2 The Application Kit

Introduction 5
Messaging 6
Messages 6
Message Protocols 7
Message Ownership 7
Message Loops 7
System Messages 8
Specialized BLoopers 9
Message-Specific Dispatching 9
Picking a Handler and a Hook Function 10
Application-Defined Messages 11
Posting Messages. 11
Sending Messages 11
Dropping a Message 12
Two-Way Communication 13
Specifying the Target. 14
Preferred Handlers 14
Message Filters 14
System Messages in the Application Kit 15
System Management Messages. 15
Application Messages. 16
Setting Up an Application .17
Icons. 18
Application Information 18
Signatures. 18
Launch Information 19
Other Information 20

BApplication 21
Overview 21
Derived Classes 21
Constructing the Object and Running the Message Loop .22
be_app 22
main(). 23
Configuration Messages Received on Launch 23

Quitting 24
Aborted Run 24
Locking 25
Hook Functions 25
Constructor and Destructor 26
Member Functions. 27

BClipboard **43**
Overview 43
Using the Clipboard. 43
Example 1: Adding Data to the Clipboard. 44
Example 2: Retrieving Data from the Clipboard . . .44
Member Functions. 45

BHandler **49**
Overview 49
Hook Functions 50
Constructor and Destructor 50
Member Functions. 50

BLooper. **55**
Overview 55
Running the Loop 55
Receiving and Dispatching Messages. 55
Acting as the Handler .56
Eligible Handlers 56
Hook Functions 57
Constructor and Destructor 57
Member Functions. 58

BMessage **69**
Overview 69
Message Contents 69
Message Constants 70
Type Codes 71
Publishing Message Protocols 72
Error Reporting 73
Data Members 73
Constructor and Destructor 73
Member Functions. 74
Operators 87

BMessageFilter **89**
Overview 89
Hook Functions 89
Constructor and Destructor 90
Member Functions. 91

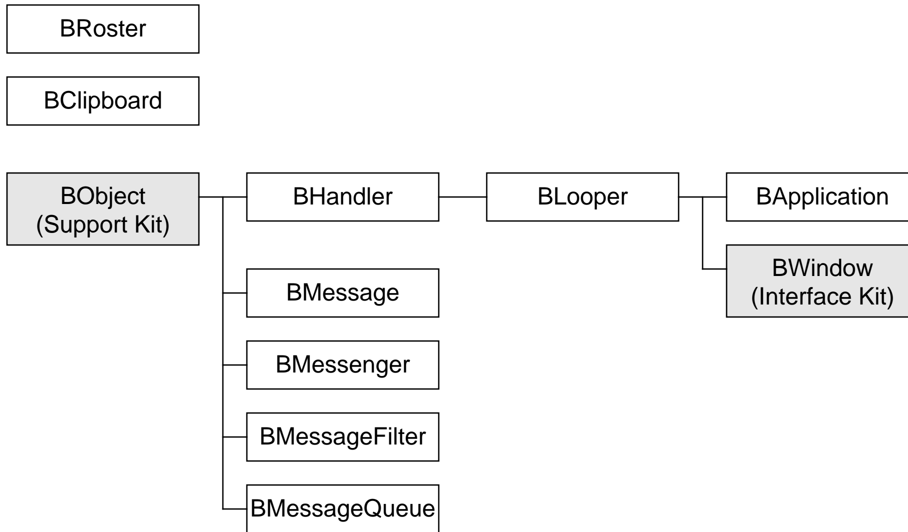
BMessageQueue **93**
- Class Description 93
- Constructor and Destructor 93
- Member Functions 94

BMessenger **97**
- Overview 97
- Constructor and Destructor 97
- Member Functions 99
- Operators 102

BRoster **103**
- Overview 103
- Constructor and Destructor 104
- Member Functions 104

Global Variables, Constants, and Defined Types **109**
- Global Variables 109
- Constants 110
- Defined Types 114

Application Kit Inheritance Hierarchy



2 The Application Kit

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

There are four parts to the Application Kit:

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

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

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

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

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

  The BApplication object also runs the application's main message loop, where it receives messages that concern the application as a whole. Externally, this object

represents the application to other applications; internally, it's the center where application-wide services and global information can be found. Because of it's pivotal role, it's assigned to a global variable, be_app, to make it easily accessible.

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

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

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

Messaging

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

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

Messages

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

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

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

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

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

Message Protocols

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

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

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

Message Ownership

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

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

Message Loops

In the Be model, messages are delivered to threads running *message loops*. Arriving messages are placed in a queue, and are then taken from the queue one at a time. After getting a message from the queue, the thread decides how it should be handled and

dispatches it to an object that can respond. When the response is finished, the thread deletes the message and takes the next one from the queue—or, if the queue is empty, waits until another message arrives.

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

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

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

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

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

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

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

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

System Messages

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

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

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

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

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

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

Specialized BLoopers

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

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

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

Message-Specific Dispatching

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

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

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

- Those that are left to the application to implement. These functions are named for what they announce—for what led to the function call rather than for what the function might be implemented to do. KeyDown() is an example.
- Those that have a default implementation to cover the common case. Like those in the first group, these functions also are named for the occurrence that prompted the function call. ScreenChanged() is an example.
- Those that are fully implemented to perform a particular task. These are functions that you can call, but they're also hooks that are called for you. Like most ordinary functions, they're named for what they do—like Activate()—not for what led to the function call.

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

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

Picking a Handler and a Hook Function

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

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

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

Application-Defined Messages

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

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

Posting Messages

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

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

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

Sending Messages

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

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

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

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

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

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

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

Dropping a Message

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

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

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

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

Two-Way Communication

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Specifying the Target

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

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

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

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

Preferred Handlers

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

Message Filters

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

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

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

common filters, a BLooper can, like other BHandlers, maintain a list of filters specific to its role as a target handler.)

System Messages in the Application Kit

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

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

System Management Messages

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

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

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

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

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

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

Application Messages

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

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

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

- A B_READY_TO_RUN message reports that the application has finished launching and configuring itself and its main thread is ready to respond to messages.

- A B_APP_ACTIVATED message is delivered when the application becomes the active application—the one that the user is currently engaged with—or when it relinquishes that status to another application.

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

- A B_QUIT_REQUESTED message is taken by the BApplication object to be a request to shut the entire application down, not just one thread. An application that has a user interface usually interprets some user action (such as clicking a "Quit" menu item) as a request to quit and, in response, posts a B_QUIT_REQUESTED message to the BApplication object. An application that is the servant of other applications may get the request from a remote source.

- A B_ABOUT_REQUESTED message requests information about the application, usually through an "About . . ." item in the application's main menu. The application should set up this item to post a B_ABOUT_REQUESTED message to the BApplication object. In response, the BApplication object should display a window with general information about the application.

Other application messages report information from remote sources:

- A B_ARGV_RECEIVED message is delivered either on-launch or after-launch when the application receives strings of characters the user typed on the command line, or when the application is launched by another application and is passed a similar array of character strings.

- A B_REFS_RECEIVED message passes the application one or more references to database records. Typically, this means the user has chosen some files from the file panel, double-clicked a document icon in the Browser, or dragged the icon and dropped in on the application icon.

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

The system is the source of one repeated message:

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

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

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

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

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

Setting Up an Application

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

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

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

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

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

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

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

Icons

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

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

Application Information

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

Signatures

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

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

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

Launch Information

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

B_MULTIPLE_LAUNCH Several instances of the application can be running at once. It can be launched any number of times from the same executable file.

This is the normal behavior for most utilities, such as the compiler, tar, or Heap Watch. It's also appropriate for an application that can deal with only one document at a time, and therefore must be launched anew each time it's asked to handle another file.

B_SINGLE_LAUNCH Normally, only one instance of the application can be running. However, if the user copies the executable file for the application, it can be launched once from each copy.

This is the normal behavior for most applications, including applications that can deal with more than one document at a time.

B_EXCLUSIVE_LAUNCH When the application is running, no other instance of the same application can be launched from any source.

This is appropriate for applications that require exclusive ownership of a system resource, like the telephone line.

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

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

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

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

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

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

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

Other Information

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

B_ARGV_ONLY — The application doesn't participate in the messaging system. Therefore, the only information it can receive are command-line arguments, *argc* and *argv*, passed to the main() function.

B_ARGV_ONLY is assumed if the application doesn't have a BApplication object.

B_BACKGROUND_APP — The application doesn't have a user interface and therefore shouldn't appear in the Browser's application list.

BApplication

Derived from: public BLooper

Declared in: <app/Application.h>

Overview

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

The BApplication object has these primary responsibilities:

- *It makes a connection to the Application Server.* Any application that puts a window on-screen or relies on other system services needs this connection. It's made automatically when the BApplication object is constructed.
- *It runs the application's main message loop.* The BApplication object is a kind of BLooper, but instead of spawning an independent thread, it runs a message loop in the application's main thread (the thread that the **main()** function executes in). This loop receives and processes messages that affect the entire application, including the initial messages received from remote applications. It gets several messages from the Browser (such as reports of what documents to open). Any application that's known to the Browser or that cooperates with other applications needs a main message loop.
- *It's the home for application-wide elements of the user interface.* For example, it sets up the application's main menu and runs the file panel, which permits users to navigate the file system and pick files to open. It also lets you set, hide, and show the application's cursor. The ability to define the look of the cursor is provided by BApplication's **SetCursor()** function.

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

Derived Classes

BApplication typically serves as the base class for a derived class that specializes it and extends it in ways that are appropriate for a particular application. It declares (and inherits

declarations for) a number of hook functions that you can implement in a derived class to augment and fine-tune what it does.

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

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

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

Constructing the Object and Running the Message Loop

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

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

be_app

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

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

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

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

main()

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

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

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

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

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

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

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

Configuration Messages Received on Launch

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

For example, when the user double-clicks a document icon to launch an application, the Browser passes the application a **B_REFS_RECEIVED** message with information about the

document. When launched from the command line, the application gets a B_ARGV_RECEIVED message listing the command-line arguments. When launched by the BRoster object, it might receive an arbitrary set of configuration messages.

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

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

Quitting

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

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

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

Aborted Run

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

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

Locking

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

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

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

Hook Functions

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

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

Constructor and Destructor

BApplication()

BApplication(ulong *signature***)**

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

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

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

~BApplication()

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

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

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

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

Member Functions

AboutRequested()

virtual void **AboutRequested**(void)

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

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

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

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

Activate()

virtual void **Activate**(void)

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

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

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

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

AppActivated()

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

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

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

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

ArgvReceived()

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

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

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

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

```
Splotch myArtwork
```

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

```
Splotch yourArtwork
```

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

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

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

CloseFilePanel() *see* RunFilePanel()

CountWindows()

long **CountWindows**(void) const

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

See also: the BWindow class in the Interface Kit

DispatchMessage()

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

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

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

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

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

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

FilePanelClosed()

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

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

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

GetAppInfo()

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

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

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

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

is simply a shorthand for:

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

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

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

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

HandlersRequested()

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

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

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

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

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

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

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

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

HideCursor(), ShowCursor(), ObscureCursor()

void **HideCursor**(void)

void **ShowCursor**(void)

void **ObscureCursor**(void)

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

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

IsCursorHidden()

bool **IsCursorHidden**(void) const

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

See also: **HideCursor()**

IsFilePanelRunning() *see* RunFilePanel()

IsLaunching()

bool **IsLaunching**(void) const

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

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

See also: **ReadyToRun()**

MainMenu() *see* SetMainMenu()

MenusWillShow()

virtual void **MenusWillShow**(void) const

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

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

ObscureCursor() *see* HideCursor()

Pulse()

virtual void **Pulse**(void)

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

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

The default version of this function is empty.

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

Quit()

virtual void **Quit**(void)

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

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

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

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

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

QuitRequested()

virtual bool **QuitRequested**(void)

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

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

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

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

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

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

ReadyToRun()

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

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

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

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

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

RefsReceived()

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

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

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

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

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

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

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

Run()

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

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

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

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

RunFilePanel(), CloseFilePanel(), IsFilePanelRunning()

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

void CloseFilePanel(void)

bool IsFilePanelRunning(void)
```

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

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

  ```
  WishMaker : Open
  ```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

When the user operates the "Open" (or *openButtonLabel*) button, the file panel sends a message to the BApplication object. If a customized *message* is provided, it's used as the

model for the message that's sent. If a *message* isn't provided, a standard **B_REFS_RECEIVED** message is sent instead. It has one data entry:

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

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

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

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

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

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

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

SetMainMenu(), MainMenu()

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

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

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

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

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

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

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

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

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

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

be_app->SetMainMenu(menu);
```

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

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

SetCursor()

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

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

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

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

- The second byte specifies the depth of the cursor image, in bits per pixel. Currently, only monochrome one-bit-per-pixel images are acceptable.

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

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

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

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

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

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

See also: **HideCursor()**

SetPulseRate()

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

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

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

See also: **Pulse()**

ShowCursor() *see* HideCursor()

VolumeMounted(), VolumeUnmounted()

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

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

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

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

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

See also: the BVolume class in the Storage Kit

WindowAt()

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

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

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

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

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

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

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

See also: **CountWindows()**

BClipboard

Derived from: *none*

Declared in: <app/Clipboard.h>

Overview

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

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

Using the Clipboard

The central BClipboard functions are these:

- AddData() lets you add a new item of data to the clipboard. The data that's added is copied from an argument passed to the function. Each clipboard item is identified (primarily) by its data type (which is represented by one of the standard type constants, such as B_ASCII_TYPE or B_REF_TYPE, that are defined in **app/AppDefs.h**).
- FindData() retrieves data from the clipboard by providing the caller with a pointer to a specific item. This pointer points to data that resides on the clipboard—the function doesn't copy the data.

You *must* bracket calls to AddData() and FindData() with calls to Lock() and Unlock(). This prevents other applications from accessing the clipboard while your application is using it. Conversely, if some other application (or if another thread in your application) holds the lock to the clipboard when you call Lock(), your application (or thread) will hang until the current lock holder calls Unlock()—in other words, Lock() will always succeed, even if it has to wait forever to do so. Currently, unfortunately, there's no way to tell if the

clipboard is already locked, nor can you specify a time limit beyond which you won't wait for the lock.

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

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

Example 1: Adding Data to the Clipboard

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

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

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

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

Example 2: Retrieving Data from the Clipboard

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

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

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

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

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

Member Functions

AddData(), AddText()

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

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

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

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

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

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

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

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

Clear()

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

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

Commit()

void **Commit**(void)

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

CountEntries()

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

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

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

DataOwner()

BMessenger **DataOwner**(void)

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

FindData(), FindText()

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

const char *__FindText__(long **numBytes*)

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

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

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

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

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

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

Lock(), Unlock()

bool **Lock**(void)

void **Unlock**(void)

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

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

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

BHandler

Derived from: public BObject

Declared in: <app/Handler.h>

Overview

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

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

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

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

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

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

Hook Functions

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

Constructor and Destructor

BHandler()

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

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

~BHandler()

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

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

Member Functions

AddFilter() *see* SetFilterList()

FilterList() *see* SetFilterList()

HandlersRequested()

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

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

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

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

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

Looper()

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

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

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

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

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

MessageReceived()

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

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

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

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

```
            case COMMAND_THREE:
                . . .
                break;
            default:
                inherited::MessageReceived(message);
                break;
            . . .
            }
        }
```

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

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

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

NextHandler() *see* SetNextHandler()

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

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

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

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

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

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

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

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

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

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

SetName(), Name()

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

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

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

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

SetNextHandler(), NextHandler()

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

BHandler ***NextHandler**(void) const

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

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

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

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

See also: **MessageReceived()**

BLooper

Derived from: public BHandler

Declared in: <app/Looper.h>

Overview

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

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

Running the Loop

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

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

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

Receiving and Dispatching Messages

Messages are posted to the BLooper's thread by calling its **PostMessage()** function. This simply puts the message in a queue. Messages can also be delivered to the BLooper's

queue—somewhat more indirectly—by a BMessenger object or by the **SendReply()** function of a BMessage object.

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

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

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

Acting as the Handler

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

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

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

Eligible Handlers

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

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

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

send and by **PostMessage()** when a BHandler is named as the target of a message posted to the BLooper.

Hook Functions

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

Constructor and Destructor

BLooper()

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

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

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

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

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

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

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

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

~BLooper()

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

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

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

See also: **Quit()**

Member Functions

AddCommonFilter() *see* SetCommonFilterList()

AddHandler(), RemoveHandler()

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

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

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

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

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

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

CommonFilterList() *see* SetCommonFilterList()

CurrentMessage(), DetachCurrentMessage()

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

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

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

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

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

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

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

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

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

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

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

DispatchMessage()

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

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

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

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

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

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

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

For example:

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

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

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

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

HandlersRequested()

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

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

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

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

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

IsLocked() *see* LockOwner()

Lock(), Unlock()

bool **Lock**(void)

void **Unlock**(void)

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

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

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

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

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

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

See also: **LockOwner()**

LockOwner(), IsLocked()

inline thread_id **LockOwner**(void) const

inline bool **IsLocked**(void) const

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

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

See also: **Lock()**

Looper()

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

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

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

MessageQueue()

BMessageQueue ***MessageQueue**(void) const

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

See also: the BMessageQueue class

PostMessage()

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

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

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

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

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

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

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

is equivalent to:

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

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

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

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

PreferredHandler()

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

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

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

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

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

The BLooper version of this function simply returns **NULL**, to indicate that generic BLoopers don't have a preferred handler. Note, however, that when a **NULL** handler is passed to **PostMessage()**, that function designates the BLooper itself as the target. For

example, if **PreferredHandler()** returned **NULL** in the line of code shown above, the message would be dispatched to *myLooper* by default. Thus, in effect, a generic BLooper is its own preferred handler, even though **PreferredHandler()** returns **NULL**.

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

Quit()

virtual void **Quit**(void)

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

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

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

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

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

See also: **QuitRequested()**

QuitRequested()

virtual bool **QuitRequested**(void)

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

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

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

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

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

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

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

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

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

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

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

RemoveCommonFilter() *see* SetCommonFilterList()

Run()

virtual thread_id **Run**(void)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Thread(), Team()

thread_id **Thread**(void) const

team_id **Team**(void) const

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

Unlock() *see* Lock()

BMessage

Derived from: public BObject

Declared in: <app/Message.h>

Overview

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

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

Message Contents

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

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

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

and this code locates it:

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

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

Message Constants

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

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

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

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

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

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

Type Codes

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

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

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

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

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

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

and the underbar. For example, **B_DOUBLE_TYPE** and **B_POINTER_TYPE** are defined as follows:

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

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

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

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

Publishing Message Protocols

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

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

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

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

Error Reporting

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

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

Data Members

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

Constructor and Destructor

BMessage()

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

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

```
#define RECEIPT_ACKNOWLEDGED  0x80
```

a complete message can be created as simply as this:

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

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

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

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

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

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

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

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

~BMessage()

virtual ~**BMessage**(void)

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

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

Member Functions

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

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

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

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

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

long **AddDouble**(const char **name*, double *aDouble*)

long **AddRef(**const char **name*, record_ref *aRef***)**

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

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

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

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

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

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

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

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

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

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

If more than one item of data is added under the same name, the BMessage creates an array of data for that name. Each successive call appends another data element to the end

of the array. For example, the following code creates an array named “primes” with 37 stored at index 0, 223 stored at index 1, and 1,049 stored at index 2.

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

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

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

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

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

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

CountNames()

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

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

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

See also: **GetInfo()**

Error()

long **Error**(void)

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

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

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

Possible error returns include the following:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Flatten(), Unflatten()

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

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

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

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

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

GetInfo()

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

Provides information about the data entries stored in the BMessage.

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

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

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

- It places a pointer to the name of the data entry in the variable referred to by *nameFound*.
- It puts the code for the type of data the entry contains in the variable referred to by *typeFound*. This will be the same as the *type* requested, unless the requested type is **B_ANY_TYPE**, in which case *typeFound* will be the actual type stored under the name.

- It records the number of data items stored within the entry in the variable referred to by *countFound*.

If **GetInfo()** can't find data of the requested *type* at *index*, it registers an error, sets the *countFound* variable to 0, and returns **FALSE**.

This version of **GetInfo()** can be used to iterate through all the BMessage's data. For example:

```
char   *name;
ulong   type;
long    count;

for ( long i = 0;
      msg->GetInfo(B_ANY_TYPE, i, &name, &type, &count);
      i++ ) {
    . . .
}
```

If the index is incremented from 0 in this way, all data of the requested type will have been read when **GetInfo()** returns **FALSE**. If the requested type is **B_ANY_TYPE**, as shown above, it will reveal the name and type of every entry in the BMessage.

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

HasData(), HasBool(), HasLong(), HasFloat(), HasDouble(), HasRef(), HasMessenger(), HasPoint(), HasRect(), HasObject(), HasString()

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

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

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

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

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

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

bool **HasMessenger**const char **name*, long *index* = 0**)**

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

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

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

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

These functions test whether the BMessage contains data of a given name and type.

If *type* is **B_ANY_TYPE** and no *index* is provided, **HasData()** returns **TRUE** if the BMessage stores any data at all under the specified *name*, regardless of its type, and **FALSE** if the name passed doesn't match any within the object.

If *type* is a particular type code, **HasData()** returns **TRUE** only if the BMessage has a *name* entry that stores data of that type. If the *type* and *name* don't match, it returns **FALSE**.

If an *index* is supplied, **HasData()** returns **TRUE** only if the BMessage has a *name* entry that stores a data item of the specified *type* at that particular *index*. If the index is out of range, it returns **FALSE**.

The other functions—**HasBool()**, **HasFloat()**, **HasPoint()**, and so on—are specialized versions of **HasData()**. They test for a particular type of data stored under the specified *name*.

An error code is set (which **Error()** will return) whenever any of these functions returns **FALSE**.

See also: **GetInfo()**

IsEmpty() *see* MakeEmpty()

IsReply() *see* WasSent()

IsSourceRemote() *see* WasSent()

IsSourceWaiting() *see* WasSent()

IsSystem()

bool **IsSystem**(void)

Returns **TRUE** if the **what** data member of the BMessage object identifies it as a system-defined message, and **FALSE** if not.

Unlike the **GetInfo()** and **HasData()** functions, a return of **FALSE** does not indicate an error. **IsSystem()** resets the error code that **Error()** returns to **B_NO_ERROR** whether the BMessage is a system message or not.

MakeEmpty(), IsEmpty()

long **MakeEmpty**(void)

bool **IsEmpty**(void)

MakeEmpty() removes and frees all data that has been added to the BMessage, without altering the **what** constant. It returns **B_NO_ERROR**.

IsEmpty() returns **TRUE** if the BMessage has no data (whether or not it was emptied by **MakeEmpty()**), and **FALSE** if it has some.

Both functions reset the error code to **B_NO_ERROR** in all cases.

See also: **RemoveName()**

Previous() *see* WasSent()

PrintToStream()

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

Prints information about the BMessage to the standard output stream (**stdout**). Each entry of named data is reported in the following format,

```
#entry name, type = type, count = count
```

where *name* is the name that the data is registered under, *type* is the constant that indicates what type of data it is, and *count* is the number of data items in the named array.

RemoveName()

bool **RemoveName(**const char **name***)**

Removes all data entered in the BMessage under *name*, frees the memory that was allocated to hold the data, and returns **TRUE**. If there is no data entered under *name*, this function registers an error (**B_NAME_NOT_FOUND**) and returns **FALSE**.

See also: **MakeEmpty()**

ReplaceData(), ReplaceBool(), ReplaceLong(), ReplaceFloat(), ReplaceDouble(), ReplaceRef(), ReplaceMessenger(), ReplacePoint(), ReplaceRect(), ReplaceObject(), ReplaceString()

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

long **ReplaceBool(**const char **name*, bool *aBool***)**
long **ReplaceBool(**const char **name*, long *index*, bool *aBool***)**

long **ReplaceLong(**const char **name*, long *aLong***)**
long **ReplaceLong(**const char **name*, long *index*, long *aLong***)**

long **ReplaceFloat(**const char **name*, float *aFloat***)**
long **ReplaceFloat(**const char **name*, long *index*, float *aFloat***)**

long **ReplaceDouble(**const char **name*, double *aDouble***)**
long **ReplaceDouble(**const char **name*, long *index*, double *aDouble***)**

long **ReplaceRef(**const char **name*, record_ref *aRef***)**
long **ReplaceRef(**const char **name*, long *index*, record_ref *aRef***)**

long **ReplaceMessenger(**const char **name*, BMessenger *aMessenger***)**
long **ReplaceMessenger(**const char **name*, long *index*, BMessenger *aMessenger***)**

long **ReplacePoint(**const char **name*, BPoint *aPoint***)**
long **ReplacePoint(**const char **name*, long *index*, BPoint *aPoint***)**

long **ReplaceRect(**const char **name*, BRect *aRect***)**
long **ReplaceRect(**const char **name*, long *index*, BRect *aRect***)**

long **ReplaceObject(**const char **name*, BObject **anObject***)**
long **ReplaceObject(**const char **name*, long *index*, BObject **anObject***)**

long **ReplaceString(**const char **name*, const char **aString***)**
long **ReplaceString(**const char **name*, long *index*, const char **aString***)**

These functions replace a data item in the *name* entry with another item passed as an argument. If an *index* is provided, they replace the item in the *name* array at that index; if an *index* isn't mentioned, they replace the first (or only) item stored under *name*. If an *index* is provided but it's out-of-range, the replacement fails.

ReplaceData() replaces an item in the *name* entry with *numBytes* of *data*, but only if the *type* code that's specified for the data matches the type of data that's already stored in the entry. The *type* must be specific; it can't be **B_ANY_TYPE**.

The other functions are simplified versions of **ReplaceData()**. They each handle the specific type of data declared for their last arguments. They succeed if this type matches the type of data already in the *name* entry, and fail if it does not.

If successful, all these functions return **B_NO_ERROR**. If unsuccessful, they register and return an error code—**B_BAD_INDEX** if the *index* is out-of-range, **B_NAME_NOT_FOUND** if the *name* entry doesn't exist, or **B_BAD_TYPE** if the entry doesn't contain data of the specified type.

See also: **AddData()**

ReturnAddress() *see* WasSent()

SendReply()

> long **SendReply(**BMessage **message*, BMessage ***reply***)**
> long **SendReply(**ulong *command*, BMessage ***reply***)**
> long **SendReply(**BMessage **message*, BHandler **replyTarget* = NULL**)**
> long **SendReply(**ulong *command*, BHandler **replyTarget* = NULL**)**

Sends a reply *message* back to the sender of the BMessage (in the case of a synchronous reply) or to a target BHandler (in the case of an asynchronous reply). Whether the reply is synchronous or asynchronous depends on how the message it replies to was sent:

- The reply is delivered synchronously if the message sender is waiting for it to arrive. The function that sent the BMessage doesn't return until it receives the reply. If an expected reply has not been sent by the time the BMessage object is deleted, a default **B_NO_REPLY** message is returned to the sender.

- The reply is delivered asynchronously if the message sender isn't waiting for a reply. In this case, the sending function designates a target BHandler (and, through the BHandler, a target BLooper) for any replies that might be sent, then returns immediately after putting the BMessage in the pipeline. The default target for a reply is the sender's BApplication object.

SendReply() works only for BMessage objects that have been processed through a message loop and delivered to you. However, it doesn't work for messages that were posted to the loop, only for those that were sent or dragged. If it's called when a reply isn't allowed, the *message* is deleted and an error is recorded.

The *message* that's passed to **SendReply()** should not be modified, passed to another messaging function, used as a model message, or deleted. It becomes the responsibility of the messaging service and the eventual receiver.

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

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

SendReply() sends a message—a reply message, to be sure, but a message nonetheless. It therefore is just another message-sending function. It behaves exactly like the other message-sending function, BMessenger's **SendMessage()**:

- By passing it a *reply* argument, you can ask for a synchronous reply to the reply message it sends. It won't return until it receives the reply.

- By supplying a *targetHandler* argument, you can arrange for an expected asynchronous reply. If a specific target isn't specified, the BApplication object will handle the reply if one is sent.

This function returns **B_NO_ERROR** if the reply is successfully sent. If not, it returns one of the error codes explained under the **Error()** function.

See also: **BMessenger::SendMessage()**, **BLooper::DetachCurrentMessage()**, **WasSent()**, **Error()**

Unflatten() *see* Flatten()

WasDropped(), DropPoint()

bool **WasDropped**(void)

BPoint **DropPoint**(BPoint **offset* = NULL)

WasDropped() returns **TRUE** if the user delivered the BMessage by dragging and dropping it, and **FALSE** if the message was posted or sent in application code or if it hasn't yet been delivered at all.

DropPoint() reports the point where the cursor was located when the message was dropped (when the user released the mouse button). It directly returns the point in the screen coordinate system and, if an *offset* argument is provided, returns it by reference in coordinates based on the image or rectangle the user dragged. The *offset* assumes a coordinate system with (0.0, 0.0) at the left top corner of the dragged rectangle or image.

Since any value can be a valid coordinate, **DropPoint()** produces reliable results only if **WasDropped()** returns **TRUE**.

See also: **BView::DragMessage()**

WasSent(), IsSourceRemote(), IsSourceWaiting(), IsReply(), Previous(), ReturnAddress()

bool **WasSent**(void)

bool **IsSourceRemote**(void)

bool **IsSourceWaiting**(void)

bool **IsReply**(void)

BMessage ***Previous**(void)

BMessenger **ReturnAddress**(void)

These functions can help if you're engaged in an exchange of messages or managing an ongoing communication.

WasSent() indicates whether it's possible to send a reply to a message. It returns **TRUE** for a BMessage that was sent or dropped, and **FALSE** for a message that was posted or has not yet been delivered by any means. (When, in a future release, it's possible to reply to a

posted message, this function would be more clearly named **WasDelivered()**.) Regardless of the return value, **WasSent()** sets the current error code to **B_NO_ERROR**.

IsSourceRemote() returns **TRUE** if the message had its source in another application, and **FALSE** if the source is local or the message hasn't been delivered yet. It resets the error code to **B_NO_ERROR** in both cases.

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

IsReply() returns **TRUE** if the BMessage is a reply to a previous message (if it was sent by the **SendReply()** function), and **FALSE** if not. It resets the error code to **B_NO_ERROR** in either case.

Previous() returns the previous message, or **NULL** if the BMessage isn't a reply.

ReturnAddress() returns a BMessenger that can be used to reply to the BMessage. Calling the BMessenger's **SendMessage()** function is equivalent to calling **SendReply()**, except that the return message won't be marked as a reply. If a reply isn't allowed (if the BMessage wasn't sent or dropped), a **B_BAD_VALUE** error is registered to indicate that the returned BMessenger is invalid. Call **Error()** to check. If the BMessenger is valid, **Error()** will return **B_NO_ERROR**.

If you want to use the **ReturnAddress()** BMessenger to send a synchronous reply, you must do so before the BMessage is deleted and default reply is sent.

See also: **BMessenger::SendMessage()**, **SendReply()**

Operators

new

void ***operator new**(size_t *numBytes*)

Allocates memory for a BMessage object, or takes the memory from a previously allocated cache. The caching mechanism is an efficient way of managing memory for objects that are created frequently and used for short periods of time, as BMessages typically are.

delete

void **operator delete**(void **memory*, size_t *numBytes*)

Frees memory allocated by the BMessage version of **new**, which may mean restoring the memory to the cache.

BMessageFilter

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

Overview

A BMessageFilter is an object that holds a hook function, **Filter()**, that can look at incoming messages before they're dispatched to their designated handlers. The object also keeps the conditions that must be met for the function to be called. Applications implement the **Filter()** function in classes derived from BMessageFilter.

A BMessageFilter can be attached to a message loop in one of two ways:

- If assigned to a BHandler object, the filter will be applied only to messages targeted to the BHandler.
- If assigned to a BLooper object as a common filter, it can be applied to any message regardless of the designated target. (A BLooper can also be assigned specific filters in its role as a BHandler.)

All applicable filters in both categories are applied to a message before it's dispatched to the target BHandler. Common filters are applied before handler-specific filters.

The same BMessageFilter object can be assigned to more than one BHandler or BLooper object; it will not be destroyed when the BHandler or BLooper is deleted.

See also: **BHandler::SetFilterList()**, **BLooper::SetCommonFilterList()**

Hook Functions

Filter()	Implemented by derived classes to respond to a incoming message before the message is dispatched to a target BHandler.

Constructor and Destructor

BMessageFilter()

BMessageFilter(message_delivery *delivery*, message_source *source***)**
BMessageFilter(message_delivery *delivery*, message_source *source*,
ulong *command***)**

Initializes the BMessageFilter object so that its **Filter()** function will be called for every incoming message that meets the specified *delivery*, *source*, and *command* criteria. The first argument, *delivery*, is a constant that specifies how the message must arrive:

B_DROPPED_DELIVERY	Only messages that were dragged and dropped should be filtered.
B_PROGRAMMED_DELIVERY	Only messages that were posted or sent in application code (by calling **PostMessage()** or a **Send...()** function) should be filtered.
B_ANY_DELIVERY	All messages, no matter how they were delivered, should be filtered.

The second argument, *source*, specifies where the message must originate:

B_LOCAL_SOURCE	Only messages that originate locally, from within the application, should be filtered.
B_REMOTE_SOURCE	Only messages that are delivered from a remote source should be filtered.
B_ANY_SOURCE	All messages, no matter what their source, should be filtered.

Filtering can also be limited to a particular type of message. If a *command* constant is specified, only messages that have **what** data members matching the constant will be filtered. If a *command* isn't specified, the command constant won't be a criterion in selecting which messages to filter; any message that meets the other criteria will be filtered, no matter what its **what** data member may be.

The filtering criteria are conjunctive; an arriving message must meet all the criteria specified for **Filter()** to be called.

See also: **Filter()**

~BMessageFilter()

virtual **~BMessageFilter(**void**)**

Does nothing.

Member Functions

Command(), FiltersAnyCommand()

inline ulong **Command**(void)

inline bool **FiltersAnyCommand**(void)

Command() returns the command constant (**what** data member) that an arriving message must match for the filter to apply. **FiltersAnyCommand()** returns **TRUE** if the filter applies to messages regardless of their **what** data members, and **FALSE** if it's limited to a certain type of message.

Because all command constants are valid, including negative numbers and 0, **Command()** returns a reliable result only if **FiltersAnyCommand()** returns **FALSE**.

See also: the BMessageFilter constructor, the BMessage class

Filter()

virtual filter_result **Filter**(BMessage **message*, BHandler ***target*)

Implemented by derived classes to examine an arriving message just before it's dispatched. The *message* is passed as the first argument; the second argument indirectly points to the *target* BHandler object that's slated to respond to the message.

You can implement this function to do anything you please with the *message*, including replace the designated *target* with another BHandler object. For example:

```
filter_result MyFilter::Filter(BMessage *msg, BHandler **target)
{
    . . .
    if ( *target->IsIndisposed() )
        *target = *target->FindReplacement();
    . . .
    return B_DISPATCH_MESSAGE;
}
```

The replacement target must be associated with the same BLooper as the original target. If the new target has filters that apply to the *message*, those filtering functions will be called before the message is dispatched.

This function should return a constant that instructs the BLooper whether or not to dispatch the message as planned:

B_DISPATCH_MESSAGE	Dispatch the message.
B_SKIP_MESSAGE	Don't dispatch the message and don't filter it any further; this function took care of handling it.

The default (BMessageFilter) version of this function does nothing but return **B_DISPATCH_MESSAGE**.

See also: the BMessageFilter constructor

FiltersAnyCommand() *see* Command()

MessageDelivery()

inline message_delivery **MessageDelivery**(void)

Returns the constant, set when the BMessageFilter object was constructed, that describes the category of messages that can be filtered, based on how they were delivered.

See also: the BMessageFilter constructor

MessageSource()

inline message_source **MessageSource**(void)

Returns the constant, set when the BMessageFilter object was constructed, that describes the category of messages that can be filtered, based on the source of the message.

See also: the BMessageFilter constructor

BMessageQueue

Derived from: public BObject

Declared in: <app/MessageQueue.h>

Class Description

A BMessageQueue maintains a queue where messages (BMessage objects) are temporarily stored as they wait to be received in a message loop. Every BLooper object uses a BMessageQueue to manage the flow of incoming messages; all messages delivered to the BLooper's thread are placed in the queue. The BLooper removes the oldest message from the queue, passes it to a BHandler, waits for the thread to finish its response, deletes the message, then returns to the queue to get the next message.

For the most part, applications can ignore the queue—that is, they can treat it as an implementation detail. Messages are posted to a thread (placed in the queue) by calling BLooper's **PostMessage()** function. Or they can be sent to the main thread of another application by constructing a BMessenger object and calling **SendMessage()**.

A BLooper calls upon a BHandler's **MessageReceived()** function—and other, message-specific hook functions—to handle the messages it takes from the queue. Applications can simply implement the functions that are called to respond to received messages and not bother about the mechanics of the message loop and queue.

However, if necessary, you can manipulate the queue directly, or perhaps just look ahead to see what messages are coming. The BLooper has a **MessageQueue()** function that returns its BMessageQueue object.

See also: the BMessage class, **BLooper::MessageQueue()**

Constructor and Destructor

BMessageQueue()

BMessageQueue(void)

Ensures that the queue starts out empty. Messages are placed in the queue by calling **AddMessage()** and are removed by calling **NextMessage()**.

BMessageQueues are constructed by BLooper objects.

See also: **AddMessage()**, **NextMessage()**

~BMessageQueue()

virtual **~BMessageQueue**(void)

Deletes all the objects in the queue and all the data structures used to manage the queue.

Member Functions

AddMessage()

void **AddMessage**(BMessage **message*)

Adds *message* to the queue.

See also: **NextMessage()**

CountMessages()

long **CountMessages**(void) const

Returns the number of messages currently in the queue.

FindMessage()

BMessage ***FindMessage**(ulong *what*, long *index* = 0) const
BMessage ***FindMessage**(long *index*) const

Returns a pointer to the BMessage that's positioned in the queue at *index*, where indices begin at 0 and count only those messages that have **what** data members matching the *what* value passed as an argument. If a *what* argument is omitted, indices count all messages in the queue. If an *index* is omitted, the first message that matches the *what* constant is found. The lower the index, the longer the message has been in the queue.

If no message matches the specified *what* and *index* criteria, this function returns **NULL**.

The returned message is not removed from the queue.

See also: **NextMessage()**

IsEmpty()

bool **IsEmpty**(void) const

Returns **TRUE** if the BMessageQueue contains no messages, and **FALSE** if it has at least one.

See also: **CountMessages()**

Lock(), Unlock()

bool **Lock**(void)

void **Unlock**(void)

These functions lock and unlock the BMessageQueue, so that another thread won't alter the contents of the queue while it's being read. **Lock()** doesn't return until it has the queue locked; it always returns **TRUE**. **Unlock()** releases the lock so that someone else can lock it. Calls to these functions can be nested.

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

NextMessage()

BMessage ***NextMessage**(void)

Returns the next message—the message that has been in the queue the longest—and removes it from the queue. If the queue is empty, this function returns **NULL**.

RemoveMessage()

void **RemoveMessage**(BMessage **message*)

Removes a particular *message* from the queue and deletes it.

See also: **FindMessage()**

Unlock() *see* Lock()

BMessenger

Derived from: public BObject

Declared in: <app/Messenger.h>

Overview

A BMessenger object is an agent for sending messages to a particular destination. Each BMessenger knows about a BLooper object and a specific BHandler for that BLooper. The messages it sends are delivered to the BLooper and—provided they're not system messages—dispatched by the BLooper to the BHandler. The destination objects can belong to the same application as the message sender, but typically are in a remote application. It's more efficient to post a message within the same application than to ask a BMessenger to send it.

BMessenger objects can be transported across application boundaries. You can create one for a particular BLooper/BHandler combination in your application, then pass it by value to a remote application. That application can then use the BMessenger to target the BHandler in your application. This is, in fact, the only way for an application to get a BMessenger that can target a remote object other than a BApplication object.

Constructor and Destructor

BMessenger()

BMessenger(ulong *signature*, team_id *team* = –1**)**
BMessenger(const BHandler **target***)**
BMessenger(const BMessenger *&messenger***)**
BMessenger(void**)**

Initializes the BMessenger so that it can send messages to an application identified by its *signature* or by its *team*. The application must be running when the BMessenger is constructed.

If the *signature* passed is **NULL**, the application is identified by its team only. If the *team* specified is –1, as it is by default, the application is identified by its signature only. If both a real *signature* and a valid *team* identifier are passed, they must match—the *team* must be for the application that the *signature* identifies. If more than one instance of the *signature* application happens to be running, the *team* picks out a particular instance as the

BMessenger's target. Without a valid *team* argument, the constructor arbitrarily picks one of the instances.

BMessengers constructed in this way send messages to the main thread of the remote application, where they're received and handled by that application's BApplication object. This type of messenger is needed to initiate communication with another application.

A BMessenger can also be an agent for a *target* BHandler object. It sends messages to the BLooper associated with the BHandler, and the BLooper dispatches them to the BHandler.

The *target* BHandler object must be able to tell the BMessenger (through its **Looper()** function) which BLooper object it's associated with. The BMessenger asks for this information at the time of construction. Therefore, the *target* must either be a BLooper itself or have been added to a BLooper's list of eligible handlers. < For the BMessenger to remain valid, the *target* BHandler must retain its affiliation with the same BLooper. >

The purpose of constructing a BMessenger for a local target is to give a remote application access to that object. You can add the BMessenger to a message and send the message to the remote application. That application can then use the BMessenger to target the BHandler in your application.

A BMessenger can also be constructed as a copy of another BMessenger,

```
BMessenger newOne(anotherMessenger);
```

or be assigned from another object:

```
BMessenger newOne = anotherMessenger;
```

If the constructor can't make a connection to the *signature* application—possibly because no such application is running—it registers a **B_BAD_VALUE** error, which the **Error()** function will return. If passed an invalid *team* identifier, it registers a **B_BAD_TEAM_ID** error. If the *team* and the *signature* don't match, it registers a **B_MISMATCHED_VALUES** error. If it can't discover a BLooper from the *target* BHandler, it registers a **B_BAD_HANDLER** error.

It's a good idea to check for an error before asking the new BMessenger to send a message. For example:

```
BMessenger *outlet = new BMessenger(some_signature);
if ( outlet->Error() == B_NO_ERROR ) {
    BMessage *msg = new BMessage(CHANGE_NAME);
    msg->AddString("old", formerName);
    msg->AddString("new", currentName);
    outlet->SendMessage(msg);
    if ( outlet->Error() == B_NO_ERROR )
        . . .
}
```

A BMessenger can send messages to only one destination. Once constructed, you can cache it and reuse it repeatedly to communicate with that object. It should be freed after it's no longer needed (or if there's a long delay between messages and it's possible that the

user might have quit the destination application and restarted it again, or that the application may have destroyed the target BHandler).

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

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

~BMessenger()

~BMessenger(void)

Frees all memory allocated by the BMessenger, if any was allocated at all.

Member Functions

Error()

long **Error**(void)

Returns an error code that describes what went wrong with the attempt to construct the BMessenger or to have it send a message, or **B_NO_ERROR** if nothing went wrong. Possible errors include:

B_BAD_VALUE	The constructor can't connect the BMessenger to the remote application, most likely because an application with the specified signature isn't running.
B_MISMATCHED_VALUES	The constructor failed because the specified signature and team arguments designated two different applications.
B_BAD_TEAM_ID	The constructor can't establish a connection to the specified team, most likely because there is no such team.
B_BAD_HANDLER	The BHandler passed to the constructor was not associated with a BLooper.
B_BAD_PORT_ID	**SendMessage()** can't deliver the message, most likely because the destination application has been killed.

Calling this function resets the error code to **B_NO_ERROR**, so you must cache the value returned if you need to check the current error more than once.

FindHandler(), FindAllHandlers()

BMessenger **FindHandler**(BMessage **message*)
BMessenger **FindHandler**(long *index*, const char **class* = NULL)
BMessenger **FindHandler**(const char **name*, const char **class* = NULL)

BMessage ***FindAllHandlers**(const char **class* = NULL)

These functions send a **B_HANDLERS_REQUESTED** message and wait for a **B_HANDLERS_INFO** reply.

The *message* is passed to **FindHandler()** should have **B_HANDLERS_REQUESTED** as the **what** data member and should ask for a BMessenger for just one BHandler. If an *index* or a *name* is passed instead of a *message*, **FindHandler()** creates the message and adds that information in an entry named "index" or "name". If the index or name is restricted to a *class*, it adds the class name in an entry labeled "class".

When it gets the reply, **FindHandler()** returns the requested BMessenger. It may register a **B_ERROR**, **B_NAMED_NOT_FOUND**, or **B_BAD_INDEX** error taken from the reply, or a **B_BAD_PORT_ID** error if there's a problem sending the message. In the case of any error, the BMessenger is not to be trusted.

FindAllHandlers() requests BMessengers for a group of BHandlers, which may be restricted to a particular *class*. It returns the **B_HANDLERS_INFO** reply, or **NULL** if there's an error in sending a message.

See the various **HandlersRequested()** functions for information on the protocols that the software kits currently expect the **B_HANDLERS_REQUESTED** and **B_HANDLERS_INFO** messages to follow.

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

IsValid()

bool **IsValid**(void)

Returns **TRUE** if the destination BLooper object to which the BMessenger sends messages remains valid, and **FALSE** if not (if, for example, it has been deleted).

This function doesn't check whether the target BHandler is still valid; it reports only on the status of the destination BLooper.

SendMessage()

long **SendMessage(**BMessage **message*, BMessage ***reply***)**
long **SendMessage(**ulong *command*, BMessage ***reply***)**
long **SendMessage(**BMessage **message*, BHandler **replyTarget* = NULL**)**
long **SendMessage(**ulong *command*, BHandler **replyTarget* = NULL**)**

Sends a *message*. The BMessage object becomes the responsibility of the BMessenger. You shouldn't try to modify it, post it, send it again, use it as a model message, or free it; it will be freed automatically when it's no longer needed.

If a *command* is passed instead of a full *message*, this function constructs a BMessage object with *command* as its **what** data member and sends it just like any other message. This is simply a convenience for sending messages that contain no data. The following two lines of code are roughly equivalent:

```
myMessenger->SendMessage(NEVERMORE);
myMessenger->SendMessage(new BMessage(NEVERMORE));
```

This function can ask for a synchronous reply to the message or designate a BHandler for an asynchronous reply:

- Supplying a *reply* argument requests a message back from the destination. Before returning, **SendMessage()** waits for the reply and places a pointer to the BMessage it receives in the variable that *reply* refers to.

  The caller is responsible for deleting the *reply* message. If the destination doesn't send a reply, the system sends one with **B_NO_REPLY** as the **what** data member. Check the reply message before proceeding. If there's an error in sending the message, the variable that *reply* refers to is set to **NULL**.

- If a *reply* isn't requested, **SendMessage()** returns immediately; any reply to the *message* will be received asynchronously. If a *replyTarget* is specified, the reply will be directed to that BHandler object. If one isn't specified, it will be directed to the BApplication object.

  The *replyTarget* is subject to the same restriction as a target BHandler passed to the BMessenger constructor: It must be associated with a BLooper object (or be a BLooper itself) < and it must retain that association until the reply arrives >.

If all goes well, **SendMessage()** returns **B_NO_ERROR**. If not, it returns an error code, typically **B_BAD_PORT_ID**. The return value is also registered with the **Error()** function; see that function for more information.

(It's an error for a thread to send a message to itself and expect a synchronous reply. The thread can't respond to the message and wait for a reply at the same time.)

See also: **BMessage::SendReply()**

Team()

inline team_id **Team**(void)

Returns the identifier for the team that receives the messages the BMessenger sends.

Operators

= (assignment)

BMessenger &**operator =**(const BMessenger &*messenger*)

Assigns one BMessenger to another. After the assignment the two objects are identical and independent copies of each other, with no shared data.

new

void ***operator new**(size_t *numBytes*)

Prevents confusion with a private version of the **new** operator used internally by the Application Kit. This version of **new** is no different from the operator used with other classes.

BRoster

Derived from: *none*

Declared in: <app/Roster.h>

Overview

The BRoster object keeps a roster of all applications currently running on the BeBox. It can provide information about any of those applications, add another application to the roster by launching it, or get information about an application to help you decide whether to launch it.

There's just one roster and it's shared by all applications. When an application starts up, a global variable, be_roster, is initialized to point to the shared object. You always access the roster through this variable; you never directly instantiate a BRoster in application code.

The BRoster identifies applications in three ways:

- By record_ref references to the executable files where they reside.
- By their signatures. The signature is a unique identifier for the application assigned in a resource at compile time or by the BApplication constructor at run time. You can obtain signatures for the applications you develop by contacting Be's developer support staff. They can also tell you what the signatures of other applications are. (See the introduction to this chapter for more on signatures.)
- At run time, by their team_ids. A team is a group of threads sharing an address space; every application is a team.

If an application is launched more than once, the roster will include one entry for each instance of the application that's running. These instances will have the same signature, but different team identifiers.

Constructor and Destructor

The BRoster class doesn't have a public constructor or destructor. This is because an application doesn't need to construct or destroy a BRoster of its own. The system constructs one BRoster object for all applications and assigns it to the **be_roster** global variable. A BRoster is therefore readily available from the time the application is launched until the time it quits.

Member Functions

GetAppInfo(), GetRunningAppInfo(), GetActiveAppInfo()

long **GetAppInfo**(ulong *signature*, app_info **appInfo*) const
long **GetAppInfo**(record_ref *executable*, app_info **appInfo*) const

long **GetRunningAppInfo**(team_id *team*, app_info **appInfo*) const

long **GetActiveAppInfo**(app_info **appInfo*) const

These functions provide information about the application identified by its *signature*, by a database reference to its *executable* file, by its *team*, or simply by its status as the current active application. They place the information in the structure referred to by *appInfo*.

GetRunningAppInfo() reports on a particular instance of a running application, the one that was assigned the *team* identifier at launch. **GetActiveAppInfo()** similarly reports on a running application, the one that happens to be the current active application.

If it can, **GetAppInfo()** also tries to get information about an application that's running. If a running application has the *signature* identifier or was launched from the *executable* file, **GetAppInfo()** queries it for the information. If more than one instance of the *signature* application is running, or if more than one instance was launched from the same *executable* file, it arbitrarily picks one of the instances to report on.

Even if the application isn't running—if none of the applications currently in the roster are identified by *signature* or were launched from the *executable* file—**GetAppInfo()** can still provide some information about it, perhaps enough information for you to call **Launch()** to get it started.

If they're able to fill in the **app_info** structure with meaningful values, these functions return **B_NO_ERROR**. However, **GetActiveAppInfo()** returns **B_ERROR** if there's no active application. **GetRunningAppInfo()** returns **B_BAD_TEAM_ID** if *team* isn't, on the face of it, a valid team identifier for a running application. **GetAppInfo()** returns **B_BAD_VALUE** if the *signature* doesn't correspond to an application on-disk, and simply **B_ERROR** if the *executable* doesn't refer to a valid record in the database or doesn't refer to a record for an executable file.

The **app_info** structure contains the following fields:

ulong **signature**	The signature of the application. (This will be the same as the *signature* passed to **GetAppInfo()**.)
thread_id **thread**	The identifier for the application's main thread of execution, or –1 if the application isn't running. (The main thread is the thread in which the application is launched and in which its **main()** function runs.)
team_id **team**	The identifier for the application's team, or –1 if the application isn't running. (This will be the same as the *team* passed to **GetRunningAppInfo()**.)
port_id **port**	The port where the application's main thread receives messages, or –1 if the application isn't running.
record_ref **ref**	A reference to the file that was, or could be, executed to run the application. (This will be the same as the *executable* passed to **GetAppInfo()**.)
ulong **flags**	A mask that contains information about the behavior of the application.

The **flags** mask can be tested (with the bitwise **&** operator) against these two constants:

B_BACKGROUND_APP	The application won't appear in the Browser's application menu (because it doesn't have a user interface).
B_ARGV_ONLY	The application can't receive messages. Information can be passed to it at launch only, in an array of argument strings (as on the command line).

The **flags** mask also contains a value that explains the application's launch behavior. This value must be filtered out of **flags** by combining **flags** with the **B_LAUNCH_MASK** constant. For example:

```
ulong behavior = theInfo.flags & B_LAUNCH_MASK;
```

The result will match one of these three constants:

B_EXCLUSIVE_LAUNCH	The application can be launched only if an application with the same signature isn't already running.
B_SINGLE_LAUNCH	The application can be launched only once from the same executable file. However, an application

	with the same signature might be launched from a different executable. For example, if the user copies an executable file to another directory, a separate instance of the application can be launched from each copy.
B_MULTIPLE_LAUNCH	There are no restrictions. The application can be launched any number of times from the same executable file.

These flags affect BRoster's **Launch()** function. **Launch()** can always start up a **B_MULTIPLE_LAUNCH** application. However, it can't launch a **B_SINGLE_LAUNCH** application if a running application was already launched from the same executable file. It can't launch a **B_EXCLUSIVE_LAUNCH** application if an application with the same signature is already running.

See also: "Launch Information" on page 19 of the chapter introduction, **Launch()**, **BApplication::GetAppInfo()**

GetAppList()

void **GetAppList(**BList **teams***)** const
void **GetAppList(**ulong *signature*, BList **teams***)** const

Fills in the *teams* BList with team identifiers for applications in the roster. Each item in the list will be of type **team_id**. It must be cast to that type when retrieving it from the list, as follows:

```
team_id who = (team_id)teams->ItemAt(someIndex);
```

The list will contain one item for each instance of an application that's running. For example, if the same application has been launched three times, the list will include the **team_id**s for all three running instances of that application.

If a *signature* is passed, the list identifies only applications running under that signature. If a *signature* isn't specified, the list identifies all running applications.

See also: **TeamFor()**, the BMessenger constructor

IsRunning() *see* TeamFor()

Launch()

long **Launch(**ulong *signature*, BMessage **message* = NULL,
team_id **team* = NULL**)**
long **Launch(**ulong *signature*, BList **messages*,
team_id **team* = NULL**)**
long **Launch(**ulong *signature*, long *argc*, char ***argv*,
team_id **team* = NULL**)**

long **Launch(**record_ref *executable*, BMessage **message* = NULL,
team_id **team* = NULL**)**
long **Launch(**record_ref *executable*, BList **messages*,
team_id **team* = NULL**)**
long **Launch(**record_ref *executable*, long *argc*, char ***argv*,
team_id **team* = NULL**)**

Launches the application identified by its *signature* or by a reference to its *executable* file in the database.

If a *message* is specified, it will be sent to the application on-launch where it will be received and responded to before the application is notified that it's ready to run. Similarly, if a list of *messages* is specified, each one will be delivered on-launch. The BMessage objects (and the container BList) will be deleted for you.

Sending an on-launch message is appropriate only if it helps the launched application configure itself before it starts getting other messages. To launch an application and send it an ordinary message, call **Launch()** to get it running, then set up a BMessenger object for the application and call BMessenger's **SendMessage()** function.

Instead of messages, you can launch an application with an array of argument strings that will be passed to its **main()** function. *argv* contains the array and *argc* counts the number of strings. If the application accepts messages, this information will also be packaged in a **B_ARGV_RECEIVED** message that the application will receive on-launch.

If successful, **Launch()** places the identifier for the newly launched application in the variable referred to by *team* and returns **B_NO_ERROR**. If unsuccessful, it sets the *team* variable to –1, destroys all the messages it was passed (and the BList that contained them), and returns one of the following error codes:

B_BAD_VALUE	The *signature* passed is not valid or it doesn't designate an available application. This return value may also signify that an attempt is being made to send an on-launch message to an application that doesn't accept messages (that is, to a **B_ARGV_ONLY** application).
B_ERROR	The *executable* file can't be found.

B_ALREADY_RUNNING	The application is already running and can't be launched again (it's a **B_SINGLE_LAUNCH** or **B_EXCLUSIVE_LAUNCH** application).
B_LAUNCH_FAILED	The attempt to launch the application failed for some other reason, such as insufficient memory.

See also: the BMessenger class, **GetAppInfo()**

RemoveApp()

void **RemoveApp(**team_id *team***)**

Removes the application identified by *team* from the roster of running applications.

TeamFor(), IsRunning()

team_id **TeamFor(**ulong *signature***)** const
team_id **TeamFor(**record_ref *executable***)** const

bool **IsRunning(**ulong *signature***)** const
bool **IsRunning(**record_ref *executable***)** const

Both these functions query whether the application identified by its *signature*, or by a reference to its *executable* file in the database, is running. **TeamFor()** returns its team identifier if it is, and **B_ERROR** if it's not. **IsRunning()** returns **TRUE** if it is, and **FALSE** if it's not.

If the application is running, you probably will want its team identifier (to set up a BMessenger, for example). Therefore, it's most economical to simply call **TeamFor()** and forego **IsRunning()**.

If more than one instance of the *signature* application is running, or if more than one instance was launched from the same *executable* file, **TeamFor()** arbitrarily picks one of the instances and returns its **team_id**.

See also: **GetAppList()**

Global Variables, Constants, and Defined Types

This section lists the global variables, constants, and defined types that are defined by the Application Kit. There's just a few defined types, three global variables—**be_app**, **be_roster**, and **be_clipboard**—and a handful of constants. Error codes are documented in the chapter on the Support Kit.

Although the Application Kit defines the constants for all system messages (such as **B_REFS_RECEIVED**, **B_ACTIVATE**, and **B_KEY_DOWN**), only those that mark system management and application messages are listed here. Those that designate interface messages are documented in the chapter on the Interface Kit.

Global Variables

be_app

<app/Application.h>

BApplication ***be_app**

This variable provides global access to your application's BApplication object. It's initialized by the BApplication constructor.

See also: the BApplication class

be_clipboard

<app/Clipboard.h>

BClipboard ***be_clipboard**

This variable gives applications access to the shared repository of data for cut, copy, and paste operations. It's initialized at startup; an application has just one BClipboard object.

See also: the BClipboard class

be_roster

<app/Roster.h>

BRoster ***be_roster**

This variable points to the global BRoster object that's shared by all applications. The BRoster keeps a roster of all running applications and can add applications to the roster by launching them.

See also: the BRoster class

Constants

Application Flags

<app/Roster.h>

Defined constant
B_BACKGROUND_APP
B_ARGV_ONLY
B_LAUNCH_MASK

These constants are used to get information from the **flags** field of an **app_info** structure.

See also: **BRoster::GetAppInfo()**, "Launch Constants" below

Application Messages

<app/AppDefs.h>

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

These constants represent the system messages that are received and recognized by the BApplication class. Application messages concern the application as a whole, rather than any particular window thread. See the introduction to this chapter and the BApplication class for details.

See also: "Application Messages" on page 16 of the chapter introduction, "System Management Messages" on page 113 below

Cursor Constants

<app/AppDefs.h>

const unsigned char **B_HAND_CURSOR**[]
const unsigned char **B_I_BEAM_CURSOR**[]

These constants contain all the data needed to set the cursor to the default hand image or to the standard I-beam image for text selection.

See also: **BApplication::SetCursor()**

Data Type Codes

<app/AppDefs.h>

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

These constants are used in a BMessage object to describe the types of data the message holds. **B_ANY_TYPE** refers to all types; the others refer only to a particular type. See the BMessage class for more information on what they mean.

See also: "Type Codes" on page 71 of the BMessage class overview

filter_result Constants

<app/MessageFilter.h>

Enumerated constant

B_SKIP_MESSAGE
B_DISPATCH_MESSAGE

These constants list the possible return values of a filter function.

See also: **BMessageFilter::Filter()**

Launch Constants

<app/Roster.h>

Defined constant

B_MULTIPLE_LAUNCH
B_SINGLE_LAUNCH
B_EXCLUSIVE_LAUNCH

These constants explain whether an application can be launched any number of times, only once from a particular executable file, or only once for a particular application signature. This information is part of the **flags** field of an **app_info** structure and can be extracted using the **B_LAUNCH_MASK** constant.

See also: **BRoster::GetAppInfo()**, “Application Flags” above

Message Constants

<app/AppDefs.h>

Enumerated constant

B_NO_REPLY
B_MESSAGE_NOT_UNDERSTOOD

B_HANDLERS_INFO

B_SIMPLE_DATA

B_CUT
B_COPY
B_PASTE

These constants mark messages that the system sometimes puts together, but that aren’t dispatched like system messages. See “Standard Messages” in the *Message Protocols* appendix for details.

See also: **BMessage::SendReply()**, the BTextView class in the Interface Kit, **BHandler::HandlersRequested()**

message_delivery Constants

<app/MessageFilter.h>

Enumerated constant

B_ANY_DELIVERY
B_DROPPED_DELIVERY
B_PROGRAMMED_DELIVERY

These constants distinguish the delivery criterion for the application of a BMessageFilter.

See also: the BMessageFilter constructor

message_source Constants

<app/MessageFilter.h>

Enumerated constant

B_ANY_SOURCE
B_REMOTE_SOURCE
B_LOCAL_SOURCE

These constants list the possible constraints on the message source for the application of a BMessageFilters.

See also: the BMessageFilter constructor

System Management Messages

<app/AppDefs.h>

Enumerated constant

B_QUIT_REQUESTED
B_HANDLERS_REQUESTED

These constants represent system messages that are used to help run the messaging system. They're received and recognized by generic BLooper objects.

See also: "System Management Messages" on page 15 of the introduction, "Application Messages" on page 110 above

Defined Types

app_info

<app/Roster.h>

```
typedef struct {
    ulong signature;
    thread_id thread;
    team_id team;
    port_id port;
    record_ref ref;
    ulong flags;
} app_info
```

This structure is used by BRoster's **GetAppInfo()**, **GetRunningAppInfo()**, and **GetActiveAppInfo()** functions to report information about an application. See those functions for a description of its various fields.

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

filter_result

<app/MessageFilter.h>

typedef enum { . . . } **filter_result**

This type distinguishes between the **B_SKIP_MESSAGE** and **B_DISPATCH_MESSAGE** return values for a filter function.

See also: **BMessageFilter::Filter()**

message_delivery

<app/MessageFilter.h>

typedef enum { . . . } **message_delivery**

This type enumerates the delivery criteria for filtering a message.

See also: the BMessageFilter constructor

message_source

<app/MessageFilter.h>

typedef enum { . . . } **message_source**

This type enumerates the source criteria for filtering a message.

See also: the BMessageFilter constructor