arguments to the **write_port()** and **read_port()** functions; see the function descriptions, below, for the precise protocol.

When you read a port, you have to supply a buffer into which the port mechanism can copy the data from the message buffer in the port's queue. 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 */
```

Message Ownership

Just as ports aren't owned by the threads that create them, neither are messages owned by the threads that place them in a port's queue. For example, if you write a message to a port and then kill the thread that wrote the message, the message remains in the port's queue. Furthermore, you can't "erase" a message once it's been written to a port—the only way to remove a message from a port queue is to read it.

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 for the additional messages. The minimum queue length is one. 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.

The function returns **B_BAD_ARG_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.

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—however, you can't delete a port if there are any threads that are blocked in **read_port()** or **write_port()** calls on the port.

The function returns **B_BAD_PORT_ID** if *port* isn't a valid port; it returns **B_PORT_BUSY** if there are any blocked threads waiting to use it. Otherwise it returns **B_NO_ERROR**.

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.

get_port_name()

long **get_port_name(**port_id *port*, char **name***)**

Copies the port's name, as assigned by the **create_port()** function, into *name*. The *name* should be at least **B_OS_NAME_LENGTH** bytes long. If the port doesn't have a name, the *name* argument is returned as a **NULL** string.

If port is a valid port identifier, the function returns **B_NO_ERROR**; otherwise, **B_BAD_PORT_ID** is returned.

port_buffer_size()

long **port_buffer_size(**port_id *port***)**

Returns the length of the message buffer for the message that's at the head of *port*'s queue. You call this function in order to allocate a sufficiently large buffer in which to retrieve the message data. As with **read_port()**, this function blocks if the port is currently empty.

If *port* doesn't identify an existing port, **B_BAD_PORT_ID** is returned.

port_count()

long **port_count(**port_id *port***)**

Returns the number of messages that are currently in *port*'s message queue. If *port* isn't a valid port identifier, **B_BAD_PORT_ID** is returned.

read_port()

long **read_port(**port_id *port*,
long **msg_code*,
void **msg_buf*,
long *buf_size***)**

Removes the message at the head of *port*'s message queue and returns its contents in the *msg_code* and *msg_buf* arguments. The size of the *msg_buf* buffer, in bytes, is given in *buf_size*. See the **write_port()** function for more information on the final three arguments.

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()**.

The function returns **B_BAD_PORT_ID** if *port* isn't valid, otherwise it returns **B_NO_ERROR**. (Note that if the port is deleted while this function is blocked, the function will immediately return **B_BAD_PORT_ID**.)

write_port()

long **write_port(**port_id *port*,
long *msg_code*,
void **msg_buf*,
long *buf_size***)**

Places a message at the tail of port's message queue. The message consists of *msg_code* and *msg_buf*:

- *msg_code* holds the message code. This is a mask, flag, or other predictable value that gives a general representation of the message.
- *msg_buf* 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 *buf_size* argument. The buffer can be arbitrarily long.

If the port's queue is full when you call **write_port()**, the junction will block. It returns when a **read_port()** call frees a slot in the queue for the new message.

The function returns **B_BAD_PORT_ID** if *port* isn't valid, otherwise it returns **B_NO_ERROR**. (Note that if the port is deleted while this function is blocked, the function will immediately return **B_BAD_PORT_ID**.)

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 alternative function, **acquire_sem_timeout()** 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_timeout()** 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 30). 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, however, 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 33.

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 queue. Each call to **release_sem()** "releases" the thread at the head of that semaphore's queue (if there are any waiting threads), 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 be released.

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 and so the function returns immediately, allowing the thread to continue (in other words, without having to pass through the queue). 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 semaphore'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 exemplified 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.

You can get a semaphore's thread count by calling **get_sem_count()**; the count is returned by reference in the second argument.

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. The operating system can support only a limited number of semaphores at a time, so it's important that you delete your semaphores when you're finished with them.

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, 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 zero, 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_count(), acquire_sem_timeout()

long **acquire_sem(**sem_id *sem***)**

long **acquire_sem_count(**sem_id *sem*, long *count***)**

long **acquire_sem_timeout(**sem_id *sem*, double *timeout***)**

These functions attempt to acquire the semaphore identified by the *sem* argument. Except in the case of an error, **acquire_sem()** and **acquire_sem_count()** don't return until the semaphore has actually been acquired. **acquire_sem_timeout()** provides a timeout facility: If the semaphore hasn't been acquired within *timeout* microseconds, the function returns anyway.

The **acquire_sem()** and **acquire_sem_timeout()** functions acquire the semaphore but once (they decrement the thread count by one); **acquire_sem_count()** attempts to acquire the semaphore *count* times (it decrements the thread count by *count*).

Other than the timeout and the acquisition count, there's no difference between the three acquisition functions. Specifically, any semaphore can be acquired through any of these functions; you always release a semaphore through **release_sem()** (or **release_sem_count()**) 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 **acquire_sem_timeout()** surpasses its designated time limit, it returns **B_TIMED_OUT**.

If the semaphore is successfully acquired, the functions return **B_NO_ERROR**.

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. */
```

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.

The operating system allows only a limited number of semaphores at a time. Because of this, you should delete your semaphores when you're finished with them.

delete_sem()

long **delete_sem(**sem_id *sem***)**

Deletes the semaphore identified by the argument and marks the identifier as invalid. If there are any threads waiting in the semaphore's thread queue, they're immediately de-queued and so allowed to continue execution.

If *sem* is a valid semaphore identifier, this function returns **B_NO_ERROR**; otherwise it returns **B_BAD_SEM_ID**.

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.

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).

get_sem_name()

long **get_sem_name(**sem_id *sem*, char **name***)*

Copies the semaphore's name, as assigned by the **create_sem()** function, into *name*. The *name* buffer should be at least **B_OS_NAME_LENGTH** (32) characters long. If the semaphore doesn't have a *name*, the name argument is returned as a **NULL** string.

A semaphore's name is only meant to be used as a debugging aid.

If *sem* is a valid semaphore identifier, the function returns **B_NO_ERROR**; otherwise, **B_BAD_SEM_ID** is returned.

release_sem(), release_sem_count()

long **release_sem(**sem_id *sem***)**

long **release_sem_count(**sem_id *sem*, long *count***)**

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_count()** does the same, but for *count* threads.

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_ARG_VALUE** otherwise.

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 locations that comprise it 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 the CPU's on-chip memory (or "core") 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 core.

Identifying an Area

An area is uniquely identified (across your computer) by its **area_id** number. The **area_id** is assigned automatically by **create_area()**, a function that does what it says. Most of the area functions require an **area_id** argument.

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()**; the function returns an **area_id** number based on the area name argument. Note, however that area names are rot unique—any number of areas can be assigned the same name (the assignment is made when the area is created).

Because area names aren't unique, they should be considered to be "one-time" identifiers only. Once you've gotten an **area_id** value through the **find_area()** function, all subsequent area calls that refer to that area should be done on the basis of the ID—you shouldn't call **find_area()** each time. 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 application is running on your computer).

You can also find areas through the **area_at()** function. The function takes an index argument that's used to locate and return the area in the system's list of areas. This list includes *all* areas—not just the areas that have been created by the calling thread (or team. **area_at()** is intended to be used for system diagnostics; it isn't meant as a means for communicating ;areas between applications (for example).

Using an Area

Ultimately, you use an area for the virtual memory that it represents. In other words, 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.

Cloning an Area

If you want to read or write another area's data, the first thing you should do, having acquired an **area_info**, is "clone" the area. You do this by calling the **clone_area()** function. The function returns a new **area_info** number that identifies your clone of the original area. All further references to the area should be based on the ID of the clone.

Areas have no concept of "ownership." An area created through **clone_area()** is just as valid as one crested through **create_area()**. The significance of this is particularly felt when you delete your areas: The memory that underlies an area isn't freed until all areas that refer to it have been deleted. Thus, if you clone an area and then the original area is deleted, your clone will still be valid—the memory that it refers to won't be yanked out

from underneath it. The areas that you create in your application are automatically deleted when your application exits. To force a deletion, you can call **delete_area()**.

The physical memory that lies beneath a cloned area is never implicitly copied—the area mechanism doesn't perform a "copy-on-write" (for example). If two areas refer to the same memory because of cloning, a data modification that's affected through one area will be seen by the other area.

Functions

area_at()

area_id **area_at(**long *index***)**

Returns the **area_id** of the area that resides at the *index*'th slot in the system's list of areas. This list includes all areas that the system has allocated, not just those areas that "belong" to the calling thread (or to its team). Valid index values start at zero; if *index* is out-of-bounds, **B_ERROR** is returned.

This function is provided to aid system diagnostics; you shouldn't use it to "find" a specific area. Typically, you would use this function if you want to examine the paging and memory use statistics for each area (these statistics are provided by the final four fields of the area_info structure).

See also: get_area_info(). find_area(), area_for()

area_for()

area_id **area_for(**void **addr***)**

Returns the **area_id** of the area that contains the given address. The argument needn't be the starting address of an area, nor must it start on a page; boundary. If it lies anywhere within an area, the ID of that area is returned.

The address is taken to be in the local address space; accordingly, 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. Note well that this argument doesn't identify the source area. Area names are, at most, **B_OS_NAME_LENGTH** (32) characters long.
- *clone_addr* points to the address at which you want the clone area to start; it 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 the same meaning as they do for **create_area()** (for more information, see the **create_area()** description). 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 *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.

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.

create_area()

area_id **create_area(**const char **name*,
void ***addr*,
ulong *addr_spec*,
long *size*, ulong *lock*,
ulong *protection***)**

Creates an 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 **addr* is ignored; the starting address is determined by the system.

  (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 core, if the area is 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 core and won't be swapped out.

  B_LAZY_LOCK allows individual pages of memory to be brought into core 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.

The areas you create should eventually be deleted through the **delete_area()** function. Undeleted areas are automatically deleted when your application exits.

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_VA1UE	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.

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. If *area* doesn't designate an actual area, this function returns **B_ERROR**; otherwise it returns **B_NO_ERROR**.

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**.

If the argument doesn't identify an existing area, the **B_NAME_NOT_FOUND** error code is returned.

See also: area_for(), **area_at()**

get_area_info()

long **get_area_info(**area_id *area*, area_info **info***)*

Copies information about *area* into the **area_info** structure designated by *info*. The structure is defined as:

```
typedef struct area_info {
        area_id area;
        char name[B_OS_NAME_LENGTH];
        void *address;
        long size;
        ulong lock;
        ulong protection;
        team_id team;
        long ram_size;
        long copy_count;
        long in_count;
        long 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.
- **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 and 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 total number of times that any of the pages in the area have been copied because a process has written to them (in other words, it's a "copy-on-write" count).
- **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*, long *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 the old; 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.

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. Note that an area's protection only applies to access to the underlying memory through that specific area (more specifically, it applies to addresses that lie within the area). Different clones of 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**.

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, obviously, has a single app image.
- A *library image* is a dynamically linked library (DLL). Most applications link with the two system libraries (**libbe.so** and **libc.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, and how to load an add-on image (instructions for creating a DLL will be published later).

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.

To load an app image, you call the **load_executable()** function. 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 returned thread won't be running—to cause it to run you pass the **thread_id** to **resume_thread()** or **wait_for_thread()** (which are explained in the major section "Threads and Teams").

In addition to the BFile argument, **load_executable()** takes an *argc*/*argv* argument pair (which are 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**).
- Typically, you set the environment variable pointer to point to the global **environ** array (which you must declare as an **extern**).

The following example demonstrates a typical use of **load_executable()**.

```
#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;

/* Now declare the other bits. */
BFile exec_file;
record_ref exec_ref;

/* Let's use agv/agc just in case we're in a main()
 * call ourselves.
 */
char **agv;
int age;
thread_id exec_thread;
int return_value;

/* Get the ref to the executable and set the BFile with it.
 * We use get ref for path() here (and search for an
 * executable file called "/my_apps/adder"); for more
 * information on the BFile manipulations, see the Storage Kit
 * chapter. Notice, also, that we're being a bit lax about
 * checking for errors.
 */
get_ref_for_path("/my_apps/adder", &exec_ref);
exec_file.SetRef(exec_ref);

/* Set up the agv array. Let's pretend the adder program
 * takes two integers, adds them together, and returns the
 * result as main()'s exit code. There are three arguments,
 * so we allocate agv to hold four pointers (to include the
 * final NULL). Then we allocate and copy the three
 * string arguments.
 */
age = 3;

agv = (char **)malloc(sizeof(char *) * (age + 1));
```

```
agv[0] = (char *)malloc(strlen("adder")+1);
strcpy(argv[0], "adder");

agv[1] = (char *)malloc(2);
strcpy(agv[1], "5");

agv[2] = (char *)malloc(2);
strcpy(agv[2], "3" ) ;

agv[3] = NULL;

/* Finally, we call load_executable. */
exec_thread = load_executable(&exec_file, age, agv, environ);

/* 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);


/* return_value should be 8 (i.e. 5 + 3). */
```

Simple? You bet.

Using an Add-on Image

To use an add-on image, you need to compile its code in a special fashion, load the compiled image (from a running application), and then (from that application) play with the symbols that the image brings with it. Note that unlike loading an executable, loading an add-on doesn't create a separate team or even 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.

The following sections demonstrate these acts.

Compiling an Add-on Image

To compile an add-on image, you modify the "loader flags" that you pass to the compiler. In the **makefile** for your add-on image, add the following line:

```
LDFLAGS := -G -nodefaults -export all referenced_files
```

Here's what the flags do:

- The **-G** flag tells the compiler to create a shared library. There's actually no difference between the *binary* that holds a library image, and one that holds an add-on image.

- **-nodefaults** tells the compiler not to use the standard loader defaults that it might otherwise inherit.
- **-export all** tells the compiler to create a symbol table that includes all (non- static) symbols that it finds in the add-on code.
- *referenced_files* is a list of other image files (typically libraries) that contain symbols that the add-on references. For example, if the code in your add-on calls a system function such as **spawn_thread()**, you would need to include the **libbe.so** library in the list. Note that the library needs to be specified as a full pathname that locates the file as it resides on the machine on which you're compiling. If you're compiling on the BeBox, for example, the *libraries* list would probably look like this:

```
LDFLAGS := -G -nodefaults -export all /system/lib/libbe.so
```

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 within your application's address space.

For example, let's say you've created an add-on image that's stored in the file **/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("/addons/adder", &addon_ref);
addon_file.SetRef(addon_ref);

/* Load the add-on. */
addon_image = load_add_on(&addon_file) ;
```

Symbols

After you've loaded an add-on into your application, you'll want to examine the symbols that it has brought with it. To get information about a symbol, you call the **get_image_symbol()** function. 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 an integer that gives the symbol's *symbol type*. The only types you should care about are variables (for which you pass 1) and functions (you pass 2). (Symbolic constants for these values will be issued in the next release.)

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 in order to encodes 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_nth_image_info()

long **get_nth_image_info(**team_id *team*,
long *n*,
image_info **info***)*

Returns information about the *n*'th image (of whatever type) that's loaded into team. The information is returned in the image_info structure argument. The structure is defined as:

```
typedef struct {
        long volume;
        long directory;
        char name[B_FILE_NAME_LENGTH];
        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:

Field	Meaning
name	The name of the file whence sprang the image.
text	The address of the image's text segment.
text_size	The size of the text segment, in bytes;.
data	The address of the data segment.
data_size	The size of the 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 function returns **B_ERROR** if the designated image doesn't exist. Otherwise, it returns **B_NO_ERROR**.

get_nth_image_symbol(), get_image_symbol()

long **get_nth_image_symbol(**image_id *image*,
long *n*,
char **name*,
int **name_length*,
int **symbol_type*,
void ***location***)**

long **get_image_symbol(**image_id *image*,
char **symbol_name*,
int *symbol_type*,
void ***location***)**

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 then re-call **get_nth_image_symbol()** with an adequately lengthened *name* buffer, and an increased *name_length* value.

  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'll need to reset the *name_length* value before each call.

- The function sets *symbol_type* to 1 if the symbol is a variable, or 2 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.

Currently, the only way to get an **image_id** number is to load an add-on image. Since you don't (actively) load your app image or library images, you can't discover their **image_id** numbers, so you can't step through their symbols.

resolve_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 52.

The functions return **B_ERROR** if the designated image or symbol is invalid. 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** that represents the loaded image, **image_id** numbers are positive integers; if this function returns a negative number, the load failed. An example that demonstrates the use of **load_add_on()** is given in "Loading an Add-on Image" on page 52.

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 team's main thread. To start the application running, you need to pass the thread to **resume_thread()** or **wait_for_thread()**. An example that demonstrates the use of this function is given in "Loading an App Image" on page 49.

If the **thread_id** value is negative (less than **B_NO_ERROR**), the load failed.

Miscellaneous Functions

Declared in: <kernel/OS.h>

Overview

The functions listed below are useful, but don't fall under a grand heading such as threads or ports.

Atomic Functions

atomic_add(), atomic_and(), atomic_or()

long **atomic_add(**void ***atomic_variable*, long *add_value***)**
long **atomic_and(**void ***atomic_variable*, long *and_value***)**
long **atomic_or(**void ***atomic_variable*, long *or_value***)**

These functions perform the named operation (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.

Time Functions

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.

These functions aren't intended for scrupulously accurate measurement.

system_time()

double **system_time(**void**)**

Returns the number of microseconds that have elapsed since the computer was last booted.

Byte Swapping

read_16_swap(), read_32_swap(), write_16_swap(), write_32_swap()

short **read_16_swap(**void **address***)**
long **read_32_swap(**void **address***)**

void **write_16_swap(**void **address*, short *value***)**
void **write_32_swap(**void **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*.

System Information

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. The structure is defined as:

```
typedef struct {
        double boot_time;
        long cpu_count;
        cpu_info cpu_infos[B_MAX_CPU_NUM];
        long max_pages;
        long used_pages;
        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 structure's fields are:

Field	Meaning
boot_time	The time at which the computer was; last booted, measured in microseconds since January 1st, 1970.
cpu_count	The number of CPUs that are screwed into the motherboard.
cpu_infos	An array of **cpu_info** structures, one for each CPU.
max_pages	The total number of pages of memory in RAM.
used_pages	The number of pages of RAM that are currently being used.
max_sems	The total number of semaphores that the system can create.
used_sems	The number of semaphores that are currently active.
max_ports	The total number of ports that the system can create.
used_ports	The number of ports that are currently active.
max_threads	The total number of threads that the system can create.

used_threads The number of threads that are currently active.

max_teams The total number of teams that the system can create.

used_teams The number of teams that are currently active.

volume The volume ID of the volume that contains the current kernel.

directory The directory ID of the directory that contains the current kernel (but see the note, below).

name The file name of the current kernel.

The **cpu_info** structure is defined as:

```
typedef struct {
        double active_time;
} cpu_info
```

active_time measures the amount of time, in microseconds, that the CPU has actively been working since the machine was last booted.

Constants, Defined Types, and Structures

Constants

Area Allocation Constants

<kernel/OS.h>

Constant

B_ANY_ADDRESS
B_EXACT_ADDRESS
B_BASE_ADDRESS
B_CLONE_ADDRESS

These constants, which are used by the **create_area()** and **clone_area()** functions, describe where an area is to be allocated with respect to a given address.

See also: create_area() and **clone_area()** in "Areas"

Area Lock Constants

<kernel/OS.h>

Constant

B_NO_LOCK
B_LAZY_LOCK
B_FULL_LOCK

These constants, which are used by the **area_create()** function, describe the circumstances under which an area's memory is swapped in and out of core memory.

See also: area_create() in "Areas"

Area Protection Constants

<kernel/OS.h>

Constant

B_READ_AREA
B_WRITE_AREA

These constants, which are used by the **area_create()** and **area_clone()** functions, describe an area's read and write protection.

See also: area_create() in "Areas"

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.

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.

Operating System 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 entity. The constant is also used as a name-length limit by other kits.

Page Size

<kernel/OS.h>

Constant	Value
B_PAGE_SIZE	4096

The **PAGE_SIZE** constant gives the size, in bytes, of a page of RAM. The page size is used when you're creating, cloning, or resizing an area.

See also: create_area() in "Areas"

Port Message Count

<kernel/OS.h>

Constant	Value
B_MAX_PORT_COUNT	128

This constant gives the maximum number of messages a port can hold at a time.

See also: create_port() in "Ports"

Thread Priority Constants

<kernel/OS.h>

Constant
B_LOW_PRIORITY
B_NORMAL_PRIORITY
B_DISPLAY_PRIORITY
B_REALTIME_PRIORITY

These constants represent the thread priority levels. The higher a thread's priority, the more attention it gets from the CPUs; the constants are listed here from lowest to highest priority.

See also: the introduction to "Threads"

thread_state Constants

<kernel/OS.h>

Enumerated	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_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 enumerated constants, of type thread_state, represent the various states that a thread can be in.

See also: the introduction to "Threads"

Defined Types and Structures

area_id

<kernel/OS.h>

typedef long **area_id**

Used by the area functions to identify areas.

See also: the introduction to "Areas"

area_info

<kernel/OS.h>

```
typedef struct {
        area_id area;
        char name[B_OS_NAME_LENGTH];
        void *address;
        long size;
        ulong lock;
        ulong protection;
        team_id team;
        long ram_size;
        long copy_count;
        long in_count;
        long 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:

Field	Meaning
area	The **area_id** that identifies the area.
name	The name that was assigned to the area when it was created or cloned.
address	A pointer to the area's starting address.

size The size of the area, in bytes.

lock A constant that represents the area's locking scheme, one of **B_FULL_LOCK**, **B_LAZY_LOCK**, or **B_NO_LOCK**.

protection A mask that specifies whether the area's memory can be read and written. It's a combination of **B_READ_AREA** and **B_WRITE_AREA**.

team The **team_id** of the thread that created or cloned this area.

ram_size The amount of the area, in bytes, that's currently swapped in.

copy_count A count of the number of times any page in the area has been copied because a process has written to it (in other words, this is a "copy on write" count).

in_count A count of the number of times any page from the area has been swapped in.

out_count A count of the number of times any page from the area has been swapped out.

See also: get_area_info() in "Areas"

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

port_id

```
<kernel/OS.h>

typedef long port_id
```

Used by the port functions to identify ports.

See also: the introduction to "Ports"

sem_id

<kernel/OS.h>

typedef long **sem_id**

Used by the semaphore functions to identify semaphores.

See also: the introduction to "Semaphores"

system_info

```
typedef struct {
        double boot_time;
        long cpu_count;
        long cpu_info;
        long cpu_infos[B_MAX_CPU_NUM];
        long max_pages;
        long used_pages;
        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 contains all sorts of useful information about the computer. It's returned through a call to **get_system_info()**. The structure's fields are:

Field	Meaning
boot_time	The time at which the computer was last booted, measured in microseconds since January 1st, 1970.
cpu_count	The number of CPUs that are screwed into the motherboard.
cpu_infos	An array of **cpu_info** structures, one for each CPU.
max_pages	The total number of pages of memory in RAM.
used_pages	The number of pages of RAM that are currently being used.
max_sems	The total number of semaphores that the system can create.
used_sems	The number of semaphores that are currently active.
max_ports	The total number of ports that the system can create.

used_ports	The number of ports that are currently active.
max_threads	The total number of threads that the system can create.
used_threads	The number of threads that are currently active.
max_teams	The total number of teams that the system can create.
used_teams	The number of teams that are currently active.
volume	The volume ID of the volume that contains the current kernel.
directory	The directory ID of the directory that contains the current kernel (but see the note, below).
name	The file name of the current kernel.

The directory field is, alas, 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 in the next season.

team_id

<kernel/OS.h>

typedef long **team_id**

Used by the thread functions (and others) to identify teams.

See also: the introduction to "Threads"

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. A pointer to a function that adheres to this protocol must be passed to as an argument when you call **spawn_thread()**.

See also: the introduction to "Threads"

thread_id

<kernel/OS.h>

typedef long **thread_id**

Used by the thread functions (and others) to identify threads.

See also: the introduction to "Threads"

team_info

```
<kernel/OS.h>

typedef struct {
        team_id team;
        long object_count;
        long thread_count;
        long area_count;
        thread_id debugger_nub_thread;
        port_id debugger_nub_port;
} team_info
```

This structure holds information about a team. It's returned by functions such as **get_team_info()**. The fields are:

Field	Meaning
team	The **team_id** number of this team.
object_count	The number of "objects" or executable files (including libraries) that are loaded in the team.
thread_count	The number of threads that comprise the team.
area_count	The number of areas that the team can reference.
debugger_nub_thread	A thread that's used by the debugger to execute operations on this team.
debugger_nub_port	The port through which the debugger communicates with the team.

See also: get_team_info() in "Threads"

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:

Field	Meaning
id	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_SEM_WAIT**.
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: the introduction to "Threads"

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

8 The Device Kit

8 The Device Kit

This kit contains encapsulated interfaces to the various connectors and devices that can be attached to a BeBox. Currently, it contains just one class—BSerialPort. A BSerialPort object can represent any of the four RS-232 serial ports that are visible on the back of the machine. Other classes for other connectors will be added in future releases.

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:

- No 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()**)
- A timeout of one tenth of a second to read a target of one byte (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.

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 their contents won't be read (by the **Read()** function) and won't be transmitted over the connection (after having been written by **Write()**).

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(), SetTimeout(), SetNumBytes()

long **Read(**void **buffer*, long *maxBytes***)**

void **SetTimeout(**long *timeout***)**

void **SetNumBytes(**long *targetBytes***)**

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*. **Read()** fails if the BSerialPort object doesn't have a port open.

The actual number of bytes that **Read()** will place in the buffer before returning depends on the *timeout* and *targetBytes* set by the other two functions:

- **SetTimeout()** sets a time limit on how long **Read()** will wait for a character to arrive in the input buffer. The *timeout* is expressed in tenths of a second and is limited to 255 (25.5 seconds); it's set to 255 if a value over that amount is specified. The default setting is 1 (0.1 second).

  There is no time limit if *timeout* is set to 0.

- **SetNumBytes()** sets the target number of bytes that **Read()** will attempt to place in the *buffer* before it returns. The target is limited to 255 bytes; it's set to 255 if a value over that amount is specified. The default target is 1 byte.

  There is no target number of bytes to read if *targetBytes* is set to 0.

The way that **Read()** applies the *timeout* depends on the value for *targetBytes*, and the way it understands *targetBytes* depends on the *timeout*. The two values must be interpreted together:

- If neither *timeout* nor *targetBytes* is specified (both are set to 0), **Read()** reads as many bytes as it can from the input buffer, up to *maxBytes*. If the input buffer contains less than *maxBytes*, it reads them all and returns. It doesn't wait for more data to arrive.

- If only a *timeout* is specified (*targetBytes* is 0), **Read()** returns after reading a single byte—or, if the *timeout* expires, without reading anything.

- If only *targetBytes* is specified (*timeout* is 0), **Read()** returns after reading the target number of bytes, or *maxBytes* if less. It waits without time limit for the requisite number of bytes to arrive.

- If both a *timeout* and *targetBytes* are specified, the *timeout* is activated only after the first byte of data is read. It's then applied independently to each successive byte; the clock is reset to zero whenever another byte is read. **Read()** returns after reading *targetBytes* of data, or *maxBytes* if less. It returns sooner if the *timeout* expires on any byte.

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 **SYS_ERROR** (-1) if there's an error on 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()**

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**. < This function isn't implemented for the current release. >

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. No flow control is the default.

SetNumBytes() *see* Read()

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**. < This function isn't implemented for the current release. >

See also: SetDTR()

SetStopBits() *see* SetDataBits()

SetTimeout() *see* Read()

StopBits() *see* SetDataBits()

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 the constants and types defined for the: Device Kit, which currently contains only the BSerialPort class. Everything listed here is explained more fully in the descriptions of BSerialPort functions.

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: SetDataBits() in the BSerialPort class

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: SetDataRate() in the BSerialPort class

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: SetFlowControl() in the BSerialPort class

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: SetDataBits() in the BSerialPort class

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: SetDataBits() in the BSerialPort class

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 and **SetDataBits()** in the BSerialPort class

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 and **SetDataRate()** in the BSerialPort class

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 and **SetDataBits()** in the BSerialPort class

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 and **SetDataBits()** in the BSerialPort class

9 The Network Kit

9 The Network Kit

The Network Kit is a collection of global C functions that let you establish a link to and communicate with other computers through the TCP or UDP protocols. The names and intents of the functions, with two exceptions, adhere to the precedent set by the BSD network/socket implementation, although, note, some BSD functions are yet to be implemented. The two exceptions are:

- The Kit provides a **getusername()** function to make up for the current lack of system-wide password information.
- To close a socket, use the Be-specific **closesocket()** instead of **close()**.

See the header files **net/socket.h** and **net/netdb.h** for a complete list of the network API that's supported by the Kit. The Kit is, otherwise, currently undocumented.

10 The Support Kit

10 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:

- Common defined types,
- The error codes used in all the kits,
- The BList class for organizing allocated data (especially objects),
- The root BObject class, and
- A system for getting class information at run time.

Use as much or as little of this kit as you like.

Class Information

Declared in: <support/ClassInfo.h>

The class-information system is a set of macros that can supply information at run time about an object's class, such as its name and whether it derives from some other class. There are two parts to the system:

- The macros themselves, and
- The things you need to do to allow classes you design to participate in the system.

The following sections explore these two topics.

Notable by its absence from this list is the BClassInfo class. This class is the mechanism behind the class-information system—every class that participates in the system is given a BClassInfo object that supplies information about the class. An important feature of the system, however, is that you never have to instantiate (or otherwise locate) a BClassInfo object to take advantage of the information that it provides. So while you should be aware that the class exists (its declaration is, by necessity, public), you needn't be concerned with it. It's documented at the end of this discussion for completeness only.

The Class-Information Macros

An object of a class that participates in the class-information system (and this includes almost all the classes supplied by Be) 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_***)**

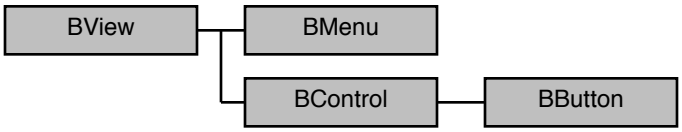
bool **is_instance_of(***object*, *class***)**

bool **is_kind_of_object(***class***)**

where *object* is a pointer to any type of object and *class* is the name of any class.

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, but not to a class to which it doesn’t rightly belong. The agent for this kind of safe casting is the following macro,

class ***cast_as(***object*, *class***)**

where *object* is a pointer to any type of object and *class* is the name of any class.

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, if it's safe to do so.

Suppose, for example, that you want to know whether the window where a particular view is located is a specialized kind of window that you've designed for your application—whether it's a My Window object and not just a generic El Window. If it is, you want to call some of the specific functions you implemented for the My Window class.

The view will reveal the window where it's located,

```
BWindow *window = myView->Window();
```

but since the returned object is typed as a BWindow, it won't give you access to any MyWindow functions, even if it's really an instance of the MyWindow class.

You could use the **is_kind_of()** macro to discover whether it would be safe to cast the *window* variable to the MyWindow class, then cast it if the answer was positive. However, the **cast_as()** macro provides a simpler solution that accomplishes this in one step:

```
MyWindow *mine;

if ( mine = cast_as(window, MyWindow) ) {
    /* Go ahead, it's a MyWindow object. *
     * 'mine' is correctly initialized. */
}
else
    /* Oops, no it isn't; 'mine' is NULL. */
```

Participating in the System

To take part in the class-information system, your classes need to include a pair of lines:

- A *declaration* line goes in the class declaration (in the header file).
- A *definition* line goes in the class implementation file.

There are two sets of both of these lines; the set you use depends on the (immediate) base class from which you're deriving your class.

If the Base Class Participates

If your class derives from a base class that participates in the class-information system, use this set:

B_DECLARE_CLASS_INFO(*base***)**

B_DEFINE_CLASS_INFO(*class***, *base*)**

where *base* is the name of the base class and *class* is the name of the new class you're defining.

For example, let's say you've created a PaperView class that derives from BView. BView participates in the class-information system (as do all Be classes, except where noted in the class description), so PaperView's declaration would look like this in the **PaperView.h** header file:

```
#include <interface/BView.h>

class PaperView : public BView
{
    B_DECLARE_CLASS_INF()(BView);
    /* Data and function declarations go here. */
}
```

And the implementation would follow this form in **PaperView.cpp**:

```
#include "PaperView.h"

B_DEFINE_CLASS_INF()(PaperView, BView);

PaperView::PaperView(. . .) : BView(. . .)
{
    . . .
}
```

If the Base Class Doesn't Participate

If your class doesn't derive from a participating class, use these lines instead of those shown above:

B_DECLARE_ROOT_CLASS_INFO()

B_DEFINE_ROOT_CLASS_INFO(*class***)**

The disposition of these lines is the same as their **B_DECLARE_CLASS_INFO** and **B_DEFINE_CLASS_INFO** counterparts—as the example below illustrates. In the example, a

class TopDog inherits from nobody. Notice that, in this case, the **TopDog.h** header must explicitly include **ClassInfo.h**:

```
#include <support/ClassInfo.h>

class TopDog
{
    B_DECLARE_ROOT_CLASS_INF()();
    /* Data and function declarations go here. */
}
```

The implementation file need only include the class header:

```
#include "TopDog.h"

B_DEFINE_ROOT_CLASS_INF()(TopDog);

TopDog::TopDog(. . .)
{
    . . .
}
```

The only Be classes that *don't* participate in the class-information system are the Interface Kit's BRect and BPoint, since they're simple data containers that have been sheared of all extra baggage. If your class derives from any other Be class, you should use the first set of class-information lines.

inherited

Every class that participates in the class-information system, and has a base class that also participates, gets a private keyword—**inherited**—typed to its base class. This term comes with the **B_DECLARE_CLASS_INFO** declaration; it's designed to simplify references to base-class functions, especially when overriding a virtual function in order to add something to it. For example:

```
bool MyClass::DoSomething(long foolish)
{
    . . .
    return inherited::DoSomething(foolish);
}
```

When every derived class has access to this same term, all classes can refer to inherited functions in the same way.

Caveats

The class-information system is accurate only for classes that explicitly participate. If, for example, the PaperView class didn't include the declaration and definition lines as shown above, the class-information macros would regard all PaperView objects as BView objects.

The macros fail entirely when applied to a class that not only doesn't participate in the system, but also doesn't derive (directly or indirectly) from any classes that do. The failure is reported as a compile-time error. Therefore, it's strongly suggested that your class derive from BObject if no other Be class is a fit base.

Finally, the class-information system doesn't accommodate multiple inheritance.

BClassInfo

Derived from: *none*

Declared in: <support/ClassInfo.h>

Overview

A BClassInfo object represents and stores information about a class. Its functions provide information about the class it represents (not about the BClassInfo class itself). Under normal circumstances, this information should be obtained through the macros described in the previous section rather than through BClassInfo functions.

Constructor

BClassInfo()

BClassInfo(const char ***name*, const BClassInfo ***base***)**

Initializes a BClassInfo object from the *name* of the class it is to represent and the BClassInfo object for the *base* class that the represented class derives from.

BClassInfo objects should be constructed using the declaration and definition lines described under "Class Information" above.

Member Functions

CanCast()

bool **CanCast(**const BClassInfo ***another***)** const

Returns **TRUE** if instances of the represented class can be cast to the class that *another* stands for, and **FALSE** if they can't.

DerivesFrom()

bool **DerivesFrom(**const BClassInfo **another*, bool *directOnly* = FALSE**)** const

Returns **TRUE** if the represented class derives from the class that *another* stands for, and **FALSE** if not.

If the *directOnly* flag is **TRUE**, this function returns **TRUE** only if the class that *another* stands for is the immediate base class of the represented class.

IsSameAs()

bool **IsSameAs(**const BClassInfo **another***)** const

Returns **TRUE** if the represented class is the class that *another* other stands for, and **FALSE** if the two classes differ.

Name()

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

Returns the name of the represented class.

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 copied object.

~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 *list* to the BList. The items from the other list are inserted at *index*—or, if no index is given, 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: MakoEmpty()

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.

You can also index directly into the array, if you're certain that the *index* is in-range and you want to avoid the overhead of checking it each time an item is requested. The **Items()** function returns a pointer to the list. For example:

```
myType item = (myType)Items()[index];
```

See also: FirstItem(), **LastItem()**

Items()

inline void ***Items(**void**)** const

Returns a pointer to the BList's list. Acquiring the "list pointer" is useful if, for example, you want to directly manipulate the order of items in the list. (But see the **SortItems()** function for a simple way to sort a list).

Although the practice is discouraged, you can also step through the list of items by incrementing the list pointer. 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*)(comt void *, const void *)**)**

Rearranges the items in the list. The items are quick-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&**)**

Assigns the contents on one BList object to another:

```
BList newList = oldList;
```

After the assignment, the two lists are duplicates of one another. Each object has its own independent copy of list data; destroying one of the objects won't affect the other.

However, only the items in the list are copied, not the data they point to. Each set of items references the same underlying data.

BLocker

Derived from: public BObject

Declared in: <support/Locker.h>

Overview

The BLocker class offers a simple way to set up a locking mechanism similar to the ones used in the BLooper and BMessageQueue classes in the Application Kit and the BBitmap class in the Interface Kit.

Typically, a BLocker instance is declared as a private data member of a class that also declares public **Lock()** and **Unlock()** functions:

```
class MyObject : public BObject
{
public:
    void Lock();
    void Unlock();
    . . .
private:
    BLocker lock;
    . . .
};
```

Lock() and **Unlock()** are then implemented as simple calls to their BLocker counterparts:

```
void MyObject::Lock()
{
    lock.Lock();
    . . .
}

void MyObject::Unlock()
{
    . . .
    lock.Unlock();
}
```

Like other locking mechanisms, this one relies on a semaphore. It builds on the semaphore to permit nested calls to the locking functions. However, it lacks some features that you'd get if you used the semaphore directly—such as a timeout mechanism and the ability to discover how many threads are waiting to attain the lock.

See also: "Semaphores" in the chapter on the Kernel Kit

Constructor and Destructor

BLocker()

BLocker(const char **name***)**

Sets up the locking mechanism and assigns *name* to the semaphore that will be used to operate the lock. The name doesn't appear elsewhere in the API, but it may show up in the debugger.

-BLocker()

virtual ~**BLocker(**void**)**

Gets rid of the semaphore.

Member Functions

CheckLock()

bool **CheekLock(**void**)** const

Checks to see whether the calling thread is the thread that currently owns the lock. If it is, all is well and **CheckLock()** returns **TRUE**. If it's not, **CheckLock()** returns **FALSE** and deposits you in the debugger so you can find out why not.

While developing your application, it's a good idea to call **CheckLock()** before each **Unlock()** call. After the application has been thoroughly debugged, you can remove the **CheckLock()** calls.

Lock(), Unlock()

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

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

These functions lock and unlock whatever object or data structure the BLocker is serving. **Lock()** doesn't return until it has secured the lock; **Unlock()** releases the lock (or takes a step in that direction) and returns immediately.

These calls can be nested. The lock isn't released until every **Lock()** call is balanced by a call to **Unlock()**

It's an error to call **Unlock()** from a thread that doesn't own the lock. For debugging purposes, you can call **CheckLock()** before calling **Unlock()** to make sure this doesn't happen in your code.

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** *.

As a further incentive, simply by deriving from BObject your classes will be recognized by the class-information mechanism. (They may not be recognized correctly, but at least a class-information query on your objects won't stop the compiler in its tracks. See "Class Information" on page 5 for details.)

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.

Constants, Defined Types, and Macros

This section lists the general-purpose constants, defined types, and macros that are grouped in the Support Kit and used throughout the Be application-programming interface. Included are commonly used constants such as **NULL** and its synonym **NIL**, basic defined types such as **bool** and **ulong**, and a couple of simple macros. The error codes for all kits are also defined here.

Not listed are the class information macros; they're documented under "Class Information" on page 5.

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 is defined as "", a string consisting only of the null character.

Error Codes—General

<support/Errors.h>

Enumerated constant	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 general 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.ERROR = -1	This is a convenient catchall for general errors.
B_NO_ERROR = 0	Everything's OK.

Defined constant	Meaning
B_ERRORS_END	Marks the end of all Be-defined error codes.

Error codes are returned by various functions to indicate the success or to describe the failure of a requested operation. The general errors listed above are used throughout the Be API. Not all of these constants (or those listed in the sections below) are actually used—some are reserved for future releases—but you should avoid using the same names in the return codes that you create yourself.

All 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 */
```

Be reserves all error-code values that are less than or equal to **B_ERRORS_END** (all values from **LONG_MIN** through **B_ERRORS_END**). All the constants listed above, except for **B_ERROR** and **B_NO_ERROR**, are within these bounds. If you want to define your own negative-valued error codes, you should begin with the value (**B_ERRORS_END** + 1).

Error Codes—Application Kit

<support/Errors.h>

Enumerated constant	Meaning
B_UNEXPECTED_REPLY	A reply is being sent to a local message.
B_DUPLICATE_REPLY	A previous reply has already been sent.
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: the **Error()** functions in the BMessage and BMessenger classes

Error Codes—Debugger

<support/Errors.h>

Enumerated constant

B_DEBUGGER_ALREADY_INSTALLED

This constant signals that the debugger has already been installed for a particular team and can't be installed again.

Error Codes—Kernel Kit

<support/Errors.h>

Enumerated constant	Meaning
B_BAD_SEM_ID	Semaphore identifier (**sem_id**) is invalid.
B_NO_MORE_SEMS	All semaphores are currently taken.
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.

These error codes are returned by functions in the Kernel Kit, and occasionally by functions defined in higher level kits. See the specific functions for details.

Error Codes—Media Kit

<support/Errors.h>

Enumerated constant	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.

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.

B_PFI, B_PFL, B_PFV

<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.

Macros

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).