The Application Kit

The Application Kit − Table of Contents

The Application Kit

The Application Kit is the starting point for all applications. Its classes establish an application as an identifiable entityone that can cooperate and communicate with other applications.

The Application Kit is divided into these topics:

- *The BApplication class*. All but the simplest of applications must have one (and only one) BApplication object. This object is typically an instance of a BApplication subclass that you create especially for your application. The BApplication object makes a connection to the App Server and runs the application's main message loop.

- *Messaging*. The kit provides a messaging service that lets threads talk to each other. This service can deliver messages within your own application, or from one application to another. It's also used by the system to deliver user event messages (key clicks, mouse moves) to your application. Most of the Application Kit's classes are involved in the messaging system.

- *Scripting*. The objects that you create can be controlled by commands issued from other applications.

- *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 BCursor class*. You use BCursor objects to represent distinct cursors. Functions defined by BApplication and BView let you assign your cursors to your entire application, or to individual views .

Messaging

The Application Kit provides a message−passing system lets your application send messages to and receives messages from other applications (including the Be−defined servers and apps), and from other threads in your own application.

The primary messaging classes are:

- BMessage represents a message.
- BLooper runs a loop that receives in−coming messages and figures out which BHandler should handle them.
- BHandler defines hook functions that are called to handle in−coming messages.
- BMessenger represents a message's destination (a combination of BLooper and BHandler), whether it's local or remote. The object is most useful for sending messages to other applicationsyou don't need it for local calls.

The other messaging classes are:

- BMessageQueue is a fifo that holds a BLooper's in−coming messages.
- BMessageFilter is a device that can examine and (potentially) reject or re−route in−coming messages.
- BInvoker is a convenience class that lets you treat a message and its target (the BHandler that will handle the message) as a single object.
- BMessageRunner lets you send the same message over and over, at regular intervals.

The rest of this chapter looks at...

- The essential features of the four fundamental classes. ("Features of the Fundamental Classes")
- How a BLooper decides which BHandler should handle an in−coming message. ("From Looper to Handler")
- The different methods for sending messages and receiving replies. ("Sending a Message").

describes how the classes fit together in the messaging system with an emphasis on what you can do in your application to take part.

Features of the Fundamental Classes

Looked at collectively, the four fundamental messaging classes comprise a huge chunk of API. Fortunately, the essential part of this API is pretty small; that's what we're going to look at here.

The BMessage Class

In the BMessage class, there's one essential data member, and two essential functions:

- The **`what`** data member is an arbitrary **`uint32`** value that describes (symbolically) what the message is about. You can set and examine **`what`** directlyyou don't have to use functions to get to it. The **`what`** value is called the object's *command constant*. The BeOS defines some number of command constants (such as **`B_QUIT_REQUESTED`**, and **`B_MOUSE_DOWN`**), but you'll also be creating constants of your own. Keep in mind that the constant's value is meaninglessit's just a code that identifies the "intent" of the message (and it's only meaningful if the receiver recognizes the constant).
- The two essential functions are **`AddData()`** and **`FindData()`**. These functions add data to a message you're about to send, and retrieve it from a message you just received. A BMessage can hold any amount of data; each data item (or "field") is identified by name, type, and index. For example, you can ask a message for the third boolean value named "IsEnabled" that it contains. In general, you use type−specific functions such as **`Add/FindString()`** and **`Add/FindInt32()`** rather than **`Add/FindData()`**. The query we just posed would actually look like this:

```
/* The args are:  name, index, value (returned by reference) */
bool returnValue;
aMessage.FindBool("IsEnabled", 2, &returnValue);
```

In summary, a BMessage contains **(1)** a command constant and **(2)** a set of data fields. Every BMessage that's used in the messaging system must have a command constant, but not every object needs to have data fields. (Other parts of the BeOS use BMessages for their data only. The BClipboard object, for example, ignores a BMessage's command constant.)

When discussing system−generated BMessage objects, we refer to the object by its command constant. For example, "a **`B_MOUSE_DOWN`**" means "a BMessage that has **`B_MOUSE_DOWN`** as its command constant".

Notice that a BMessage doesn't know how to send itself. However, as we'll see later, it does know how to reply to its sender once it's in the hands of the recipient.

The BLooper Class

BLooper's role is to receive messages and figure out what to do with them. There are four parts to this job, embodied in these functions:

- Every BLooper spawns a thread in which it runs a message loop. It's in this thread that the object receives messages. But you have to kick the BLooper to get it to run; you do this by calling the **`Run()`** function. When you're done with the obejctwhen you no longer need it to receive messagesyou call **`Quit()`**.
- Although you never invoke it directly, **`DispatchMessage()`** is the guts of the message loop. All messages that the looper receives show up in individual invocations of **`DispatchMessage()`**. The function decides where the message should go next, which is mostly a matter of deciding whether **(1)** the message should be handled by a system−defined hook function, or **(2)** passed to BHandler's **`MessageReceived()`** function (which we'll talk about in a moment). Three other important aspects of **`DispatchMessage()`** are...
- It runs in the BLooper's message thread (or *message loop*); this is a separate thread that the object spawns specifically to receive in−coming messages.
- Individual **`DispatchMessage()`** invocations are synchronous with regard to the loop. In other words, when a message shows up, **`DispatchMessage()`** is called and runs to completion before the next message can be processed. (Messages that show up while **`DispatchMessage()`** is busy aren't lostthey're queued up in a BMessageQueue object.)
- It's fully implemented by BLooper (and kit classes derived from BLooper). You should rarely need to override it in your application.
- The **`PostMessage()`** function delivers a message to the BLooper. In other words, it invokes **`DispatchMessage()`** in the looper's message thread. You call **`PostMessage()`** directly in your code. For example, here we create a BMessage and post it to our BApplication object (BApplication inherits from BLooper):

```
/* This form of the BMessage constructor sets the command constant. */
be_app->PostMessage(new BMessage(YOUR_CONSTANT_HERE))
```

In the Be kits, the BApplication and BWindow classes inherit from BLooper.

The BHandler Class

BHandler objects are called upon to handle the messages that a BLooper receives. A BHandler depends on two essential function:

- **`MessageReceived()`** is the function that a BLooper calls to dispatch an in−coming message to the BHandler (the BMessage is passed as the function's only argument). This is a hook function that a BHandler subclass implements to handle the different types of messages that it expects to receive. Most implementations examine the message's command constant and go from there. A typical outline looks like this:

```
 void MyHandler::MessageReceived(BMessage *message)
{
   /* Examine the command constant. */
   switch ( message->what ) {

   case YOUR_CONSTANT_HERE:
      /* Call a function that handles this sort of message. */
      HandlerFunctionA();
      break;

   case ANOTHER_CONSTANT_HERE:
      /* ditto */
      HandlerFunctionB();
      break;

   default:
      /* Messages that your class doesn't recognize must be passed
       * on to the base class implementation. */
      baseClass::MessageReceived(message);
      break;
   }
}
```

- BHandler's other essential function is defined by BLooper: **`BLooper::AddHandler()`**. This function adds the (argument) BHandler object to the (invoked−upon) BLooper's list of candidate handlers (its *handler chain*). If a BHandler wants to handle messages that are received by a BLooper, it must first be added to the BLooper's handler chain.

BLooper inherits from BHandler, and automatically adds itself to its own handler chain. This means that a BLooper can handle the messages that it receivesthis is the default behaviour for most messages. We'll examine this issue in depth later in this chapter.

The other classes that inherit from BHandler are BView and BShelf (both in the Interface Kit).

The BMessenger Class

A BMessenger's most important feature is that it can send a message to a remote application. To do this takes two steps, which point out the class' essential features:

- You identify the application that you want to send a message to (the "target") in the BMessenger constructor. An application is identified by its *app signature* (a MIME string).
- The **`SendMessage()`** function sends a message to the target.

BMessengers can also be used to target local looper/handler pairs.

From Looper to Handler

A BLooper pops a message from its message queue and, within its **`DispatchMessage()`** function, dispatches the message by invoking a BHandler function. But **(1)** which BHandler and **(2)** which function?

Finding a Handler

First, let's answer the "which BHandler" question. To determine which BHandler should handle an in−coming message, a BLooper follows these steps:

1. *Does the BMessage target a specific BHandler?* Both the **`BLooper::PostMessage()`** and **`BMessenger::SendMessage()`** functions provide additional arguments that let you specify a *target handler* that you want to have handle the message (you can also set the target in the BMessenger constructor). If a BHandler is specified, the BMessage will show up in that object's **`MessageReceived()`** function (or it will invoke a message−mapped hook function, as explained below).

2. *Does the BLooper designate a preferred handler?* Through its **`SetPreferredHandler()`** function, a BLooper can designate one of the objects in its handler chain as its *preferred handler*, and passes all messages to that object.

3. *The BLooper handles the BMessage itself.* If there's no target handler or preferred handler designation, the BLooper handles the message itselfin other words, the message is passed to the BLooper's own **`MessageReceived()`** function (or message−mapped hook).

We should mention here that both the BApplication and the BWindow class fine−tune this decision process a bit. However, the meddling that they do only applies to system messages (described below). The messages that you define yourself (i.e. the command constants that your application defines) will always follow the message dispatching path described here.

If you look at the **`DispatchMessage()`** protocol, you'll notice that it has a BMessage and a BHandler as arguments. In other words, the "which handler" decision described here is actually made before **`DispatchMessage()`** is called. In general, this is an implementation detail that you shouldn't worry about. If you want to think that **`DispatchMessage()`** makes the decisionand we've done nothing to disabuse you of this notiongo ahead and think it.

Finding a Function

As described above, a BLooper passes a BMessage to a BHandler by invoking the latter's **`MessageReceived()`** function. This is true of all messages that you create yourself, but it isn't true of certain messages that the system defines and sends. These system−generated messages (or *system messages*)particularly those that report user events such as **`B_MOUSE_DOWN`** or **`B_APP_ACTIVATED`**invoke specific hook functions.

For example, when the user presses a key, a **`B_KEY_DOWN`** message is sent to the active BWindow object.. From within its **`DispatchMessage()`** function, BWindow invokes the **`MouseDown()`** function of the BView that currently holds *keyboard focus*. (When a BView is made the focus of keyboard events, its window promotes it to preferred handler.)

So the question of "which function" is fairly simple: If the BMessage is a system message that's mapped to a hook function, the hook function is invoked. If it's not mapped to a hook function, the BHandler's **`MessageReceived()`** function is invoked.

A full list of system messages and the hook functions that they're mapped to is given in the System Messages Appendix. Note that not all system messages are mapped to hook functions; some of them do show up in **`MessageReceived()`**.

Inheritance and the Handler Chain

Let's look at **`MessageReceived()`** again. It was asserted, in a code snippet shown earlier, that a typical **`MessageReceived()`** implementation should include an invocation of the base class' version of the function:

```
void MyHandler::MessageReceived(BMessage *message)
{
   switch ( message->what ) {

   /* Command constants that you handle go here. */

   default:
      /* Messages that your class doesn't recognize must be passed
       * on to the base class implementation. */
      baseClass::MessageReceived(message);
      break;
   }
}
```

This isn't just a good ideait's an essential part of the messaging system. Forwarding the message to the base class does two things: It lets messages **(1)** pass up the class hierarchy, and **(2)** pass along the handler chain (in that order).

Passing up the class hierarchy is mostly straight−forwardit's no different for the **`MessageReceived()`** function than it is for any other function. But what happens at the top of the hierarchyat the BHandler class itselfadds a small wrinkle. BHandler's implementation of **`MessageReceived()`** looks for the next handler in the BLooper's handler chain and invokes that object's **`MessageReceived()`** function.

Sending a Message

There are two functions that send messages to distinct recipients:

- **`BLooper::PostMessage()`** can be used if the target (the BLooper that the **`PostMessage()`** function is invoked upon) lives in the same application as the message sender.

- **`BMessenger::SendMessage()`** lets you send messages to remote applications. The BMessenger object acts as a proxy for the remote app. (**`SendMessage()`** can also be used to send a message to a local BLooper, for reasons that we'll discuss later.)

The PostMessage() Function

You can post a message if the recipient BLooper is in your application:

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), targetHandler);
```

As shown here, you can specify the handler that you want to have handle a posted message. The only requirement is that the BHandler must belong to the BLooper.

If the handler argument is NULL, the message is handled by the looper's *preferred handler*

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), NULL);
```

By using the default handler, you let the looper decide who should handle the message.

The creator of the BMessage retains ownership and is responsible for deleting it when it's no longer needed.

The SendMessage() Function

If you want to send a message to another application, you have to use BMessenger's **`SendMessage()`** function. First, you construct a BMessenger object that identifies the remote app by signature...

```
BMessenger messenger("application/x-some-app");
```

...and then you invoke **`SendMessage()`**:

```
messenger.SendMessage(new BMessage(DO_SOMETHING));
```

The creator of the BMessage retains ownership and is responsible for deleting it when it's no longer needed.

Handling a Reply

Every BMessage that you send identifies the application from which it was sent. The recipient of the message can reply to the message whether you (the sender) expect a reply or not. By default, reply messages are handled by your BApplication object. If you want reply messages to be handled by some other BHandler, you specify the object as a final argument to the **`PostMesssage()`** or **`SendMessage()`** call:

```
myLooper->PostMessage(new BMessage(DO_SOMETHING), targetHandler, replyHandler);
/* and */
myMessenger.SendMessage(&message, replyHandler);
```

The reply is sent asynchronously with regard to the PostMessage()/SendMessage() function.

SendMessage() (only) lets you ask for a reply message that's handed back synchronously in the SendMessage() call itself:

```
BMessage reply;
myMessenger.SendMessage(&message, &reply);
```

`SendMessage()` doesn't return until a reply is received. A default message is created and returned if the recipient doesn't respond quickly enough.

Receiving a Message

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 message(READY);
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.

Handler Associations

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.

Handlers that belong to the same BLooper can be chained in a linked list. If an object can't respond to a message, the system passes the message to its next handler.

BLooper's **`AddHandler()`** function sets up the looper−handler association; BHandler's **`SetNextHandler()`** sets the handler−handler link.

Message Filters

The BMessageFilter class lets create filtering functions that examine and re−route (or reject) incoming messages before they're processed by a BLooper. Message filters can also be applied to individual BHandler objects.

Message Protocols

Both the source and the destination of a message must agree upon its formatthe command constant and the names and types of data fields. They must also agree on details of the exchangewhen 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.

Scripting

Scripting provides a means for programatically controlling some other application by sending it special scripting commands. These commands are defined by the "scripted" application itself. For example, if you want some other application to be able to tell your application to perform the "FlipOver" operation, you have to publish the format of the "FlipOver" command. The set of operations that you want to expose is called a "suite."

The BeOS defines some number of suites that correspond to particular classes. For example, all BApplication objects respond to the commands defined in the "vnd.Be−application" suite. One of the commands in the suite gives you access to the application's windows. When you've located the window that you want, you can move it, close it, resize it, and so on, according to the commands in the "vnd.Be−window" suite.

Basics

The scripting framework defines the following notions: commands, properties, and specifiers. If you are familiar with AppleScript, these are equivalent to verbs, nouns, and adjectives. Commands act on a specific instance of a property, as determined by the specifiers.

Commands

The command conveys the action of a scripting command and is stored in the *what* field of the scripting BMessage. There are six standard commands (defined in **be/app/Message.h**):

- **`B_COUNT_PROPERTIES`** counts the number of instances of a property.
- **`B_CREATE_PROPERTY`** creates a new instance of a property.
- **`B_DELETE_PROPERTY`** destroys an instance of a property.
- **`B_EXECUTE_PROPERTY`** executes an instance of a property.
- **`B_GET_PROPERTY`** gets the value of an instance of a property.
- **`B_SET_PROPERTY`** sets of the value of an instance of a property. The "data" field contains the new value of the property.

Each of these commands acts on a "property," which is nothing more than a scriptable feature of an object. As a real world example, the windows owned by an application are properties, as is the title of each window. The particular interpretation of the command depends upon the property it is acting on. For example, **`B_DELETE_PROPERTY`**, acting on the "Entry" property of a Tracker window, causes a file to be moved to the trash. However, the same command acting on the "Selection" property of the same window removes files from the list of selected items.

Scriptable objects should limit themselves to this set of commands. If an object uses a nonstandard command, it runs the risk of being unusable by general scripting tools.

Properties and Specifiers

A property represents a scriptable feature of an object. Properties are named; these names are strings unique within a class. For example, a BWindow defines properties such as "Frame," "Title," and "View." The data type of the property and its allowable values are determined by the property. For example, the window's "Frame" accepts **`B_RECT_TYPE`** values while the "Title" is a **`B_STRING_TYPE`**.

Sometimes a property is represented by another object. For example, BWindow's "View" designates a BView, an object which has a set of properties distinct from those of BWindow.

An object may have more than one instance of a given property. For example, the "Window" property of BApplication, has as many instances as there are windows in the application. As a result, there is some ambiguity when you ask for *the* Window of an application. Instead, it's more correct to ask for the first Window, or the Window named "Snyder." In other words, a property is not enough to identify a feature; a specific instance must be picked out as well.

Specifiers are used to target ("specify") particular instances of properties. A specifier is a BMessage containing the following elements:

- The name of the property in the "property" field, stored as a **`B_STRING_TYPE`**.
- The specifier constant, indicating a method of identifying a specific instance of the property, in the *what* field along with any necessary supporting fields.

There are seven standard specifier constants (defined in **<be/app/Message.h>**):

- **`B_DIRECT_SPECIFIER`** . The property name is sufficient specification by itself, usually because there's only one instance of the property. If there's more than one value for the property, a **`B_DIRECT_SPECIFIER`** would specify them all.
- **`B_NAME_SPECIFIER`** . The specifier message has a "name" field of type **`B_STRING_TYPE`** with the name of a particular instance of the property.
- **`B_ID_SPECIFIER`** . The specifier message has an "id" field with a unique identifying value of type **`int32`** for a particular instance of the property.
- **`B_INDEX_SPECIFIER`** . The specifier message has an **`int32`** field named "index" with the index to a particular instance of the property.
- **`B_REVERSE_INDEX_SPECIFIER`** . The index counts backwards from the end of the list.
- **`B_RANGE_SPECIFIER`** . In addition to an "index" field, the specifier message has an additional **`int32`** field named "range", identifying "range" items beginning at "index."

- **`B_REVERSE_RANGE_SPECIFIER`** . The "index" counts from the end of the list backwards. Depending on the kind of data and the message protocol, the "range" may extend toward the front of the list from the index or toward the end of the list. In other words, the index works in reverse, the range may or may not.

As with messages, the precise meaning of a given specifier depends upon the context. Additionally, there may be user−defined (or perhaps more properly object−defined) specifiers. User−defined specifier constants should be greater than **`B_SPECIFIERS_END`** to prevent conflicts with Be−defined specifiers.

Specifiers are added to the "specifier" field of a scripting message using **`BMessage::AddSpecifier()`**. There are several variants of this method, including shortcuts for adding **`B_DIRECT_SPECIFIER`**, **`B_INDEX_SPECIFIER`**, **`B_RANGE_SPECIFIER`**, and **`B_NAME_SPECIFIER`** specifiers. For all other specifiers, you must manually construct the specifier and add it to the scripting message with **`AddSpecifier()`**. For example, to add a **`B_ID_SPECIFIER`**:

```
BMessage specifier(B_ID_SPECIFIER); // create a new specifier
specifier.AddInt32("id", 2827); // add the id number to the specifier
message.AddSpecifier(&specifier); // add the specifier to the message
```

You *must* use **`AddSpecifier()`** to add specifiers to a BMessage; it performs additional scripting support work that **`AddMessage()`** doesn't.

The Specifier Stack

In general, an application will not be able to obtain a BMessenger for the target object; instead, it'll have to settle for a BMessenger targeting the BApplication of the program containing the desired object. In these cases, a single specifier may be insufficient to target a scripting message. The true power of specifiers lies in their ability to be chained together in the specifier stack.

An example best illustrates the operation of the specifier stack. The following code snippet creates a message that will target the frame of the second view of the window named "egg" in the target application:

```
message.AddSpecifier("Label");
message.AddSpecifier("MenuBar");
message.AddSpecifier("Window", 1);
```

Repeated calls to **`AddSpecifier()`** build the specifier stack. The order of the calls is very important; the specifiers are evaluated in the opposite order from which they were added. When this message is received by the target application, it will first peel off the third specifier and direct the message to the second window of the application. The BWindow will then peel off the second specifier and direct the message to the window's key menu bar. The first specifier ("Label") is then processed by the BMenuBar. This process is covered in more detail below under "ResolveSpecifier()".

Replies

A reply is generated for every scripting request. The reply message contains the following fields:

- The *what* data member defaults to **`B_REPLY`** unless some other constant is appropriate. For example, if the message was not understood, the object responds with a **`B_MESSAGE_NOT_UNDERSTOOD`** BMessage.
- The **`B_INT32_TYPE`** field "error" contains the error code for the operation. This field is always present.
- Responses to a successful **`B_GET_PROPERTY`** request will additionally contain the value or values of the requested property in the "result" array. The data will be of a type appropriate for the property.

Any scriptable objects that you create should also obey the above protocol. Of course, individual objects are free to define their own protocols for relaying additional information in the reply; in these cases, consult the documentation for the class in question.

Creating and Sending Scripting Messages

The scripting facilities of an application can be invoked in three easy steps:

- Set the command constant for the scripting message.
- Construct the specifier stack for the scripting message.
- Send the scripting message to the target application.

Example

Suppose we want to fetch the frame rectangle of the second view of the window titled "egg" in an application with the signature "application/x−fish". The code:

```
BMessage message, reply;
BRect result;

// set the command constant
message.what = B_GET_PROPERTY;

// construct the specifier stack
message.AddSpecifier("Frame"); // B_DIRECT_SPECIFIER
message.AddSpecifier("View", 1); // B_INDEX_SPECIFIER
message.AddSpecifier("Window", "egg"); // B_NAME_SPECIFIER

// send the message and fetch the result
BMessenger("application/x-fish").SendMessage(&message, &reply);
reply.FindRect("result", &result)
```

Short and sweet.

Suites

There is one missing element in the scripting system, namely the ability to query an object for its scripting abilities. This is useful when the controlling application doesn't know the precise type of the object it is scripting. Having a method of discovering the scripting abilities of an object enables more dynamic uses of scripting.

An object's scripting abilities are organized into one or more scripting "suites," a set of supported messages and associated specifiers. A suite is identified by a MIME−like string with the "suite" supertype. For example, BControl implements the "suite/vnd.Be−control" scripting suite. Nothing prevents two objects from implementing the same suite; two sound editors, for example, could have different implementations of a common scripting suite for filtering audio data.

To ask an object for its supported scripting suites, send it a standard scripting message with a **`B_GET_PROPERTY`** request for the "Suites" property:

```
message.what = B_GET_PROPERTY;
message.AddSpecifier("Suites");

... add remaining specifiers here ...

messenger.SendMessage(&message, &reply);
```

The target object responds with a **`B_REPLY`** BMessage with the following fields:

- The error code in "error".
- An array named "suites" containing the names of the suites supported by the object.
- An array named "messages" containing flattened BPropertyInfo objects describing the supported messages and specifiers for the various supported suites.

Less usefully, you can send a **`B_GET_SUPPORTED_SUITES`** BMessage directly to an object and obtain its supported suites in an identically−formed reply.

Every scriptable object supports the "suite/vnd.Be−handler" suite by dint of its BHandler heritage. This suite is sometimes referred to as the "universal suite." It performs the following functions:

- Implements the "Suites" propery and responds to **`B_GET_SUPPORTED_SUITES`** messages, as described above.
- Implements the "Messenger" property, allowing the caller to obtain a BMessenger to the object, simplifying further communication with the object.
- Implements the "InternalName" property, returning the name of the BHandler.
- Responds to any other scripting requests with a **`B_MESSAGE_NOT_UNDERSTOOD`** BMessage. This is a "catch−all" response after all the other objects in the hierarchy have rejected the scripting request.

Making Objects Scriptable

Since scripting messages are passed via BMessengers, objects accepting scripting messages must be derived from BHandler. Typically, adding scripting support entails little more than overriding the following methods:

- **`ResolveSpecifier()`** to direct the scripting message to the appropriate BHandler.
- **`MessageReceived()`** to implement the scripting requests.
- **`GetSupportedSuites()`** to publish the supported scripting suites.

ResolveSpecifier()

virtual BHandler ***`ResolveSpecifier(`**BMessage **message*, int32 *index*, BMessage **specifier*, int32 *what*, const char **property***)**

Implemented by derived classes to determine the proper handler for a scripting message. The message is targeted to the BHandler, but the specifiers may indicate that it should be assigned to another object. It's the job of **`ResolveSpecifier()`** to examine the current specifier (or more, if necessary) and return the object that should either handle the message or look at the next specifier. This function is called before the message is dispatched and before any filtering functions are called.

The first argument, *message*, points to the scripting message under consideration. The current specifier is passed in *specifier*; it will be at index *index* in the specifier array of *message*. Finally, *what* contains the **`what`** data member of *specifier* while *property* contains the name of the targetted property.

ResolveSpecifier() returns a pointer to the next BHandler that should look at the message. Here, it has four options:

- If the *specifier* identifies a BHandler belonging to another BLooper, it should send the *message* to the BLooper and return **NULL**. The message will be handled in the message loop of the other BLooper; it won't be further processed in this one. For example, a BHandler that kept a list of proxies might use code like the following:

```
if ( (strcmp(property, "Proxy") == 0)
```

Since this function resolved the specifier at *index*, it calls **PopSpecifier()** to decrement the index before forwarding the message. Otherwise, the next handler would try to resolve the same specifier.

- If the *specifier* picks out another BHandler object belonging to the same BLooper, **ResolveSpecifier()** can return that BHandler. For example:

```
if ( proxy ) {
    message->PopSpecifier();
```

This, in effect, puts the returned object in the BHandler's place as the designated handler for the message. The BLooper will give the returned handler a chance to respond to the message or resolve the next specifier.

- If it can resolve all remaining specifiers and recognizes the message as one that the BHandler itself can handle, it should return the BHandler (**this**). For example:

```
if ( (strcmp(property, "Value") == 0)
```

This confirms the BHandler as the message target. **ResolveSpecifier()** won't be called again, so it's not necessary to call **PopSpecifier()** before returning.

- If it doesn't recognize the property or can't resolve the specifier, it should call (and return the value returned by) the inherited version of **ResolveSpecifier()**.

The BApplication object takes the first path when it resolves a specifier for a "Window" property; it sends the message to the specified BWindow and returns **NULL**. A BWindow follows the second path when it resolves a specifier for a "View" property; it returns the specified BView. Thus, a message initially targeted to the BApplication object can find its way to a BView.

BHandler's version of **ResolveSpecifier()** recognizes a **B_GET_PROPERTY** *message* with a direct *specifier* requesting a "Suite" for the supported suites, "Messenger" for the BHandler, or the BHandler's "InternalName" (the same name that its **Name()** function returns). In all three cases, it assigns the BHandler (**this**) as the object responsible for the message.

For all other specifiers and messages, it sends a **B_MESSAGE_NOT_UNDERSTOOD** reply and returns **NULL**. The reply message has an "error" field with **B_SCRIPT_SYNTAX** as the error and a "message" field with a longer textual explanation of the error.

MessageReceived()

virtual status_t **MessageReceived(**BMessage **message***)**

MessageReceived() is called to process any incoming scripting messages. Scripting messages are treated in this regard much as any other BMessage. **MessageReceived()** should be implemented to carry out the actions requested by scripting commands.

GetSupportedSuites()

virtual status_t **GetSupportedSuites(**BMessage **message***)**

Implemented by derived classes to report the suites of messages and specifiers they understand. This function is called in response to either a **B_GET_PROPERTIES** scripting message for the "Suites" property or a **B_GET_SUPPORTED_SUITES** message.

Each derived class should add the names of the suites it implements to the "suites" array of *message*. Each item in the array is a MIME−like string with the "suite" supertype. In addition, the class should add corresponding flattened BPropertyInfo objects in the "messages" array. A typical implementation of **`GetSupportedSuites()`** looks like:

```
status_t MyHandler::GetSupportedSuites(BMessage *message)
{
   message−>AddString("suites", "suite/vnd.Me−my_handler"));
   BPropertyInfo prop_info(prop_list);
   message−>AddFlat("messages", &prop_info);
   return BHandler::GetSupportedSuites(message);
}
```

The value returned by **`GetSupportedSuites()`** is added to *message* in the int32 "error" field. BHandler's version of this function adds the universal suite "suite/vnd.Be−handler" to *message* then returns **`B_OK`**.

BApplication

Derived from: BLooper > BHandler > BArchivable

Declared in: be/app/Application.h

Library: libbe.so

Summary

The BApplication class defines an object that represents your application, creates a connection to the App Server, and runs your app's main message loop. An app can only create one BApplication object; the system automatically set the global **be_app** object to point to the BApplication object you create.

A BApplication object's most pervasive task is to handle messages that are sent to your app, a subject that's described in detail below. But message handling aside, you can also use your BApplication object to...

- *Control the cursor*. BApplication defines functions that hide and show the cursor, and set the cursor's image. See **SetCursor()**.
- *Access the window list*. You can iterate through the windows that your application has created with **WindowAt()**.
- *Get information about your application*. Your app's signature, executable location, and launch flags can be retrieved through **GetAppInfo()**. Additional informationicons, version strings, recognized file typescan be retrieved by creating an BAppFileInfo object based on your app's executable file. BAppFileInfo is defined in the Storage Kit.

be_app and Subclassing BApplication

Because of its importance, the BApplication object that you create is automatically assigned to the global **be_app** variable. Anytime you need to refer to your BApplication objectfrom anywhere in your codeyou can use **be_app** instead.

Unless you're creating a very simple application, you should subclass BApplication. But be aware that the **be_app** variable is typed as (BApplication *). You'll have to cast **be_app** when you call a function that's declared by your subclass:

```
((MyApp *)be_app)->MyAppFunction();
```

Constructing the Object and Running the Message Loop

As with all BLoopers, to use a BApplication you construct the object and then tell it to start its message loop by calling the **Run()** function. However, unlike other loopers, BApplication's **Run()** doesn't return until the application is told to quit. And after **Run()** returns, you delete the objectit isn't deleted for you.

Typically, you create your BApplication object in your **main()** functionit's usually the first object you create. The barest outline of a typical **main()** function looks something like this:

```
#include <Application.h>

main()1
{
   2new BApplication("application/x-vnd.your-app-sig")3;

   /* Further initialization goes here -- read settings, set globals, etc. */

   be_app->Run()4;

   /* Clean up -- write settings, etc. */

   delete be_app;
}
```

1 The **main()** function doesn't declare *argc* and *argv* parameters (used for passing along command line arguments). If the user passes command line arguments to your app, they'll show up in the **ArgvReceived()** hook function.

2 Why no pointer assignment? The constructor automatically assigns the object to **be_app**, so you don't have to assign it yourself.

3 The string passed to the constructor sets the application's signature. This is a precautionary measureit's better to add the signature as a resource than to define it here (a resource signature overrides the constructor signature). Use the **FileTypes** app to set the signature as a resource.

4 As explained in the BLooper class, **Run()** is almost always called from the same thread in which you construct the BApplication object. (More accurately, the constructor locks the object, and **Run()** unlocks it. Since locks are scoped to threads, the easiest thing to do is to construct and **Run()** in the same thread.)

Application Messages

After you tell your BApplication to run, its message loop begins to receive messages. In general, the messages are handled in the expected fashion: They show up in BApplication's **MessageReceived()** function (or that of a designated BHandler; for more on how messages are dispatched to handlers, see <x>).

But BApplication also recognizes a set of application messages that it handles by invoking corresponding hook functions. The hook functions are invoked by **DispatchMessage()** so the application messages never show up in **MessageReceived()**.

Overriding the hook functions that correspond to the application messages is an important part of the implementation of a BApplication subclass.

In the table below, the application messages (identified by their command constants) are listed in roughly the order your BApplication can expect to receive them.

B_ARGV_RECEIVED	**ArgvReceived()**	Command line arguments are delivered through this message.
B_REFS_RECEIVED	**RefsReceived()**	Files (entry_refs) that are dropped on your app's icon, or that are double−clicked to launch your app are delivered through this message.
B_READY_TO_RUN	**ReadyToRun()**	Invoked from within **Run()**, the application has finished configuring itself and is ready to go. If you haven't already created and displayed an initial window, you should do so here.
B_APP_ACTIVATED	**AppActivated()**	The application has just become the active application, or has relinquished that status.
B_PULSE	**Pulse()**	If requested, pulse messages are sent at regular intervals by the system.
B_ABOUT_REQUESTED	**AboutRequested()**	The user wants to see the app's About... box.

The protocols for the application messages are described <x>.

For more information on the details of when and why the hook functions are invoked, see the individual function descriptions.

A BApplication can also receive the **B_QUIT_REQUESTED** looper message. As explained in BLooper, **B_QUIT_REQUESTED** causes **Quit()** to be called, contingent on the value returned by the **QuitRequested()** hook function. However, BApplication's implementation of **Quit()** is different from BLooper's version. Don't miss it.

Other Topics

- *Locking*. As a BLooper, a BApplication mustt be locked before calling certain protected functions. The BApplication locking mechanism is inherited without modification from BLooper.
- *FileTypes settings*. The BApplication object represents your application at run−time. However, some of the characteristics of your appwhether it can be launched more than once, the file types it can open, its iconare not run−time decisions.

Hook Functions

AboutRequested()

AppActivated()

ArgvReceived()

Pulse()

ReadyToRun()

RefsReceived()

Constructor and Destructor

BApplication()

BApplication(const char **signature***)**

BApplication(const char **signature*, status_t **error***)**

BApplication(BMessage **archive***)**

The constructor creates a new object, locks it, sets the global variable **be_app** to point to it, and establishes a connection to the Application Server. From this point on, your app can receive messages, although it won't start processing them until you call **Run()**. You can also begin creating and displaying BWindow objectseven before you call **Run()**.

The *signature* constructors assign the argument as the app's application signature. The argument is ignored if a signature is already specified in a resource or attribute of the application's executable (serious apps should always set the signature as both an attribute and a resource). The signature is a MIME type string that must have the supertype "application". For more information on application signatures and how to set them, see <x>.

If you specify *error*, a pointer to a status_t, any error that occurs while constructing the BApplication will be returned in that variable. Alternately, you can call **InitCheck()** to check the results. If an error is returned by the constructor, you shouldn't call **Run()**.

The *archive* constructor is an implementation detail; see the BArchivable class.

~BApplication()

virtual **`~BApplication()`**

Closes and deletes the application's BWindows (and the BViews they contain), and severs the application's connection to the Application Server.

Never delete a BApplication object while it's runningwait until **`Run()`** returns. To stop a BApplication (and so cause **`Run()`** to return), send it a **`B_QUIT_REQUESTED`** message:

```
be_app->PostMessage(B_QUIT_REQUESTED);
```

Static Functions

AppResources()

`static BResources` ***`AppResources(`**void**`)`**

Returns a BResources object that's configured from your application's executable file. You may read the data in the BResources object, but you're not allowed to write it; see the BResources class for details. The BResources object belongs to the BApplication class and mustn't be freed.

You needn't have a **`be_app`** object to invoke this function.

Instantiate() see BArchivable::Instantiate()

Member Functions

AboutRequested()

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

Hook function that's invoked when the BApplication receives a **`B_ABOUT_REQUESTED`** message, undoubtedly because the user clicked an *About...* menu item. You should implement the function to put a window on−screen that provides the user with information about the application (version number, license restrictions, authors' names, etc).

AppActivated()

virtual void **`AppActivated(`**bool *active***`)`**

Hook function that's invoked when the application receives a **`B_APP_ACTIVATED`** message. The message is sent when the app gains or loses active application status. The *active* flag tells you which way the wind blows: **`true`** means your app is now active; **`false`** means it isn't.

The user can activate an app by clicking on or unhiding one of its windows; you can activate an app programmatically by calling **`BWindow::Activate()`** or **`BRoster::ActivateApp()`**. (With regard to the latter: This function is called only if the app has an "activatable" windowi.e. a non−modal, non−floating window).

During launch, this function is called after **`ReadyToRun()`** (provided the app is displaying an activatable window).

"be_app"

Archive() , see

ArgvReceived()

virtual void **ArgvReceived(**int32 *argc*, char ***argv***)**

Hook function that's invoked when the application receives a **B_ARGV_RECEIVED** message. The message is sent if command line arguments are used in launching the app from the shell, or if *argv*/*argc* values are passed to **BRoster::Launch()**.

This function isn't called if there were no command line arguments, or if **BRoster::Launch()** was called without *argv*/*argc* values.

When the app is launched from the shell, **ArgvReceived()**'s arguments are identical to the traditional **main()** arguments: The number of command line arguments is passed as *argc*; the arguments themselves are passed as an array of strings in *argv*. The first *argv* string identifes the executable file; the other strings are the command line arguments proper. For example, this...

```
$ MyApp file1 file2
```

...produces the *argv* array {"./MyApp", "file1", "file2"}.

BRoster::Launch() forwards its *argv* and *argc* arguments, but adds the executable name to the front of the *argv* array and increments the *argc* value.

Normally, the **B_ARGV_RECEIVED** message (if sent at all) is sent once, just before **B_READY_TO_RUN** is sent. However, if the user tries to re−launch (from the command line and with arguments) an already−running app that's set to **B_EXCLUSIVE_LAUNCH** or **B_SINGLE_LAUNCH**, the re−launch will generate a **B_ARGV_RECEIVED** message that's sent to the already−running image. Thus, for such apps, the **B_ARGV_RECEIVED** message can show up at any time.

CountWindows() see WindowAt()

DispatchMessage() see BLooper::DispatchMessage()

GetAppInfo()

status_t **GetAppInfo(**app_info **theInfo***)** const

Returns information about the application. This is a cover for

```
be_roster>GetRunningAppInfo(be_app->Team(), theInfo);
```

See **BRoster::GetAppInfo()** for more information.

HideCursor() see SetCursor()

IsCursorHidden() see SetCursor()

IsLaunching()

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

Returns **true** if the app is still launching. An app is considered to be in its launching phase until **ReadyToRun()** returns. Invoked from within **ReadyToRun()**, **IsLaunching()** returns **true**.

MessageReceived() see BHandler::MessageReceived()

ObscureCursor() see SetCursor()

Pulse() , SetPulseRate()

```
virtual void Pulse(void)

void SetPulseRate(bigtime_t rate)
```

`Pulse()` is a hook function that's called when the app receives a **`B_PULSE`** message. The message is sent once every *rate* microseconds, as set in **`SetPulseRate()`**. The first **`Pulse()`** message is sent after **`ReadyToRun()`** returns. If the pulse rate is 0 (the default), the **`B_PULSE`** messages aren't sent.

You can implement **`Pulse()`** to do whatever you want (the default version does nothing), but don't try to use it for precision timing: The pulse granularity is no better than 100,000 microseconds.

Keep in mind that **`Pulse()`** executes in the app's message loop thread along with all other message handling functions. Your app won't receive any **`Pulse()`** invocations while it's waiting for some other handler function (including **`MessageReceived()`**) to finish. In the meantime, **`B_PULSE`** messages will be stacking up in the message queue; when the loop becomes "unblocked", you'll see a burst of **`Pulse()`** invocations.

Quit() see Run()

QuitRequested()

```
virtual bool QuitRequested(void)
```

Hook function that's invoked when the app receives a **`B_QUIT_REQUESTED`** message. As described in the BLooper class (which declares this function), the request to quit is confirmed if **`QuitRequested()`** returns **`true`**, and denied if it returns **`false`**.

In its implementation, BApplication sends **`BWindow::QuitRequested()`** to each of its BWindow objects. If they all agree to quit, the windows are all destroyed (through **`BWindow::Quit()`**) and this **`QuitRequested()`** returns **`true`**. But if any BWindow refuses to quit, that window and all surviving windows are saved, and this **`QuitRequested()`** returns **`false`**.

Augment this function as you will, but be sure to call the BApplication version in your implementation.

ReadyToRun()

```
virtual void ReadyToRun(void)
```

Hook function that's called when the app receives a **`B_READY_TO_RUN`** message. The message is sent automatically during the **`Run()`** function, and is sent after the initial **`B_REFS_RECEIVED`** and **`B_ARGV_RECEIVED`** messages (if any) have been handled. This is the only application message that every running app is guaranteed to receive.

What you do with **`ReadyToRun()`** is up to youif your app hasn't put up a window by the time this function is called, you'll probably want to do it here. The default version of **`ReadyToRun()`** is empty.

RefsReceived()

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

Hook function that's called when the app receives a **`B_REFS_RECEIVED`** message. The message is sent when the user drops a file (or files) on your app's icon, or double clicks a file that's handled by your app. The message can arrive either at launch time, or while your app is already runninguse **`IsLaunching()`** to tell which.

message contains a single field named "refs" that contains one or more **`entry_ref`** (**`B_REF_TYPE`**) itemsone for each file that was dropped or double−clicked. Do with them what you will; the default implementation is empty. Typically, you would use the refs to create BEntry or BFile objects.

"Scripting Suites and Properties"

ResolveSpecifier() , see

Run() , Quit()

```
virtual thread_id Run(void)
```

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

These functions, inherited from BLooper, are different enough from their parent versions to warrant description:

- **`Run()`** doesn't spawn a new threadit runs the message loop in the thread that it's called from, and doesn't return until the message loop stops.
- **`Quit()`** doesn't kill the looper threadit tells the thread to finish processing the message queue (disallowing new messages) at which point **`Run()`** will be able to return. After so instructing the thread, **`Quit()`** returnsit doesn't wait for the message queue to empty.
- Also, **`Quit()`** doesn't delete the BApplication object. It's up to you to delete it after **`Run()`** returns. (However, **`Quit()`** *does* delete the object if it's called before the message loop startsi.e. before **`Run()`** is called.)

SetCursor() , HideCursor() , ShowCursor() , ObscureCursor() , IsCursorHidden()

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

void **`SetCursor(`**const BCursor **cursor*, bool *sync* = true**`)`**

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

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

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

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

Cursor functions:

- **`SetCursor()`** sets the cursor image that's used when this is the active application. You can pass one of the Be−defined cursor constants (**`B_HAND_CURSOR`** and **`B_I_BEAM_CURSOR`**) or create your own cursor image. The cursor data format is described below.
- You can also call **`SetCursor()`** passing a BCursor object; specifying *sync* as **`true`** forces the Application Server to immediately resynchronize, thereby ensuring that the cursor change takes place immediately. The default BCursors are **`B_CURSOR_SYSTEM_DEFAULT`** for the hand cursor and **`B_CURSOR_I_BEAM`** for the I−beam text editing cursor.
- **`HideCursor()`** removes the cursor from the screen.
- **`ShowCursor()`** restores it.
- **`ObscureCursor()`** hides the cursor until the user moves the mouse.
- **`IsCursorHidden()`** returns **`true`** if the cursor is hidden (but not obscured), and **`false`** if not.

The cursor data format is described in the "Cursor Data Format" section under BCursor.

ShowCursor() see SetCursor()

WindowAt() , CountWindows()

BWindow ***`WindowAt(`**int32 *index***`)`** const

int32 **`CountWindows(`**void**`)`** const

`WindowAt()` returns the *index*'th BWindow object in the application's window list. If *index* is out of range, the function returns **`NULL`**.

`CountWindows()` returns the number of windows in the window list.

- The windows list includes all windows explicitly created by the appwhether they're normal, floating, or modal, and whether or not they're actually displayedbut excludes private windows created by Be classes.
- The order of windows in the list has no signficance.
- Locking the BApplication object *doesn't* lock the window list. If you need coordinated access to the list, you'll have to provide your own locking mechanism that protects these functions and all BWindow construction and deletion.

Global Variables

be_app

```
BApplication *be_app;
```

be_app is the global variable that represents your BApplication object. You can refer to be_app anywhere you need a reference to the BApplication object that you created. If you want to call a function that's declared by your BApplication subclass, you have to cast be_app to your subclass:

```
   ((MyApp *)be_app)->MyAppFunction();
```

be_app_messenger

```
BMessenger *be_app_messenger;
```

be_app_messenger is a global BMessenger that targets your be_app object. It's created in the BApplication constructor.

Archived Fields

"mime_sig"	B_STRING_TYPE	Application signature.

Scripting Suites and Properties

"Name"

B_GET_PROPERTY	B_DIRECT_SPECIFIER	Gets the name of the application's main thread.

"Window"

B_COUNT_PROPERTIES	B_DIRECT_SPECIFIER	Returns CountWindows().
Not applicable.	B_NAME_SPECIFIER, B_INDEX_SPECIFIER, B_REVERSE_INDEX_SPECIFIER	The message is forwarded to the specified BWindow.

BClipboard

Derived from: none

Declared in: be/app/Clipboard.h

Library: libbe.so

Allocation: Constructor, on the stack, or use the

Summary

A BClipboard object is an interface to a *clipboard*, a resource that provides system−wide, temporary data storage. Clipboards are identified by name; if two apps want to refer to the same clipboard, they simply create respective BClipboard objects with the same name:

```
/* App A:  This creates a clipboard named "MyClipboard". */
BClipboard *appAclipboard = new BClipboard("MyClipboard");

/* App B:  This object refers to the clipboard already created by App A. */
BClipboard *appBclipboard = new BClipboard("MyClipboard");
```

The System Clipboard

In practice, you rarely need to construct your own BClipboard object; instead, you use the BClipboard that's created for you by your BApplication object. This object, which you refer to through the global **`be_clipboard`** variable, accesses the default system clipboard. Data that you write to your **`be_clipboard`** object can be read from any other app's **`be_clipboard`**. For example, the cut/copy/paste operations defined by BTextView transfer data through the system clipboard.

To access the system clipboard without creating a BApplication object, construct a BClipboard object with the name "system". The system clipboard is under the control of the useryou should only read or write the system clipboard as a direct result of the user's actions. If you create your own clipboards don't name them "system".

The Clipboard Message

To access a clipboard's data, you call functions on a BMessage that the BClipboard object hands you (through its **`Data()`** function). The BMessage follows these conventions:

- The **`what`** value is unused.
- The data is stored in a message field. The field should be typed as **`B_MIME_TYPE`**; the MIME type that describes the data should be used as the name of the field that holds the data (see "Writing to the Clipboard" for an example).
- If the BMessage contains more than one field, each field should present the same data in a different format. For example, the **StyledEdit** app writes text data in its own format (in order to encode the fonts, colors, etc.) and also writes the data as plain ASCII text (MIME type "text/plain").

Writing to the Clipboard

The following annotated example shows how to write to the clipboard.

```
BMessage *clip = (BMessage *)NULL;

if (be_clipboard->Lock()1) {
    be_clipboard->Clear()2;
    if ((clip = be_clipboard->Data()3) {
      clip->AddData("text/MyFormat", B_MIME_TYPE, myText, myLength)4;
      clip->AddData("text/plain", B_MIME_TYPE, asciiText, asciiLength)4;
      be_clipboard->Commit()5;
   }
    be_clipboard->Unlock()6;
}
```

1 **`Lock()`** your BClipboard object. This uploads data from the clipboard into your BClipboard's local BMessage object, and prevents other threads in your application from accessing the BClipboard's data. Note that locking does *not* lock the underlying clipboard dataother applications can change the clipboard while you have your object locked.

2 Prepare the BClipboard for writing by calling **`Clear()`**. This erases the data that was uploaded from the clipboard.

3 Call **`Data()`** to get a pointer to the BClipboard's BMessage object.

4 Write the data by invoking **`AddData()`** directly on the BMessage. In the example, we write the data in two different formats.

5 Call **`Commit()`** to copy your BMessage back to the clipboard. As soon as you call **`Commit()`**, the data that you added is visible to other clipboard clients.

6 **`Unlock()`** balances the **`Lock()`**. The BClipboard object can now be accessed by other threads in your application.

If you decide that you don't want to commit your changes, you should call **`Revert()`** before you unlock.

Reading from the Clipboard

Here we show how to read a simple string from the clipboard.

```
const char *text;
int32 textLen;
BMessage *clip = (BMessage *)NULL;

if (be_clipboard->Lock()1) {
   if ((clip = be_clipboard->Data();
      clip->FindData("text/plain", B_MIME_TYPE,
         (const void **)&text, &textlen)2;
   be_clipboard->Unlock()3;
}
```

1 As in writing, we bracket the operation with **`Lock()`** and **`Unlock()`**. Keep in mind that **`Lock()`** uploads data from the clipboard into our object. Any changes that are made to the clipboard (by some other application) after **`Lock()`** is called won't be seen here.

2 In this example, we only look for one hard−coded format. In a real application, you may have a list of formats that you can look for.

3 It isn't necessary to examine the clipboard data *before* you unlock it. The **`FindData()`** call could just as well have been performed after the **`Unlock()`** call.

Persistence

Inter−boot persistence: Clipboard data does *not* persist between bootsthe constructor provides a persistence flag, but it's currently unused.

Intra−boot persistence: Once you've created a clipboard, that clipboard will exist until you reboot your computer. For example, let's say you design an app that creates a clipboard called "MyClip": You launch the app, write something to "MyClip", and then quit the app. The clipboardand the data that you wrote to itwill still exist: If you relaunch your app (or any app that knows about "MyClip"), you can pick up the data by reading from the "MyClip" clipboard.

Constructor and Destructor

BClipboard()

`BClipboard(`const char **name*, bool *discard* = false**`)`**

Creates a new BClipboard object that refers to the *name* clipboard. The clipboard itself is created if a clipboard of that name doesn't already exist.

The *discard* flag is currently unused.

~BClipboard()

virtual **`~BClipboard()`**

Destroys the BClipboard object. The clipboard itself and the data it contains are not affected by the object's destruction.

Member Functions

Clear() , Commit() , Revert()

status_t **`Clear(`**void**`)`**

status_t **`Commit(`**void**`)`**

status_t **`Revert(`**void**`)`**

These functions are used when you're writing data to the clipboard. **`Clear()`** prepares your BClipboard for writing. You call **`Clear()`** just before you add new data to your clipboard message. **`Commit()`** copies your BClipboard data back to the clipboard. See "Writing to the Clipboard" for an example of these functions.

`Revert()` refreshes the BClipboard's data message by uploading it from the clipboard. The function is provided for the (rare) case where you alter your BClipboard's data message, and then decide to back out of the change. In this case, you should call **`Revert()`** (rather than **`Commit()`**). If you don't revert, your BClipboard's message will still contain your unwanted change, even if you unlock and then re−lock the object.

All three functions returns **`B_ERROR`** if the BClipboard isn't locked, and **`B_OK`** otherwise.

Commit() see Clear()

Data()

BMessage ***`Data(`**void**`)`** const

Returns the BMessage object that holds the BClipboard's data, or **`NULL`** if the BClipboard isn't locked. You're expected to read and write the BMessage directly; however, you may *not* free it or dispatch it like a normal BMessage. If you change the BMessage and want to write it back to the clipboard, you have to call **`Commit()`** after you make the change.

See "The Clipboard Message" for more information.

DataSource()

BMessenger **`DataSource(`**void**`)`** const

Returns a BMessenger that targets the BApplication object of the application that last committed data to the clipboard. The BClipboard needn't be locked.

LocalCount() , SystemCount()

uint32 **`LocalCount(`**void**`)`** const

uint32 **`SystemCount(`**void**`)`** const

These functions return the clipboard count. **`LocalCount()`** uses a cached count, while **`SystemCount()`** asks the Application Server for the more accurate system counter.

Lock() , Unlock() , IsLocked()

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

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

bool **`IsLocked(`**void**`)`**

`Lock()` uploads data from the clipboard into your BClipboard object, and locks the object so no other thread in your application can use it. You must call **`Lock()`** before reading or writing the BClipboard. **`Lock()`** blocks if the object is already locked. It returns **`true`** if the lock was acquired, and **`false`** if the BClipboard object was deleted while **`Lock()`** was blocked.

There's no way to tell **`Lock()`** to time out.

`Unlock()` unlocks the object so other threads in your application can use it.

`IsLocked()` hardly needs to be documented.

Name()

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

Returns the name of the clipboard. The object needn't be locked.

Revert() see Clear()

StartWatching() , StopWatching()

```
status_t StartWatching(BMessenger target)
status_t StopWatching(BMessenger target)
```

If you want to be alerted when the clipboard changes, call `StartWatching()`, passing a BMessenger to be the target for the notification. When the clipboard changes, a `B_CLIPBOARD_CHANGED` message will be sent to the target.

`StopWatching()` stops monitoring the clipboard for changes.

RETURN CODES

`B_OK`. No error.

- Other errors. You get the idea.

StopWatching() see StartWatching()

SystemCount() see LocalCount()

Unlock() see Lock()

BCursor

Derived from: public BArchivable

Declared in: be/app/Cursor.h

Library: libbe.so

Summary

You can use a BCursor to represent a mouse cursor as an object instead of as a block of pixel data; this can be more convenient in some situations. Also, if you want to call **`BApplication::SetCursor()`** without forcing an immediate sync of the Application Server, you have to use a BCursor.

The default BCursors are **`B_CURSOR_SYSTEM_DEFAULT`** for the hand cursor and **`B_CURSOR_I_BEAM`** for the I−beam text editing cursor.

Cursor Data Format

- The first four bytes of cursor data give information about the cursor:
- Byte 1: *Size in pixels−per−side*. Cursors are always square; currently, only 16−by−16 cursors are allowed.
- Byte 2: *Color depth in bits−per−pixel*. Currently, only one−bit monochrome is allowed.
- Bytes 3&4: *Hot spot coordinates*. Given in "cursor coordinates" where (0,0) is the upper left corner of the cursor grid, byte 3 is the hot spot's y coordinate, and byte 4 is its x coordinate. The hot spot is a single pixel that's "activated" when the user clicks the mouse. To push a button, for example, the hot spot must be within the button's bounds.
- Next comes an array of pixel color data. Pixels are specified from left to right in rows starting at the top of the image and working downward. The size of an array element is the depth of the image. In one−bit depth...
- the 16x16 array fits in 32 bytes.
- 1 is black and 0 is white.
- Then comes the pixel transparency bitmask, given left−to−right and top−to−bottom. 1 is opaque, 0 is transparent. Transparency only applies to white pixelsblack pixels are always opaque.

Constructor and Destructor

BCursor()

`BCursor(`const void **cursorData***`)`**

`BCursor(`BMessage **archive***`)`**

Initializes the new cursor object. If you specify a non−**`NULL`** value for *cursorData*, the cursor is initialized with the specified cursor data.

If you specify a **`NULL`** value for *cursorData*, the cursor is useless; since this class doesn't currently provide a means of setting the cursor data once the object is instantiated, you're out of luck, so why bother?

BCursor doesn't currently implement archiving, so you shouldn't use the second form.

~BCursor()

virtual **`~BCursor()`**

Releases any resources used by the cursor.

Static Functions

Instantiate()

static BArchivable ***Instantiate(**BMessage **archive***)**

Not currently implemented; always returns **`NULL`**.

See also: **`BArchivable::Instantiate()`**, **`instantiate_object()`**, **`Archive()`**

BHandler

Derived from: BArchivable

Declared in: be/app/Handler.h

Library: libbe.so

Summary

A BHandler object responds to messages that are handed to it by a BLooper. The BLooper tells the BHandler about a message by invoking the BHandler's MessageReceived() function.

The Handler List

To be eligible to get messages from a BLooper, a BHandler must be in the BLooper's list of *eligible handlers* (as explained in the BLooper class). The list of eligible handlers is ordered; if the "first" handler doesn't want to respond to a message that it has received, it simply calls the inherited version of **`MessageReceived()`** and the message will automatically be handed to the object's "next" handler. (System messages are not handed down the list.) The BLooper that all these BHandlers belong to is always the last the last handler in the list (BLooper inherits from BHandler).

A BHandler's next handler assignment can be changed through **`SetNextHandler()`**.

Targets

You can designate a target BHandler for most messages. The designation is made when calling BLooper's **`PostMessage()`** function or when constructing the BMessenger object that will send the message. 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.

Filtering

Messages can be filtered before they're dispatchedthat is, you can define a function that will look at the message before the target BHandler's hook function is called. The filter function is associated with a BMessageFilter object, which records the criteria for calling the function.

Filters that should apply only to messages targeted to a particular BHandler are assigned to the BHandler by **`SetFilterList()`** or **`AddFilter()`**. Filters that might apply to any message a BLooper dispatches, regardless of its target, are assigned by the parallel BLooper functions, **`SetCommonFilterList()`** and **`AddCommonFilter()`**. See those functions and the BMessageFilter class for details.

Notifiers and Observers

A BHandler can be a **notifier**. A notifier is a handler that maintains one or more states and notifies interested parties when those states change. Each state is idenfified by a 32−bit "what" code. Interested parties, called **observers**, can register to monitor changes in one or more states by calling **`StartWatching()`** and specifying the "what" code of the state they want to be notified of changes to.

This notification occurs when the BHandler calls **`SendNotices()`**; it's the handler's job to call **`SendNotices()`** whenever a state changes, to ensure that observers are kept informed of the changes. The BHandler passes to **`SendNotices()`** a message template to be sent to the observers.

When a notification is sent, observers receive a **`B_OBSERVER_NOTICE_CHANGE`** message with an int32 field **`B_OBSERVE_WHICH_CHANGE`** that contains the "what" code of the state that changed, and a **`B_OBSERVE_ORIGINAL_WHAT`** field that contains the "what" value that was in the template BMessage.

Hook Functions

`MessageReceived()`

Constructor and Destructor

BHandler()

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

BHandler(BMessage *archive)
```

Initializes the BHandler by assigning it a *name* and registering it with the messaging system. BHandlers can also be reconstructed from a BMessage *archive*.

~BHandler()

```
virtual ~BHandler()
```

Deletes any BMessageFilters assigned to the BHandler.

Static Functions

Instantiate() see BArchivable::Instantiate()

Member Functions

AddFilter() see SetFilterList()

"Archived Fields"

Archive() , see

FilterList() see SetFilterList()

GetSupportedSuites()

```
virtual status_t GetSupportedSuites(BMessage *message)
```

Implemented by derived classes to report the suites of messages and specifiers they understand. This function is called in response to either a **`B_GET_PROPERTIES`** scripting message for the "Suites" property or a **`B_GET_SUPPORTED_SUITES`** message.

Each derived class should add the names of the suites it implements to the "suites" array of *message*. Each item in the array is a MIME string with the "suite" supertype. In addition, the class should add corresponding flattened BPropertyInfo objects in the "messages" array. A typical implementation of **`GetSupportedSuites()`** looks like:

```
status_t MyHandler::GetSupportedSuites(BMessage *message)
{
   message->AddString("suites", "suite/vnd.Me-my_handler"));
   BPropertyInfo prop_info(prop_list);
   message->AddFlat("messages", &prop_info);
   return BHandler::GetSupportedSuites(message);
}
```

The value returned by **`GetSupportedSuites()`** is added to *message* in the int32 "error" field.

BHandler's version of this function adds the universal suite "suite/vnd.Be−handler" to *message* then returns **`B_OK`**.

LockLooper() , LockLooperWithTimeout() , UnlockLooper()

```
bool LockLooper(void)

status_t LockLooperWithTimeout(bigtime_t timeout)

void UnlockLooper(void)
```

These are "smart" versions of BLooper's locking functions (**`BLooper::Lock()`** et. al.). The difference between the versions is that these functions retrieve the handler's looper and lock it (or unlock it) in a pseudo−atomic operation, thus avoiding a race condition. Anytime you're tempted to write code such as this:

```
/* DON'T DO THIS */
if (myHandler->Looper()->Lock()) {
   ...
   myHandler->Looper()->Unlock();
}
```

Don't do it. Instead, do this:

```
/* DO THIS INSTEAD */
if (myHandler->LockLooper()) {
   ...
   myHandler->UnlockLooper();
}
```

Except for an additional return value in **`LockLooperWithTimeout()`**, these functions are identical to their BLooper analogues. See to **`BLooper::Lock()`** for details.

RETURN CODES

`LockLooper()` returns **`true`** if it was able to lock the looper, or if it's already locked by the calling thread, and **`false`** otherwise. If the handler changes loopers during the call, **`false`** is returned.

`LockLooperWithTimeout()` returns:

- **`B_OK`**. The looper was successfully locked.
- **`B_TIMED_OUT`**. The call timed out without locking the looper.
- **`B_BAD_VALUE`**. This handler's looper is invalid.
- **`B_MISMATCHED_VALUES`**. The handler switched loopers during the call.

LockLooperWithTimeout() see LockLooper()

Looper()

`BLooper *Looper(void) const`

Returns the BLooper object that the BHandler has been added to. The function returns **`NULL`** if the object hasn't been added to a BLooper. A BHandler can be associated with only one BLooper at a time.

Note that a BLooper object automatically adds itself (as a handler) to itself (as a looper), and a BWindow automatically adds its child views. To explicitly add a handler to a looper, you call **`BLooper::AddHandler()`**.

MessageReceived()

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

Implemented by derived classes to respond to messages that are received by the BHandler. The default (BHandler) implementation of this function responds only to scripting requests. It passes all other messages to the next handler by calling that object's version of **`MessageReceived()`**.

A typical **`MessageReceived()`** implementation distinguishes between messages by looking at its command constant (i.e. the **`what`** field). For example:

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

It's essential that all unhandled messages are passed to the base class implementation of **`MessageReceived()`**, as shown here. The handler chain model depends on it.

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

Do not delete the argument *message* ***when you're done with.*** It doesn't belong to you.

Name() see SetName()

NextHandler() see SetNextHandler()

ResolveSpecifier()

virtual BHandler ***ResolveSpecifier(**BMessage **message*, int32 *index*,
    BMessage **specifier*, int32 *what*, const char **property***)**

Implemented by derived classes to determine the proper handler for a scripting message. The message is targeted to the BHandler, but the specifiers may indicate that it should be assigned to another object. It's the job of **`ResolveSpecifier()`** to examine the current specifier (or more, if necessary) and return the object that should either handle the message or look at the next specifier. This function is called before the message is dispatched and before any filtering functions are called.

The first argument, *message*, points to the scripting message under consideration. The current specifier is passed in *specifier*; it will be at index *index* in the specifier array of *message*. Finally, *what* contains the **`what`** data member of *specifier* while *property* contains the name of the targeted property.

If the current BHandler is able to handle the scripting message, it should return a pointer to itself (*this*). If a BHandler in another BLooper is the target, it should send the message to the BLooper and return **`NULL`**. This causes the current BLooper to stop further processing of the message. Otherwise, the function should return a pointer to the BHandler that should handle the message, if no specifiers remain, or look at the next specifier, if any exist. Often, **`ResolveSpecifier()`** calls **`PopSpecifier()`** before returning so the next BHandler won't examine the same specifier.

BHandler's version of **`ResolveSpecifier()`** recognizes a **`B_GET_PROPERTY`** *message* with a direct *specifier* requesting a "Messenger" for the BHandler or the BHandler's "InternalName" (the same name that its **`Name()`** function returns). In both cases, it assigns the BHandler (**`this`**) as the object responsible for the message.

For all other specifiers and messages, it sends a **`B_MESSAGE_NOT_UNDERSTOOD`** reply and returns **`NULL`**. The reply message has an "error" field with **`B_SCRIPT_SYNTAX`** as the error and a "message" field with a longer textual explanation of the error.

For more information about this function and scripting in general, see the "Scripting" section near the beginning of this chapter.

See also: **`BMessage::AddSpecifier()`**, **`BMessage::GetCurrentSpecifier()`**

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* of filters; the list must contain pointers to instances of the BMessageFilter class or to instances of classes that derive from BMessageFilter. The new list replaces any list of filters previously assigned. All objects in the previous list are deleted, as is the BList that contains them. If *list* is **`NULL`**, the current list is removed without a replacement. **`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 without deleting it. 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
```

`SetNextHandler()` reorders the objects in the *handler* chain so that handler follows this BHandler. This BHandler and *handler* must already be part of the same chain, and the BLooper they belong to must be locked. The order of objects in the handler chain affects the way in−coming messages are handled (as explained in "Inheritance and the Handler Chain". By default handlers are placed in the order that they're added (through **`BLooper::AddHandler()`**).

`NextHandler()` returns this object next handler. If this object is at the end of the chain, it returns **`NULL`**.

StartWatching() , StartWatchingAll() , StopWatching() , StopWatchingAll()

```
status_t StartWatching(BMessenger watcher, uint32 what)

status_t StartWatching(BHandler *watcher, uint32 what)

status_t StartWatchingAll(BMessenger watcher)

status_t StartWatchingAll(BHandller *watcher)

status_t StopWatching(BMessenger watcher, uint32 what)

status_t StopWatching(BHandler *watcher, uint32 what)

status_t StopWatchingAll(BMessenger watcher)

status_t StopWatchingAll(BHandller *watcher)
```

The BHandler class provides the concept of a **notifier**. Notifiers maintain one or more states that other entities might want to monitor changes to. These states are identified by a 32−bit *what* code. Another entitya BHandler or a BMessengercan watch for changes notifiers' states. These are called **observers**.

`StartWatching()` registers the BMessenger or BHandler specified by *watcher* to be notified whenever the state specified by *what* changes. **`StartWatchingAll()`** registers the specified BMessenger or BHandler to be notified when any of the notifer's states change.

`StartWatching()` works by sending a message to the BHandler you want to observe, with a BMessenger back to the observer, so both must be attached to a looper at the time **`StartWatching()`** is called.

`StopWatching()` ceases monitoring of the state *what*. **`StopWatchingAll()`**, by some odd coincidence, stops all monitoring by the BHandler or BMessenger specified by *watcher*.

RETURN CODES

- B_OK. No error.
- **`B_BAD_HANDLER`**. The specified BHandler isn't valid.

UnlockLooper() see LockLooper()

Archived Fields

"_name"	**`B_STRING_TYPE`**	The object's name (see **`SetName()`**).

Scripting Suites and Properties

"InternalName"

B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_STRING_TYPE

Returns the handler's name.

"Messenger"

B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_MESSENGER_TYPE

Returns a BMessenger for the handler.

"Suites"

B_GET_PROPERTY	B_DIRECT_SPECIFIER	B_STRING_TYPE array

Returns an array of suites that the target supports, identified by name (e.g. "suite/vnd.Be−handler").

BInvoker

Derived from: none

Declared in: be/app/Invoker.h

Library: libbe.so

Summary

BInvoker is a convenience class that bundles up everything you need to create a handy message−sending package. The BInvoker contains: **(a)** a BMessage, **(b)** a BMessenger (that identifies a target handler), and **(c)** an optional BHandler that handles replies. You set these ingredients, invoke **`Invoke()`**, and off goes the message to the target. Replies are sent to the reply handler (**`be_app`** by default).

BInvoker uses **`BMessenger::SendMessage()`** to send its messages. The invocation is asynchronous, and there's no time limit on the reply.

BInvoker is mostly used as a mix−in class. A number of classes in the Interface Kitnotably BControlderive from BInvoker.

Constructor and Destructor

BInvoker()

`BInvoker(`BMessage **message*, BMessenger *messenger***`)`**

`BInvoker(`BMessage **message*, const BHandler **handler*, const BLooper **looper* = NULL**`)`**

`BInvoker(`void**`)`**

Initializes the BInvoker by setting its message and its messenger.

- The object's BMessage is taken directly as *message*the object is *not* copied. The BInvoker takes over ownership of the BMessage that you pass in.
- The object's BMessenger is copied from *messenger*, or initialized with *looper* and *handler*. See the BMessenger class for details on how a BMessenger identifies a target.

If you want a reply handler, you have to call **`SetHandlerForReply()`** after the constructor returns. You can reset the message and messenger through **`SetMessage()`** and **`SetTarget()`**.

~BInvoker()

virtual **`~BInvoker()`**

Deletes the object's BMessage.

Member Functions

BeginInvokeNotify() , EndInvokeNotify()

void **`BeginInvokeNotify(`**uint32 *kind* = B_CONTROL_INVOKED**`)`**

void **`EndInvokeNotify()`**

If for some reason you need to implement a method that emulates an **`InvokeNotify()`** call inside an **`Invoke()`** implementation, you should wrap the invocation code in these functions. They set up and tear down an **`InvokeNotify()`** context.

Command() see SetMessage()

HandlerForReply() see SetHandlerForReply()

Invoke() , InvokeNotify()

virtual status_t **Invoke(**BMessage **message* = NULL**)**

status_t **InvokeNotify(**BMessage **message*, uint32 *kind* = B_CONTROL_INVOKED**)**

Invoke() tells the BInvoker's messenger to send a message. If *message* is non−**NULL**, that message is sent, otherwise the object sends its default message (i.e. the BMessage that was passed in the constructor or in **SetMessage()**). The message is sent asynchronously with no time limit on the reply.

Regarding the use of the default message *vs* the argument, a common practice is to reserve the default message as a template, and pass a fine−tuned copy to **Invoke()**:

```
/* Add the current system time to a copy of the default message. */
BMessage copy(invoker.Message());
copy.AddInt64("when", system_time());
invoker.Invoke(&copy);
```

The **InvokeNotify()** function sends the *message* to the target, using the notification change code specified by *kind*. If message is **NULL**, nothing gets sent to the target, but any watchers of the invoker's handler will receive their expected notifications. By default, the *kind* is **B_CONTROL_INVOKED**the same *kind* sent by a straight **Invoke()**.

In general, you should call **InvokeNotify()** instead of **Invoke()** in new BeOS applications that run under BeOS 5 and later. You can map old code to new like this:

Invoke()	InvokeNotify(Message())
Invoke(Message())	InvokeNotify(Message())
Invoke(ModificationMessage())	InvokeNotify(ModificationMessage(), B_CONTROL_MODIFIED)

Invoke() doesn't call **SendNotices()** by default; you'll have to implement code to do it yourself. Here's how:

```
status_t BControl::Invoke(BMessage *msg) {
   bool notify = false;
   uint32 kind = InvokeKind(&notify);

   BMessage clone(kind);
   status_t err = B_BAD_VALUE;

   if (!msg && !notify) {
      // If no message is supplied, pull it from the BInvoker.
      // However, ONLY do so if this is not an InvokeNotify()
      // context -- otherwise, this is not the default invocation
      // message, so we don>t want it to get in the way here.
      // For example, a control may call InvokeNotify() with their
      // "modification" message...  if that message isn>t set,
      // we still want to send notification to any watchers, but
      // -don>t- want to send a message through the invoker.
      msg = Message();
   }
   if (!msg) {
      // If not being watched, there is nothing to do.
      if( !IsWatched() ) return err;
   } else {
      clone = *msg;
   }

   clone.AddInt64("when", system_time());
   clone.AddPointer("source", this);
   clone.AddInt32("be:value",fValue);
   clone.AddMessenger(B_NOTIFICATION_SENDER, BMessenger(this));
   if( msg ) err = BInvoker::Invoke(&clone);

   // Also send invocation to any observers of this handler.
   SendNotices(kind, &clone);

   return err;
}
```

RETURN CODES

- **`B_OK`**. The message was sent.
- **`B_BAD_VALUE`**. No default message, and no *message* argument.
- *Other errors forwarded from* **`BMessenger::SendMessage()`**.

InvokeKind()

uint32 **`InvokeKind(`**bool **notify* = NULL**`)`**

Returns the kind passed to **`InvokeNotify()`**. This should be called from within your implementation of **`Invoke()`** if you need to determine what kind was specified when **`InvokeNotify()`** was called. If you care whether **`Invoke()`** or **`InvokeNotify()`** was originally called, you can specify a pointer to a bool, *notify*, which is set to **`true`** if **`InvokeNotify()`** was called, or **`false`** if **`Invoke()`** was called.

This lets you fetch the **`InvokeNotify()`** arguments from your **`Invoke()`** code without breaking compatibility with older applications by adding arguments to **`Invoke()`**.

InvokeNotify() see Invoke()

IsTargetLocal() see SetTarget()

Message() see SetMessage()

Messenger() see SetTarget()

SetHandlerForReply() , HandlerForReply()

virtual status_t **`SetHandlerForReply(`**BHandler **replyHandler***`)`**

BHandler ***`HandlerForReply(`**void**`)`** const

`SetHandlerForReply()` sets the BHandler object that handles replies that are sent back by the target. By default (or if *replyHandler* is **`NULL`**), replies are sent to the BApplication object.

`HandlerForReply()` returns the object set through **`SetHandlerForReply()`**. If the reply handler isn't set, this function returns **`NULL`**, it *doesn't* return **`be_app`** (even though **`be_app`** will be handling the reply).

RETURN CODES

- **`SetHandlerForReply()`** always returns **`B_OK`**it doesn't check for validity.

SetMessage() , Message() , Command()

virtual status_t **`SetMessage(`**BMessage **message***`)`**

BMessage ***`Message(`**void**`)`** const

uint32 **`Command(`**void**`)`** const

`SetMessage()` sets the BInvoker's default message to point to *message* (the message is *not* copied). The previous default message (if any) is deleted; a **`NULL`** *message* deletes the previous message without setting a new one. The BInvoker owns the BMessage that you pass in; you mustn't **`delete`** it yourself.

`Message()` returns a pointer to the default message, and **`Command()`** returns its **`what`** data member. Lacking a default message, the functions return **`NULL`**.

RETURN CODES

- **`SetMessage()`** always returns **`B_OK`**.

SetTarget() , Target() , IsTargetLocal() , Messenger()

```
virtual status_t SetTarget(BMessenger messenger)

virtual status_t SetTarget(const BHandler *handler, const BLooper *looper = NULL)

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

bool IsTargetLocal(void) const

BMessenger Messenger(void) const
```

These functions set and query the BInvoker's target. This is the BHandler to which the object sends a message when **`Invoke()`** is called. The target is represented by a BMessenger object; you can set the BMessenger as a copy of *messenger*, or initialize it with *looper* and *handler*. See the BMessenger class for details on how a BMessenger identifies a target.

`Target()` returns the BHandler that's targeted by the object's messenger. If *looper* is non−**`NULL`**, the BLooper that owns the BHandler is returned by reference. If the target was set as a looper's preferred handler (i.e. **`SetTarget(NULL, looper)`**), or if the target hasn't been set yet, **`Target()`** returns **`NULL`**. The function returns **`NULL`** for both objects if the target is remote.

`IsTargetLocal()` returns **`true`** if the target lives within the BInvoker's application, and **`false`** if it belongs to some other app.

`Messenger()` returns a copy of the BMessenger object the BInvoker uses to send messages. If a target hasn't been set yet, the return will be invalid.

RETURN CODES

- **`B_OK`**. The target was successfully set.
- **`B_BAD_VALUE`**. The proposed *handler* doesn't belong to a BLooper.
- **`B_MISMATCHED_VALUES`**. *handler* doesn't belong to *looper*.

`SetTarget()` doesn't detect invalid BLoopers and BMessengers.

SetTimeout() , Timeout()

```
status_t SetTimeout(bigtime_t timeout)

bigtime_t Timeout(void) const
```

`SetTimeout()` sets the timeout that will be used when sending the invocation message to the invoker's target. By default this is **`B_INFINITE_TIMEOUT`**.

`Timeout()` returns the current setting for this value.

RETURN CODES

- B_OK. No error.

Target() see SetTarget()

Timeout() see SetTimeout()

BLooper

Derived from: public BHandler

Declared in: be/app/Looper.h

Library: libbe.so

Summary

A BLooper object creates a "message loop" that receives messages that are sent or posted to the BLooper. The message loop runs in a separate thread that's spawned (and told to run) when the BLooper receives a **`Run()`** call. If you're creating your own BLooper, you can invoke **`Run()`** from within the constructor.

You tell the loop to stop by sending the BLooper a **`B_QUIT_REQUESTED`** message, which invokes the object's **`Quit()`** function. You can also call **`Quit()`** directly, but you have to Lock() the object first (BLooper locking is explained later). **`Quit()`** deletes the BLooper for you.

The BApplication class, the most important BLooper subclass, bends the above description in a couple of ways:

- A BApplication takes over the main thread, it doesn't spawn a new one.
- You *do* have to **`delete be_app`**; you can't just **`Quit()`** it.

Messages and Handlers

You can deliver messages to a BLooper's thread by...

- Posting them directly by calling BLooper's **`PostMessage()`** function.
- Sending them through BMessenger's **`SendMessage()`** or BMessage's **`SendReply()`** function.

As messages arrive, they're added to the BLooper's BMessageQueue object. The BLooper takes messages from the queue in the order that they arrived, and calls **`DispatchMessage()`** for each one. **`DispatchMessage()`** locks the BLooper and then hands the message to a BHandler object by invoking the handler's **`MessageReceived()`** function. But which BHandler does the BLooper hand the message to? Here's the path:

- If an incoming message targets a specific BHandler, and if that BHandler is one of the BLooper's *eligible handlers* (as set through the **`AddHandler()`** function), the BLooper uses that BHandler. (See the BMessage and BMessenger classes for instructions on how to target a BHandler.)
- Otherwise it hands the message to its *preferred handler*, as set through **`SetPreferredHandler()`**.
- If no preferred handler is set, the BLooper itself handles the message (its own implementation of **`MessageReceived()`** is invoked).

After the handler is finished (when its MessageReceived() returns), the BMessage is automatically deleted and the BLooper is unlocked.

Locking

Access to many BLooper functions (and some BHandler functions) is proceed by a lock. To invoke a lock−protected functions (or groups of functions), you must first call **`Lock()`**, and then call **`Unlock()`** when you're done. The lock is scoped to the calling thread: **`Lock()`**/**`Unlock()`** calls can be nested within the thread. Keep in mind that each **`Lock()`** *must* balanced by an **`Unlock()`**.

The BLooper constructor automatically locks the object. It's unlocked when **`Run()`** is invoked. This means that the **`Run()`** functionand any other lock−protected functions that you call before you call **`Run()`**must be called from the thread that constructed the BLooper.

Allocation

Because they **`delete`** themselves when told to quit, BLoopers can't be allocated on the stack; you have to construct them with **`new`**.

Hook Functions

- **`DispatchMessage()`**
- **`QuitRequested()`**

Constructor and Destructor

BLooper()

BLooper(const char *name = NULL,
int32 *priority* = B_NORMAL_PRIORITY,
int32 *portCapacity* = B_LOOPER_PORT_DEFAULT_CAPACITY**)**

BLooper(BMessage **archive***)**

Assigns the BLooper object a *name* and then locks it (by calling **Lock()**). *priority* is a value that describes the amount of CPU attention the message loop will receive once it starts running, and *portCapacity* is the number of messages the BLooper can hold in its "message port" (this is *not* the message queue, as explained below).

After you construct the BLooper, you have to tell it to **Run()**. Because the object is locked, **Run()** can only be called from the thread that constructed the object. It's legal to invoke **Run()** from within a subclass implementation of the constructor.

Priority

A set of priority values are defined in **kernel/OS.h**; from lowest to highest, they are:

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 controlling real−time processes that need unfettered access to the CPUs.

Port Capacity

Messages that are sent to a BLooper first show up in a port (as the term is defined by the Kernel Kit), and then are moved to the BMessageQueue. The capacity of the BMessageQueue is virtually unlimited; the capacity of the port is not. Although messages are moved from the port to the queue as quickly as possible, the port can fill up. A full port will block subsequent message senders.

The default port capacity (100), should be sufficient for most apps, but you can fiddle with it through the *portCapacity* argument.

~BLooper()

virtual **~BLooper()**

Frees the message queue and all pending messages and deletes the message loop. BHandlers that have been added to the BLooper are not deleted, but BMessageFilter objects added as common filters are

In general, you should never **delete** your BLooper objects: With the exception of the BApplication object, BLoopers are destroyed by the **Quit()** function.

If you create a BLooper−derived class that uses multiple inheritance, make sure the first class your mixin class inherits from is BLooper; otherwise, you'll crash when you try to close the window. This happens because of an interaction between the window thread how C++ deletes objects of a multiply−inherited class. In other words:

```
class myClass : public BLooper, public OtherClass {
...
};
```

is safe, while

```
class myClass : public OtherClass, public BLooper {
...
};
```

is not.

Static Functions

LooperForThread()

```
static BLooper *LooperForThread(thread_id thread)
```

Returns the BLooper object that spawned the specified *thread*, or **`NULL`** if the thread doesn't belong to a BLooper.

Member Functions

AddCommonFilterList() , RemoveCommonFilterList() , SetCommonFilterList() , CommonFilterList()

```
virtual void AddCommonFilter(BMessageFilter *filter)
virtual bool RemoveCommonFilter(BMessageFilter *filter)
virtual void SetCommonFilterList(BList *filters)
BList *CommonFilterList(void) const
```

For all but **`CommonFilterList()`**, the BLooper must be locked.

These functions manage the BLooper's list of BMessageFilters. Message filters are objects that screen in−coming messages. In the case of BLooper, each message is passed through all filters in the list before it's passed on to **`DispatchMessage()`**. The order of the filters in the list is determinate. See the BMessageFilter class for details on how message filters work.

`AddCommonFilter()` adds *filter* to the end of the filter list (creating a BList container if necessary).

`RemoveCommonFilter()` removes *filter* from the list, but doesn't free the filter. It returns **`true`** if successful, and **`false`** if it can't find the specified filter.

`SetCommonFilterList()` deletes the current filter list and its contents, and replaces it with *filters*. All elements in *filters* must be BMessageFilter pointers. The BLooper takes ownership of all objects in *filters*, as well as *filters* itself. If *filters* is **`NULL`**, the current list is deleted without a replacement.

`CommonFilterList()` returns a pointer to the current list. You can examine the list but you shouldn't modify or delete it.

AddHandler() , RemoveHandler() , HandlerAt() , CountHandlers() , IndexOf()

```
void AddHandler(BHandler *handler)
bool RemoveHandler(BHandler *handler)
BHandler *HandlerAt(int32 index) const
int32 CountHandlers(void) const
int32 IndexOf(BHandler *handler) const
```

`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.

`AddHandler()` fails if the *handler* already belongs to a BLooper; a BHandler can belong to no more than one BLooper at a time. It can change its affiliation from time to time, but must be removed from one BLooper before it can be added to another. **`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.

`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**.

HandlerAt() returns the BHandler object currently located at *index* in the BLooper's list of eligible handlers, or **NULL** if the index is out of range. Indices begin at 0 and there are no gaps in the list. **CountHandlers()** returns the number of objects currently in the list; the count should always be at least 1, since the list automatically includes the BLooper itself. **IndexOf()** returns the index of the specified *handler*, or **B_ERROR** if that object isn't in the list.

For any of these functions to work, the BLooper must be locked.

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

Archive() see BArchivable::Archive()

CommonFilterList() see AddCommonFilterList()

CountHandlers() see AddHandler()

CountLockRequests() see LockingThread()

CountLocks() see LockingThread()

CurrentMessage() , DetachCurrentMessage()

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

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

The message that a BLooper passes to its handler(s) is called the "current message." These functions access the current message; they're meaningless (they return **NULL**) when called from outside the message processing loop.

CurrentMessage() simply returns a pointer to the current message without affecting the BMessage object itself. This is particularly useful to functions that respond to system messages (such as **MouseDown()** and **ScreenChanged()**), but that aren't sent the full BMessage object that initiated the response.

DetachCurrentMessage() removes the current message from the message queue and passes ownership of it to the caller; deleting the message is the caller's responsibility. This is useful if you want to delay the response to the message without tying up the BLooper. But be carefulif the message sender is waiting for a synchronous reply, detaching the message and holding on to it will block the sender.

DetachCurrentMessage() see CurrentMessage()

DispatchMessage()

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

DispatchMessage() is the BLooper's central message−processing function. It's called automatically as messages arrive in the looper's queue, one invocation per message. You never invoke **DispatchMessage()** yourself.

The default implementation passes *message* to *handler* by invoking the latter's **MessageReceived()**:

```
target->MessageReceived(message);
```

The only exception is where *message*.**what** is **B_QUIT_REQUESTED** and *handler* is the looper itself; in this case, the object invokes its own **QuitRequested()** function.

You can override this function to dispatch the messages that your own application defines or recognizes. All unhandled messages should be passed to the base class version, as demonstrated below:

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

```
            break;
         }
   }
```

Also, note that you mustn't delete *message*; it's deleted for you..

The system locks the BLooper before calling **DispatchMessage()** and keeps it locked for the duration of the function.

HandlerAt() see AddHandler()

IndexOf() see AddHandler()

IsLocked() see LockingThread()

Lock() , LockWithTimeout() , Unlock()

```
bool Lock(void)

status_t LockWithTimeout(bigtime_t timeout)

void Unlock(void)
```

Lock() locks the BLooper. Locks are held within the context of a thread; while a BLooper is locked, no other thread can invoke its most important functions (**AddHandler()**, **DispatchMessage()**, etc.)

If the looper is already locked (by some other thread), **Lock()** blocks until the looper is unlocked. To set a timeout for the block, use **LockWithTimeout()** instead. *timeout* is measured in microseconds; if it's 0, the function returns immediately (with or without the lock); if it's **B_INFINITE_TIMEOUT**, it blocks without limit.

Unlock() unlocks a locked looper. It can only be called by the thread that currently holds the lock.

Calls to **Lock()**/**LockWithTimeout()** and **Unlock()** can be nested, but locking and unlocking must always be balanced. A single **Unlock()** will *not* undo a series of **Lock()**'s.

RETURN CODES

Lock() returns **true** if it was able to lock the looper, or if it's already locked by the calling thread, and **false** otherwise.

LockWithTimeout() returns:

- **B_OK**. The looper was successfully locked.
- **B_TIMED_OUT**. The call timed out without locking the looper.
- **B_BAD_VALUE**. This looper was deleted while the function was blocked.

LockingThread() , IsLocked() , CountLocks() , CountLockRequests() , Sem()

```
thread_id LockingThread(void) const

bool IsLocked(void) const

int32 CountLocks(void) const

int32 CountLockRequests(void) const

sem_id Sem(void) const
```

These functions may be useful while debugging a BLooper.

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

IsLocked() returns **true** if the calling thread currently has the BLooper locked (if it's the locking thread) and **false** if not (if some other thread is the locking thread or the BLooper isn't locked).

CountLocks() returns the number of times the locking thread has locked the BLooperthe number of **Lock()** (or **LockWithTimeout()**) calls that have not yet been balanced by matching **Unlock()** calls.

CountLockRequests() returns the number of threads currently trying to lock the BLooper. The count includes the thread that currently has the lock plus all threads currently waiting to acquire it.

`Sem()` returns the `sem_id` for the semaphore that the BLooper uses to implement the locking mechanism.

See also: `Lock()`

LockWithTimeout() see Lock()

MessageReceived()

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

Simply calls the inherited function. For the current release, the BLooper implementation of this function does nothing of importance.

See also: `BHandler::MessageReceived()`

MessageQueue()

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

Returns the queue that holds messages delivered 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()

```
status_t PostMessage(BMessage *message)

status_t PostMessage(uint32 command)

status_t PostMessage(BMessage *message,
        BHandler *handler,
        BHandler *replyHandler = NULL)

status_t PostMessage(uint32 command,
        BHandler *handler,
        BHandler *replyHandler = NULL)
```

`PostMessage()` is similar to `BMessenger::SendMessage()`. The BMessenger version is preferred (it's a bit safer than `PostMessage()`).

Places a message at the far end of the BLooper's message queue. The message will be processed by `DispatchMessage()` when it comes to the head of the queue.

The message can be a full BMessage object (*message*), or just a command constant (*command*). In the former case, the message is copied and the caller retains ownership of the argument, which can be deleted as soon as `PostMessage()` returns. In the latter case, a BMessage is created (and deleted) for you.

handler is the designated handler for the message, and must be part of this BLooper's handler chain. If *handler* is (literally) **`NULL`**, the designated handler is the BLooper's preferred handler at the time `DispatchMessage()` is called. In the versions of `PostMessage()` that don't have a *handler* argument, the designated handler is the BLooper object itself.

Replies to the message are delivered to *replyHandler*. If a *replyHandler* isn't specified, replies are sent to `be_app_messenger`.

A BLooper should never post a message to itself from within its own message loop thread.

RETURN CODES

- B_OK. The message was successfully posted.
- **B_MISMATCHED_VALUES**. *handler* doesn't belong to this BLooper.
- *Other errors*. See the return values for **BMessenger::SendMessage()**.

PreferredHandler() see SetPreferredHandler()

Quit()

```
virtual void Quit(void)
```

Shuts down the message loop (if it's running), and deletes the BLooper. The object must be locked.

When **Quit()** is called from the BLooper's thread, the message loop is immediately stopped and any messages in the message queue are deleted (without being processed). Note that, in this case, **Quit()** doesn't return since the calling thread is dead.

When called from another thread, **Quit()** waits until all messages currently in the queue have been handled before it kills the message loop. It returns after the BLooper has been deleted.

QuitRequested()

```
virtual bool QuitRequested(void)
```

Hook function that's invoked when the BLooper receives a **B_QUIT_REQUESTED** message. You never invoke this function directly. Derived classes implement this function to return **true** if it's okay to quit this BLooper, and **false** if not. Note that this function does *not* actually quit the objectthe code that handles the **B_QUIT_REQUESTED** message does that.

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

RemoveCommonFilter() see AddCommonFilterList()

Run()

```
virtual thread_id Run(void)
```

Spawns the message loop thread and starts it running. **Run()** expects the BLooper to be locked (once only!) when it's called; it unlocks the object before it returns. Keep in mind that a BLooper is locked when it's constructed.

Calling **Run()** on a BLooper that's already running will dump you into the debugger.

RETURN CODES

- *Positive values*. The thread was successfully spawned and started; this is the thread_id for the thread.
- *Thread errors*. See **spawn_thread()** and **resume_thread()**.
- *Port errors*. See **create_port()**.

AddCommonFilter()

SetPreferredHandler() , PreferredHandler()

```
void SetPreferredHandler(BHandlerAddCommonFilter() *handler) const

BHandler *PreferredHandler(void)
```

These functions set and return the BLooper's preferred handlerthe BHandler object that should handle messages not specifically targetted to another BHandler.

To designate the current preferred handlerwhatever object that may beas the target of a message, pass **NULL** for the target handler to **PostMessage()** or to the BMessenger constructor.

Posting or sending messages to the preferred handler can be useful. For example, in the Interface Kit, BWindow objects name the current focus view as the preferred handler. This makes it possible for other objectssuch as BMenuItems and BButtonsto target messages to the BView that's currently in focus, without knowing what view that might be. 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.

By default, BLoopers don't have a preferred handler; until one is set, **PreferredHandler()** returns **NULL**. Note however, that messages targeted to the preferred handler are dispatched to the BLooper whenever the preferred handler is **NULL**. In other words, the BLooper acts as default preferred handler, even though the default is formally **NULL**.

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

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()** always returns the application's **team_id**.

Unlock() see Lock()

Constants

B_LOOPER_PORT_DEFAULT_CAPACITY

```
#define B_LOOPER_PORT_DEFAULT_CAPACITY 100
```

The default capacity of the port that holds incoming messages before they're placed in the BLooper's BMessageQueue. The capacity is set in the BLooper constructor.

BMessage

Derived from: none

Declared in: be/app/Message.h

Library: libbe.so

Allocation: new, static, or automatic

Summary

A BMessage is a bundle of structured information. Every BMessage contains a *command constant* and some number of *data fields*.

- The command constant is an **`int32`** value that describes, roughly, the purpose of the BMessage. It's stored as the public **`what`** data member. You always set and examine the **`what`** value directlyyou don't need to call a function. (As a convenience, you can set the command constant when you create your BMessage object.)
- The data fields are name−type−value triplets. A field is be primarily identified by name, but you can look for fields by name, type, or a combination of the two. The type is encoded as a constant (**`B_INT32_TYPE`**, **`B_STRING_TYPE`**, etc), and is meant to describe the type of value that the field holds. A single field can have only one name and one type, but can contain an array of values. Individual values in a field are accessible by index.

Neither the command constant nor the data fields are mandatory. You can create a BMessage that has data but no command, or that *only* has a command. However, creating a BMessage that has neither is pointless.

Preparatory Reading

BMessages are used throughout the kits to send data (or notifications) to another threadpossibly in another application. To understand how BMessages fit into the messaging system, see "Messaging".

The BMessage class also contributes a number of functions that help define the scripting system. See "Scripting" for an introduction to this system.

BMessages are also used by a number of classes (BClipboard, BArchivable, and others) for their ability to store data.

Types of Functions

The BMessage class defines five types of functions:

- *Data field functions*. These functions either set or retrieve the value of a data field. See **`AddData(), FindData())`**, **`ReplaceData()`**, and **`RemoveName()`** .
- *Info functions*. These functions retrieve information about the state and contents of the BMessage. See **`IsSystem()`** and **`GetInfo()`**.
- *Messaging functions*. These functions are part of the messaging system. .A smaller set of functions reports on the status of a received message. For example, **`IsSourceWaiting()`** tells whether the message sender is waiting for a reply, **`WasDropped()`** says whether it was dragged and dropped, and **`DropPoint()`** says where it was dropped. **`SendReply()`**
- *Scripting functions*, such as **`AddSpecifier()`** and **`PopSpecifier()`**.
- *Flattening functions*. The data in a BMessage can be flattened. See **`Flatten()`**.

BMessage Ownership

The documentation for the functions that accept or pass back a BMessage object should tell you who's responsible for deleting the object. Most functions that accept a BMessage argument copy the object, leaving the caller with the responsibility for deleting the argument. The exceptionsi.e. BMessage−accepting functions that take over ownership of the objectare listed below:

Functions that return a BMessage to you usually don't give up ownership; in general, you don't delete the BMessages that are passed to you. The exceptionsfunctions that expect the caller to take over ownership of a passed−back BMessageare listed below:

Data Members

uint32 **`what`**
A coded constant that captures what the message is about.

Constructor and Destructor

BMessage()

`BMessage(`uint32 *command***`)`**

```
BMessage(const BMessage &message)

BMessage(void)
```

Creates a new BMessage object that has the given *command* constant, or that's a copy of another BMessage. If it's a copy, the new object contains the same command constant and data fields as *message*.

See also: `BLooper::DetachCurrentMessage()`

~BMessage()

```
virtual ~BMessage()
```

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.

The system retains ownership of the messages it delivers to you. Each message loop routinely deletes delivered BMessages after the application is finished responding to them.

Member Functions

AddData() , AddBool() , AddInt8() , AddInt16() , AddInt32() , AddInt64() , AddFloat() , AddDouble() , AddString() , AddPoint() , AddRect() , AddRef() , AddMessage() , AddMessenger() , AddPointer() , AddFlat()

```
status_t AddData(const char *name, type_code type,
    const void *data,
    ssize_t numBytes,
    bool fixedSize = true,
    int32 numItems = 1)

status_t AddBool(const char *name, bool aBool)

status_t AddInt8(const char *name, int8 anInt8)

status_t AddInt16(const char *name, int16 anInt16)

status_t AddInt32(const char *name, int32 anInt32)

status_t AddInt64(const char *name, int64 anInt64)

status_t AddFloat(const char *name, float aFloat)

status_t AddDouble(const char *name, double aDouble)

status_t AddString(const char *name, const char *string)

status_t AddString(const char *name, const BString &string)

status_t AddPoint(const char *name, BPoint point)

status_t AddRect(const char *name, BRect rect)

status_t AddRef(const char *name, const entry_ref *ref)

status_t AddMessage(const char *name, const BMessage *message)

status_t AddMessenger(const char *name, BMessenger messenger)

status_t AddPointer(const char *name, const void *pointer)

status_t AddFlat(const char *name, BFlattenable *object, int32 numItems = 1)
```

These functions add data to the field named *name* and assign a data type to the field. Field names can be no longer than 255 characters. If more than one item of data is added under the same name, the BMessage creates an array of data for that name. Each time you add another value (to the same name), the value is added to the end of the arrayyou can't add a value at a specific index. A given field can only store one type of data.

`AddData()` copies *numBytes* of *data* into the field, and assigns the data a *type* code. It 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 assigned *type* must be a specific data type; it should not be **`B_ANY_TYPE`**..

When you call **`AddData()`** to place the first item in an array under a new name, you can provide it with two arguments, *fixedSize* and *numItems*, that will improve the object's efficiency. If the *fixedSize* flag is **`true`**, each item in the array must have the same number of bytes; if the flag is **`false`**, items can vary in size. *numItems* tells the object to pre−allocate storage for some number of items. This isn't a limityou can add more than *numItems* to the field.

Most of the other functions are variants of **`AddData()`** that hard−code the field's type. For example, **`AddFloat()`** assigns the type **`B_FLOAT_TYPE`**; **`AddBool()`** assigns **`B_BOOL_TYPE`**, and so on.

`AddString()`, like **`AddData()`**, takes a pointer to the data it adds, or you can use a BString object. The *string* must be null−terminated; the null character is counted and copied into the message. Similarly, **`AddRef()`** adds the pointed to **`entry_ref`** structure to the message (and the variable−length name that's one of the elements of the structure); **`AddMessage()`** adds one BMessage to another.

The other functions are simply passed the data directly. For example, **`AddInt32()`** takes an **`int32`** or **`uint32`** and **`AddMessenger()`** takes a BMessenger object, whereas **`AddData()`** would be passed a pointer to an **`int32`** and a pointer to a BMessenger. **`AddPointer()`** adds only the pointer it's passed, not the data it points to. To accomplish the same thing, **`AddData()`** would take a pointer to the pointer. (The pointer will be valid only locally; it won't be useful to a remote destination.)

`AddFlat()` flattens an *object* (by calling its **`Flatten()`** function) and adds the flat data to the message. It calls the object's **`TypeCode()`** function to learn the type code it should associate with the data. Objects that are added through AddFlat() must inherit from BFlattenable (defined in the Support Kit).

You can also provide a *numItems* hint to **`AddFlat()`** when you call it to set up a new array. **`AddFlat()`** calls the object's **`IsFixedSize()`** function to discover whether all items in the array will be the same size.

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, **`B_NO_MEMORY`** if the BMessage can't get enough memory to hold the data, and **`B_BAD_VALUE`** if the proposed *name* for the data is longer than 255 bytes. If all goes well, they return **`B_OK`**.

There's no limit on the number of named fields a message can contain or on the size of a field's data. 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 can slow the delivery mechanism. It's sometimes better to put some of the information in a common location (a file, a private clipboard, a shared area of memory) and just refer to it in the message.

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

AddSpecifier()

status_t **`AddSpecifier(`**const BMessage **message***`)`**

status_t **`AddSpecifier(`**const char **property***`)`**

status_t **`AddSpecifier(`**const char **property*, int32 *index***`)`**

status_t **`AddSpecifier(`**const char **property*, int32 *index*, int32 *range***`)`**

status_t **`AddSpecifier(`**const char **property*, const char **name***`)`**

Adds a specifier to the specifier stack. There are several variations of this method. The first adds the specifier *message* to the specifier stack. The other methods add a specifier targeting the property *property*, with specifier constants **`B_DIRECT_SPECIFIER`**, **`B_INDEX_SPECIFIER`**, **`B_RANGE_SPECIFIER`**, and **`B_NAME_SPECIFIER`**, respectively. For all other specifiers, you must construct the specifier separately and then call **`AddSpecifier()`** on the message. For more information about specifiers, see the "Scripting" section near the beginning of this chapter.

Specifiers are stored in a data array named "specifiers." However, since **`AddSpecifier()`** also sets the notion of the current specifier, specifiers should always be added to a scripting message with this method rather than with **`AddMessage()`**.

`AddSpecifier()` returns **`B_OK`** if it's able to add the specifier to the BMessage and an error code, generally only **`B_NO_MEMORY`** to indicate that it has run out of memory, if not.

See also: **`GetCurrentSpecifier()`**, **`HasSpecifiers()`**, **`PopSpecifier()`**

CountNames()

int32 **`CountNames(`**type_code *type***`)`** const

Returns the number of named data fields in the BMessage that store data of the specified *type*. An array of information held under a single name counts as one field; 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 fields. If *type* is a specific type, it counts only fields that store data registered as that type.

See also: GetInfo()

DropPoint() see WasDropped()

FindData() , FindBool() , FintInt8() , FindInt16() , FindInt32() , FindInt64() , FindFloat() , FindDouble() , FindString() , FindPoint() , FindRect() , FindRef() , FindMessage() , FindMessenger() , FindPointer() , FindFlat()

```
status_t FindData(const char *name,
      type_code type,
      int32 index,
      const void **data,
      ssize_t *numBytes) const

status_t FindData(const char *name,
      type_code type,
      const void **data,
      ssize_t *numBytes) const

status_t FindBool(const char *name,
      int32 index,
      bool *aBool) const

status_t FindBool(const char *name, bool *aBool) const

status_t FindInt8(const char *name,
      int32 index,
      int8 *anInt8) const

status_t FindInt8(const char *name,
      int8 *anInt8) const

status_t FindInt16(const char *name,
      int32 index,
      int16 *anInt16) const

status_t FindInt16(const char *name, int16 *anInt16) const

status_t FindInt32(const char *name,
      int32 index,
      int32 *anInt32) const

status_t FindInt32(const char *name, int32 *anInt32) const

status_t FindInt64(const char *name,
      int32 index,
      int64 *anInt64) const

status_t FindInt64(const char *name, int64 *anInt64) const

status_t FindFloat(const char *name,
      int32 index,
      float *aFloat) const

status_t FindFloat(const char *name, float *aFloat) const

status_t FindDouble(const char *name,
      int32 index,
      double *aDouble) const

status_t FindDouble(const char *name, double *aDouble) const

status_t FindString(const char *name,
      int32 index,
      const char **string) const

status_t FindString(const char *name, const char **string) const

status_t FindString(const char *name, BString *string) const
```

status_t **FindString(**const char **name*, int32 *index*, BString **string* **)** const

status_t **FindPoint(**const char **name*,
int32 *index*,
BPoint **point* **)** const

status_t **FindPoint(**const char **name*, BPoint **point* **)** const

status_t **FindRect(**const char **name*,
int32 *index*,
BRect **rect* **)** const

status_t **FindRect(**const char **name*, BRect **rect* **)** const

status_t **FindRef(**const char **name*,
int32 *index*,
entry_ref **ref* **)** const

status_t **FindRef(**const char **name*, entry_ref **ref* **)** const

status_t **FindMessage(**const char **name*,
int32 *index*,
BMessage **message* **)** const

status_t **FindMessage(**const char **name*, BMessage **message* **)** const

status_t **FindMessenger(**const char **name*,
int32 *index*,
BMessenger **messenger* **)** const

status_t **FindMessenger(**const char **name*, BMessenger **messenger* **)** const

status_t **FindPointer(**const char **name*,
int32 *index*,
void ***pointer* **)** const

status_t **FindPointer(**const char **name*, void ***pointer* **)** const

status_t **FindFlat(**const char **name*,
int32 *index*,
BFlattenable **object* **)** const

status_t **FindFlat(**const char **name*, BFlattenable **object* **)** const

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.

In all cases except **FindData()** and **FindString()**, the data that's retrieved from the BMessage is copied into the reference argument; the caller is responsible for freeing the copied data). For **FindData()** and the non−BString version of **FindString()**, a pointer to the data is returned; the BMessage retains ownership of the actual data and will delete the data when the object itself is deleted.

FindData() places, in **data*, a pointer to the requested data item. The size of the item in bytes is written to *numBytes*. If *type* is **B_ANY_TYPE**, it provides a pointer to the data no matter what type it actually is. But if *type* is a specific data type, it provides the pointer only if the *name* field holds data of that particular type.

The other functions are specialized versions of **FindData()**. They match the corresponding **Add...()** functions and search for named data of a particular type, as described below:

FindBool()	a **bool**	**B_BOOL_TYPE**
FindInt8()	an **int8** or **uint8**	**B_INT8_TYPE**
FindInt16()	an **int16** or **uint16**	**B_INT16_TYPE**
FindInt32()	an **int32** or **uint32**	**B_INT32_TYPE**

FindInt64()	an int64 or uint64	B_INT64_TYPE
FindFloat()	a float	B_FLOAT_TYPE
FindDouble)	a double	B_DOUBLE_TYPE
FindString()	a character string	B_STRING_TYPE
FindPoint()	a BPoint object	B_POINT_TYPE
FindRect()	a BRect object	B_RECT_TYPE
FindRef()	an entry_ref	B_REF_TYPE
FindMessage()	a BMessage object	B_MESSAGE_TYPE
FindMessenger()	a BMessenger object	B_MESSENGER_TYPE
FindPointer()	a pointer to anything	B_POINTER_TYPE

The other type−specific functions retrieve the requested data item from the message by copying it to the variable referred to by the last argument; you get the data, not just a pointer to it. For example, **FindMessenger()** assigns the BMessenger it finds in the message to the *messenger* object, whereas **FindData()** would provide only a pointer to a BMessenger. **FindPointer()** puts the found pointer in the **void*** variable that *pointer* refers to; **FindData()**, as illustrated above, would provide a pointer to the pointer. (If the message was delivered from a remote source, pointers retrieved from the message won't be valid.)

FindRef() retrieves an **entry_ref** structure; the data that's used to reconstitute the structure may have been added as an **entry_ref** (through **AddRef()**), or as a flattened BPath object (**AddFlat()**).

FindFlat() assigns the object stored in the BMessage to the *object* passed as an argumentit calls the *object*'s **Unflatten()** function and passes it the flat data from the messageprovided that the two objects have compatible types. The argument *object*'s **AllowsTypeCode()** function must return **true** when tested with the type code stored in the message; if not, **FindFlat()** fails and returns **B_BAD_VALUE**.

If these functions can't find any data associated with *name*, they return a **B_NAME_NOT_FOUND** error. If they can't find *name* data of the requested *type* (or the type the function returns), they return **B_BAD_TYPE**. If the *index* is out of range, they return **B_BAD_INDEX**. You can rely on the values they retrieve only if they return **B_OK** and the data was correctly recorded when it was added to the message.

When they fail, **FindData()** and **FindString()** provide **NULL** pointers. **FindRect()** hands you an invalid rectangle and **FindMessenger()** an invalid BMessenger. Most of the other functions set the data values to 0, which may be indistinguishable from valid values.

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

(Several functions, such as **FindRect()** and **FindInt32()**, have versions that return the found value directly. These versions don't report errors and may not be supported in the future.)

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

Flatten() , Unflatten() , FlattenedSize()

status_t **Flatten(**BDataIO **object*, ssize_t **numBytes* = NULL**)** const

status_t **Flatten(**char **address*, ssize_t *numBytes* = NULL**)** const

status_t **Unflatten(**BDataIO **object***)**

status_t **Unflatten(**const char **address***)**

ssize_t **FlattenedSize(**void**)** const

These functions write the BMessage and the data it contains to a "flat" (untyped) buffer of bytes, and reconstruct a BMessage object from such a buffer.

If passed a BDataIO *object* (including a BFile), **Flatten()** calls the object's **Write()** function to write the message data. If passed the *address* of a buffer, it begins writing at the start of the buffer. **FlattenedSize()** returns the number of bytes you must provide in the buffer to hold the flattened object. **Flatten()** places the number of bytes actually written in the variable that its *numBytes* argument refers to.

Unflatten() empties the BMessage of any information it may happen to contain, then initializes the object from data read from the buffer. If passed a BDataIO *object*, it calls the object's **Read()** function to read the message data. If passed a buffer *address*, it begins reading at the start of the buffer. It's up to the caller to make sure that **Unflatten()** reads data that **Flatten()** wrote and that pointers are positioned correctly.

`Flatten()` returns any errors encountered when writing the data, or **`B_OK`** if there is no error.

If it doesn't recognize the data in the buffer as being a flattened object or there's a failure in reading the data, **`Unflatten()`** returns **`B_BAD_VALUE`**. If it doesn't have adequate memory to recreate the whole message, it returns **`B_NO_MEMORY`**. Otherwise, it returns **`B_OK`**.

See also: the BDataIO class in the Support Kit

GetCurrentSpecifier() , PopSpecifier()

```
status_t GetCurrentSpecifier(int32 *index,
      BMessage *specifier = NULL,
      int32 *what = NULL,
      const char **property = NULL) const

status_t PopSpecifier(void)
```

`GetCurrentSpecifier()` unpacks the current specifier in the BMessage, the one at the top of the specifier stack; **`PopSpecifier()`** changes the notion of which specifier is current, by popping the current one from the stack.

These functions aid in implementing a class−specific version of BHandler's **`ResolveSpecifier()`** functionthe first gets the specifier that needs to be resolved, and the second pops it from the stack after it is resolved. You can also call them to examine relevant specifiers when handling a message that targets an object property (such as **`B_GET_PROPERTY`**).

A scripting BMessage keeps specifiers in a data array named "specifiers"; each specifier is itself a BMessage, but one with a special structure and purpose in the scripting system. See the "Scripting" section near the beginning of this chapter for an overview of the system and the place of specifiers in it.

The specifiers in a message are ordered and, until **`PopSpecifier()`** is called, the one that was added lastthe one with the greatest indexis the current specifier. **`PopSpecifier()`** merely decrements the index that picks the current specifier; it doesn't delete anything from the BMessage.

`GetCurrentSpecifier()` puts the index of the current specifier in the variable that its first argument, *index*, refers to. If other arguments are provided, it makes the *specifier* BMessage a copy of the current specifier. It also extracts two pieces of information from the *specifier*: It places the **`what`** data member of the specifier in the *what* variable and a pointer to the property name in the *property* variable. These last two output arguments won't be valid if the *specifier* argument is **`NULL`**.

Both functions fail if the BMessage doesn't contain specifiers. In addition, **`GetCurrentSpecifier()`** fails if it can't find data in the BMessage for its *specifier* and *property* arguments, and **`PopSpecifier()`** fails if the BMessage isn't one that has been delivered to you after being processed through a message loop. When it fails, **`GetCurrentSpecifier()`** returns **`B_BAD_SCRIPT_SYNTAX`**, but **`PopSpecifier()`** returns **`B_BAD_VALUE`**. On success, both functions return **`B_OK`**.

See also: **`AddSpecifier()`**, **`HasSpecifiers()`**, **`BHandler::ResolveSpecifier()`**

GetInfo()

```
status_t GetInfo(const char *name,
      type_code *typeFound,
      int32 *countFound = NULL) const

status_t GetInfo(const char *name,
      type_code *typeFound,
      bool *fixedSize) const

status_t GetInfo(type_code type, int32 index,
      char **nameFound,
      type_code *typeFound,
      int32 *countFound = NULL) const
```

Provides information about the data fields 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 **`B_OK`**. If it can't find a *name* field within the BMessage, it sets the *countFound* variable to 0, and returns **`B_NAME_NOT_FOUND`** (without modifying the *typeFound* variable).

When the *fixedSize* argument is specified, the bool referenced by *fixedSize* is set to **`true`** if all items in the array specified by name must be the same size, and **`false`** if the items can be of different sizes (see **`AddData()`**).

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

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

- It places a pointer to the name of the data field in the variable referred to by *nameFound*.
- It puts the code for the type of data the field 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 field in the variable referred to by *countFound*.

If **GetInfo()** can't find data of the requested *type* at *index*, it sets the *countFound* variable to 0, and returns **B_BAD_TYPE**. If the index is out of range, it returns **B_BAD_INDEX**.

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

```
char   *name;
uint32  type;
int32   count;

for ( int32 i = 0;
      msg−>GetInfo(B_ANY_TYPE, i, &name, &type, &count) == B_OK;
      i++ ) {
    . . .
}
```

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

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

HasSpecifiers()

bool **HasSpecifiers(**void**)** const

Returns **true** if the BMessage has specifiers added by an **AddSpecifier()** function, and **false** if not.

See also: **AddSpecifier()**, **GetCurrentSpecifier()**

IsEmpty() see MakeEmpty()

IsReply() see WasDelivered()

IsSourceRemote() see WasDelivered()

IsSourceWaiting() see WasDelivered()

IsSystem()

bool **IsSystem(**void**)** const

Returns **true** if the **what** data member of the BMessage object identifies it as a system−defined message, and **false** if not.

MakeEmpty() , IsEmpty()

status_t **MakeEmpty(**void**)**

bool **IsEmpty(**void**)** const

MakeEmpty() removes and frees all data that has been added to the BMessage, without altering the **what** constant. It returns **B_OK**, unless the message can't be altered (as it can't if it's being dragged), in which case it returns **B_ERROR**.

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

See also: **RemoveName()**

Previous() see WasDelivered()

PrintToStream()

```
void PrintToStream(void) const
```

Prints information about the BMessage to the standard output stream (**`stdout`**). Each field 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() , RemoveData()

```
status_t RemoveName(const char *name)

status_t RemoveData(const char *name, int32 index = 0)
```

`RemoveName()` removes all data entered in the BMessage under *name* and the name itself. **`RemoveData()`** removes the single item of data at *index* in the *name* array. If the array has just one data item, it removes the array and name just as **`RemoveName()`** would.

Both functions free the memory that was allocated to hold the data, and return **`B_OK`** when successful. However, if there's no data in the BMessage under *name*, they return a **`B_NAME_NOT_FOUND`** error. If message data can be read but can't be changed (as it can't for a message that's being dragged), they both return **`B_ERROR`**. If the *index* is out of range, **`RemoveData()`** returns **`B_BAD_INDEX`** (the index is too high) or **`B_BAD_VALUE`** (the value passed is a negative number).

See also: `MakeEmpty()`

ReplaceData() , ReplaceBool() , ReplaceInt8() , ReplaceInt16() , ReplaceInt32() , ReplaceInt64() , ReplaceFloat() , ReplaceDouble() , ReplaceString() , ReplacePoint() , ReplaceRect() , ReplaceRef() , ReplaceMessage() , ReplaceMessenger() , ReplacePointer() , ReplaceFlat()

```
status_t ReplaceData(const char *name,
      type_code type,
      const void *data,
      ssize_t numBytes)

status_t ReplaceData(const char *name,
      type_code type,
       int32 index,
      const void *data,
      ssize_t numBytes)

status_t ReplaceBool(const char *name, bool aBool)

status_t ReplaceBool(const char *name,
      int32 index,
      bool aBool)

status_t ReplaceInt8(const char *name, int8 anInt8)

status_t ReplaceInt8(const char *name,
      int32 index,
      int8 anInt8)

status_t ReplaceInt16(const char *name, int16 anInt16)

status_t ReplaceInt16(const char *name,
      int32 index,
      int16 anInt16)

status_t ReplaceInt32(const char *name, long anInt32)

status_t ReplaceInt32(const char *name,
      int32 index,
      int32 anInt32)

status_t ReplaceInt64(const char *name, int64 anInt64)

status_t ReplaceInt64(const char *name,
      int32 index,
      int64 anInt64)

status_t ReplaceFloat(const char *name, float aFloat)

status_t ReplaceFloat(const char *name,
      int32 index,
      float aFloat)

status_t ReplaceDouble(const char *name, double aDouble)

status_t ReplaceDouble(const char *name,
      int32 index,
      double aDouble)

status_t ReplaceString(const char *name, const char *string)

status_t ReplaceString(const char *name,
      int32 index,
      const char *string)

status_t FindString(const char *name, BString &string)

status_t FindString(const char *name, int32 index, BString &string)

status_t ReplacePoint(const char *name, BPoint point)

status_t ReplacePoint(const char *name,
      int32 index,
            BPoint point)

status_t ReplaceRect(const char *name, BRect rect)

status_t ReplaceRect(const char *name,
      int32 index,
            BRect rect)

status_t ReplaceRef(const char *name, entry_ref *ref)

status_t ReplaceRef(const char *name,
      int32 index,
      entry_ref *ref)

status_t ReplaceMessage(const char *name, BMessage *message)

status_t ReplaceMessage(const char *name,
      int32 index,
      BMessage *message)

status_t ReplaceMessenger(const char *name, BMessenger messenger)

status_t ReplaceMessenger(const char *name,
      int32 index,
            BMessenger  messenger)

status_t ReplacePointer(const char *name, const void *pointer)

status_t ReplacePointer(const char *name,
      int32 index,
      const void *pointer)

status_t ReplaceFlat(const char *name, BFlattenable *object)

status_t ReplaceFlat(const char *name,
      int32 index,
            BFlattenable *object)
```

These functions replace a data item in the *name* field 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* field 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 field. The *type* must be specific; it can't be **`B_ANY_TYPE`**.

`FindFlat()` replaces a flattened object with another *object*, provided that the type reported by the argument *object* (by its **`TypeCode()`** function) matches the type recorded for the item in the message. If not, it returns **`B_BAD_VALUE`**.

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* field, and fail if it does not. The new data is added precisely as the counterpart **Add...()** function would add it.

If successful, all these functions return **`B_OK`**. If unsuccessful, they return an error code**`B_ERROR`** if the message is read−only (as it is while the message is being dragged), **`B_BAD_INDEX`** if the *index* is out of range, **`B_NAME_NOT_FOUND`** if the *name* field doesn't exist, or **`B_BAD_TYPE`** if the field doesn't contain data of the specified type.

See also: **`AddData()`**

ReturnAddress()

BMessenger **`ReturnAddress(`**void**`)`**

Returns a BMessenger object 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 delivered), the returned BMessenger will be invalid.

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

See also: **`SendReply()`**, **`WasDelivered()`**

SendReply()

```
status_t SendReply(BMessage *message,
     BMessage *reply,
     bigtime_t sendTimeout = B_INFINITE_TIMEOUT,
     bigtime_t replyTimeout = B_INFINITE_TIMEOUT)

status_t SendReply(BMessage *message,
         BHandler *replyHandler = NULL,
     bigtime_t sendTimeout = B_INFINITE_TIMEOUT)

status_t SendReply(uint32 command, BMessage *reply)

status_t SendReply(uint32 command, BHandler *replyHandler = 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 BMessage that's sending the reply was itself sent:

- The reply is delivered synchronously if the message sender is waiting for one to arrive. The function that sent the BMessage doesn't return until it receives the reply (or a timeout expires). 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. If a reply is sent after the sender gave up waiting for it to arrive, the reply *message* disappears into the bowels of the system.
- 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 BLooper for any replies that might be sent, then returns immediately after putting the BMessage in the pipeline. Posted messages and messages that are dragged and dropped are also eligible for asynchronous replies.

`SendReply()` works only for BMessage objects that have been processed through a message loop and delivered to you. The caller retains ownership of the reply *message* passed to **`SendReply()`**; it can be deleted (or left to die on the stack) after the function returns.

`SendReply()` sends a messagea reply message, to be sure, but a message nonetheless. 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 *replyHandler* 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.

By default, **`SendReply()`** doesn't return until the reply message is delivered (placed in the BLooper's port queue). It's possible, in some circumstances, for the receiving port queue to be full, in which case **`SendReply()`** will block until a slot becomes free. However, you can limit how long **`SendReply()`** will wait to deliver the message before it gives up and returns. The *sendTimeout* argument is the number of microseconds you give the function to do its work. If the time limit is exceeded, the function fails and returns an error (**`B_TIMED_OUT`**).

When asking for a synchronous reply, separate *sendTimeout* and *replyTimeout* limits can be set for sending the message and receiving the reply. There is no time limit if a timeout value is set to **`B_INFINITE_TIMEOUT`**as it is by default. The function won't block at all if the timeout is set to 0.

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. The *command* versions of this function have infinite timeouts; they block until the message is delivered and, if requested, a synchronous reply is received.

This function returns **`B_OK`** if the reply is successfully sent. If there's a problem in sending the message, it returns the same sort of error code as BMessenger's **`SendMessage()`**. It may also report a reply−specific problem. The more informative return values are as follows:

`B_BAD_REPLY`	Attempting to reply to a message that hasn't been delivered yet.
`B_DUPLICATE_REPLY`	Sending a reply after one has already been sent and delivered.
`B_BAD_THREAD_ID`	Sending a reply to a destination thread that no longer exists.
`B_BAD_PORT_ID`	Sending a reply to a BLooper and port that no longer exist.
`B_TIMED_OUT`	Taking longer than the specified time limit to deliver a reply message or to receive a synchronous reply to the 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.

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

Unflatten() see Flatten()

WasDelivered() , IsSourceRemote() , IsSourceWaiting() , IsReply() , Previous()

bool **`WasDelivered(`**void**`)`** const

bool **`IsSourceRemote(`**void**`)`** const

bool **`IsSourceWaiting(`**void**`)`** const

bool **`IsReply(`**void**`)`** const

const BMessage ***`Previous(`**void**`)`** const

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

`WasDelivered()` indicates whether it's possible to send a reply to a message. It returns **`true`** for a BMessage that was posted, sent, or droppedthat is, one that has been processed through a message loopand **`false`** for a message that has not yet been delivered by any means.

`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.

`IsSourceWaiting()` returns **`true`** if the message source is waiting for a synchronous reply, and **`false`** if not. The source thread 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.

`Previous()` returns the previous messagethe message to which the current BMessage is a reply. It works only for a BMessage that's received as an asynchronous reply to a previous message. A synchronous reply is received in the context of the previous message, so it's not necessary to call a function to get it. But when an asynchronous reply is received, the context of the original message is lost; this function can provide it. **`Previous()`** returns **`NULL`** if the BMessage isn't an asynchronous reply to another message.

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

WasDropped() , DropPoint()

bool **`WasDropped(`**void**`)`** const

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

`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()`

Operators

= (assignment)

```
BMessage &operator =(const BMessage&)
```

Assigns one BMessage object to another. After the assignment, the two objects are duplicates of each other without shared data.

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: none

Declared in: be/app/MessageFilter.h

Library: libbe.so

Allocation: Constructor only

Summary

A BMessageFilter is a message−screening function that you "attach" to a BLooper or BHandler. The message filter sees messages just before they're dispatched (i.e. just before **`BLooper::DispatchMessage()`**), and can modify or reject the message, change the message's designated handler, or whatever else it wants to dothe implementation of the filter function isn't restricted.

To define a message filter, you have to provide a message−filtering function. You do this by implementing the **`Filter()`** hook function in a BMessageFilter subclass, or by supplying a **`filter_hook`** function to the BMessageFilter constructor. Only one filter function per object is called. If you implement **`Filter()`** *and* provide a **`filter_hook`** function, the **`filter_hook`** will win.

To attach a message filter to a looper, call **`BLooper::AddCommonFilter()`**. To add it to a handler, call **`BHandler::AddFilter()`**. Looper filters see all incoming messages; handler filters see only those messages that are targetted for that particular handler.

A BLooper or BHandler can have more than one message filter. Furthermore, a looper can have two sets of filters: a looper set and a handler set (keep in mind that BLooper is derived from BHandler). Looper filters are applied before handler filters.

A BMessageFilter object can be assigned to only one BHandler or BLooper at a time.

The BMessageFilter class is intended to be used as part of the system−defined messaging system. If you try to use one outside this system, your results may not be what you expect.

Hook Functions

`Filter()`

Constructor and Destructor

BMessageFilter()

```
BMessageFilter(message_delivery delivery,
        message_source source,
        uint32  command,
        filter_hook filter = NULL)

BMessageFilter(message_delivery delivery,
        message_source source,
        filter_hook filter = NULL)

BMessageFilter(uint32  command,
        filter_hook filter = NULL)

BMessageFilter(const BMessageFilter &object)

BMessageFilter(const BMessageFilter *object)
```

Creates and returns a new BMessageFilter. The first three arguments define the types of messages that the object wants to see:

- *delivery* specifies how the message must arrive: drag−and−drop (**`B_DROPPED_DELIVERY`**), programmatically (**`B_PROGRAMMED_DELIVERY`**), or either (**`B_ANY_DELIVERY`**). The default is **`B_ANY_DELIVERY`**.
- *source* specifes whether the sender of the message must be local vis−a−vis this app (**`B_LOCAL_SOURCE`**), remote (**`B_REMOTE_SOURCE`**), or either (**`B_ANY_SOURCE`**). The default is **`B_ANY_SOURCE`**.
- *command* is a command constant. If supplied, the **`what`** value of the incoming message must match this value.

Messages that don't fit the definition won't be sent to the object's filter function.

The *filter* argument is a pointer to a **filter_hook** function. This is the function that's invoked when a message needs to be examined (see **filter_hook** for the protocol). You don't have to supply a **filter_hook** function; instead, you can implement BMessageFilter's **Filter()** function in a subclass.

For more information, refer to the description of the member **Filter()** function.

~BMessageFilter()

```
virtual ~BMessageFilter()
```

Does nothing.

Member Functions

Command() , FiltersAnyCommand()

```
uint32 Command(void) const

bool FiltersAnyCommand(void) const
```

Command() returns the command constant (the BMessage **what** value) that an arriving message must match for the filter to apply. **FiltersAnyCommand()** returns **true** if the filter applies to all messages, and **false** if it's limited to a specific command.

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

Filter()

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

Implemented by derived classes to examine an arriving message just before it's dispatched. The first two arguments are the *message* that's being considered, and the proposed BHandler *target*. You can alter the contents of the message, and alter or even replace the handler. If you replace the handler, the new handler must belong to the same looper as the original. The new handler is given an opportunity to filter the message before it's dispatched.

The return value must be one of these two values:

- **B_DISPATCH_MESSAGE**. The message and handler are passed (by the caller) to the looper's **DispatchMessage()** function.
- **B_SKIP_MESSAGE**. The message goes no furtherit's immediately thrown away by the caller.

The default version of this function returns **B_DISPATCH_MESSAGE**.

It's possible to call your **Filter()** function yourself (i.e. outside the message−passing mechanism), but keep in mind that it's the caller's responsibility to interpret the return value.

Rather than implement the **Filter()** function, you can supply the BMessageFilter with a **filter_hook** callback when you construct the object. If you do both, the **filter_hook** (and not **Filter()**) will be invoked when the object is asked to examine a message.

FiltersAnyCommand() see Command()

Looper()

```
BLooper *Looper(void) const
```

Returns the BLooper whose messages this object filters, or **NULL** if the BMessageFilter hasn't yet been assigned to a BHandler or BLooper. To attach a BMessageFilter to a looper or handler, use **BLooper::AddCommonFilter()** or **BHandler::AddFilter()**.

MessageDelivery() , MessageSource()

```
message_delivery MessageDelivery(void) const

message_source MessageSource(void) const
```

These functions return constants, set when the BMessageFilter object was constructed, that describe the categories of messages that can be filtered. `MessageDelivery()` returns a constant that specifies how the message must be delivered (`B_DROPPED_DELIVERY`, `B_PROGRAMMED_DELIVERY`, or `B_ANY_DELIVERY`). `MessageSource()` returns how the source of the message is constrained (`B_LOCAL_SOURCE`, `B_REMOTE_SOURCE`, or `B_ANY_SOURCE`).

Operators

= (copy)

```
BMessageFilter &operator=(const BMessageFilter&)
```

Copies the filtering criteria and `filter_hook` pointer (if any) from the right−side object into the left−side object.

Constants and Defined Types

filter_hook

```
filter_result (*filter_hook)(BMessage *message,
      BHandler **target,
      BMessageFilter *messageFilter)
```

`filter_hook` defines the protocol for message−filtering functions. The first two arguments are the *message* that's being considered, and the proposed BHandler *target*. You can alter the contents of the message, and alter or even replace the handler. If you replace the handler, the new handler must belong to the same looper as the original. The new handler is given an opportunity to filter the message before it's dispatched.

messageFilter is a pointer to the object on whose behalf this function is being called; you mustn't delete this object. More than one BMessageFilter can use the same `filter_hook` function.

The return value must be one of these two values:

- `B_DISPATCH_MESSAGE`. The message and handler are passed (by the caller) to the looper's `DispatchMessage()` function.
- `B_SKIP_MESSAGE`. The message goes no furtherit's immediately thrown away by the caller.

It's possible to call your filter function yourself (i.e. outside the message−passing mechanism), but keep in mind that it's the caller's responsibility to interpret the return value.

You supply a BMessageFilter with a `filter_hook` function when you constuct the object. Alternatively, you can subclass BMessageFilter and provide an implementation of `Filter()`. If you do both, the `filter_hook` (and not `Filter()`) will be invoked when the object is asked to examine a message.

message_source

message_delivery

filter_result

BMessageQueue

Derived from: none

Declared in: be/app/MessageQueue.h

Library: libbe.so

Summary

The BMessageQueue class completes the implementation of BLooper by providing a first−in/first−out stack in which the looper can place in−coming BMessages. In general, the message dispatching mechanism of BLooper should suffice. However, if you ever need to manipulate a BMessage queue directly, you can do so.

Constructor and Destructor

BMessageQueue()

```
BMessageQueue(void)
```

Creates an empty BMessageQueue object.

~BMessageQueue()

```
virtual ~BMessageQueue()
```

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

Member Functions

AddMessage() , RemoveMessage()

```
void AddMessage(BMessage *message)

void RemoveMessage(BMessage *message)
```

AddMessage() adds *message* to the far end of the queue. **RemoveMessage()** removes a particular *message* from the queue and deletes it.

CountMessages() , IsEmpty()

```
int32 CountMessages(void) const

bool IsEmpty(void) const
```

CountMessages() returns the number of messages currently in the queue.

IsEmpty() returns **true** if the object doesn't contain any messages, and **false** otherwise.

FindMessage()

```
BMessage *FindMessage(int32 index) const

BMessage *FindMessage(uint32 what, int32 index = 0) const
```

`FindMessage()` returns a pointer to the *index*'th BMessage in the queue, where index 0 signifies the message that's been in the queue the longest. The second version lets you specify a *what* field value; in this case, only messages that match the *what* argument are counted. If no message matches the criteria, the functions return **`NULL`**.

The message is not removed from the message queue.

IsEmpty() see 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)
```

Removes and returns the oldest message from the queue. If the queue is empty, the function returns **`NULL`**.

See also: **`FindMessage()`**

RemoveMessage() see AddMessage()

Unlock() see Lock()

BMessageRunner

Derived from: (none)

Declared in: be/app/MessageRunner.h

Library: libbe.so

Allocation: Constructor only

Summary

The BMessageRunner class provides a handy mechanism for automatically sending an arbitrary message to a BMessenger at specified intervals. The application that creates the BMessageRunner can specify the message, the BMessenger to send the message to, how often to send the message, and how many times it should be sent.

The system roster handles actually dispatching the messages to the appropriate BMessengers at the desired time intervals; this class simply acts as an intermediary through which your application asks the roster to schedule sending the messages.

Constructor and Destructor

BMessageRunner()

`BMessageRunner(`BMessenger *target*, const BMessage **message*,
bigtime_t *interval*, int32 *count* = −1**`)`**

`BMessageRunner(`BMessenger *target*, const BMessage **message*,
bigtime_t *interval*, int32 *count* , BMessenger *replyTo* **`)`**

Tells the roster to send the specified *message* to the *target* BMessenger every *interval* microseconds. The message will be sent *count* times (if *count* is −1, the message will be sent forever, or until the BMessageRunner is reconfigured or deleted).

The second form of the constructor lets the application specify, in *replyTo*, the BMessenger to which replies to the message should be sent.

The BMessageRunner can be reconfigured (to change the *interval* or *count*) by calling **`SetInterval()`** and **`SetCount()`**.

After constructing a BMessageRunner, you should call **`InitCheck()`** to ensure that the object was created properly.

~BMessageRunner()

virtual **`~BMessageRunner()`**

Asks the roster to stop sending the message.

Member Functions

GetInfo()

status_t **`GetInfo(`**bigtime_t **interval*, int32 **count***`)`** const

`GetInfo()` returns in *interval* the time in microseconds that will pass between messages being sent, and in *count* the number of times the message will be sent.

RETURN CODES

`B_OK`. Information returned successfully.

- **`B_NAME_NOT_FOUND`**. The roster returned invalid information about the BMessenger.
- **`B_BAD_VALUE`**. The roster returned invalid information about the BMessenger.
- Other errors. In general, getting an error back from this function is a bad thing.

InitCheck()

```
status_t InitCheck(void) const
```

`InitCheck()` returns a result code indicating **`B_OK`** if the BMessageRunner constructor executed sucessfully, or some other value if an error occurred setting up the object. You should call this immediately after creating a BMessageRunner, and shouldn't use the object if this function returns anything but **`B_OK`**.

SetCount() , SetInterval()

```
status_t SetCount(int32 count)

status_t SetInterval(bigtime_t interval)
```

`SetCount()` sets the number of times the BMessageRunner will send the message. If you want the message to be sent forever (until the object is deleted or **`SetCount()`** is called again), specify −1.

`SetInterval()` sets the number of microseconds that will pass between messages being sent.

BMessenger

Derived from: none

Declared in: be/app/Messenger.h

Library: libbe.so

Allocation: Stack or constructor

Summary

A BMessenger represents and sends messages to a *message target*, where the target is a BLooper and, optionally, a specific BHandler within that looper. The target can live in the same application as the BMessenger (a *local target*), or it can live in some other application (a *remote target*).

BMessenger's most significant function is **`SendMessage()`**, which sends its argument BMessage to the target.

For a local target, **`SendMessage()`** is roughly equivalent, in terms of efficiency, to posting a message directly to the BMessenger's target (i.e **`BLooper::PostMessage()`**).

The global **`be_app_messenger`** BMessenger pointer, which targets **`be_app`**'s main message loop, is automatically initialized for you when you create your BApplication object. You can use it wherever BMessengers are called for.

Constructor and Destructor

BMessenger()

`BMessenger(`const BHandler **handler*,
    const BLooper **looper* = NULL,
    status_t **error* = NULL**`)`**

`BMessenger(`const char **signature*,
    team_id *team* = 1,
    status_t **error* = NULL**`)`**

`BMessenger(`const BMessenger &*messenger***`)`**

`BMessenger(`void**`)`**

Creates a new BMessenger and sets its target to a local *looper*/*handler*, to the (running) application identified by *signature* or *team*, or to the target of some other *messenger*.

- **Looper/handler**. To target a looper, supply a *looper* and pass a **`NULL`** *handler*. When the messenger sends a message, the message will be handled by *looper*'s preferred handler. If you want the message to be sent to a specific handler within a looper, supply a *handler* and pass a **`NULL`** *looper*. The handler must already be attached to a looper, and can't switch loopers after this BMessenger is constructed.
- **Signature or team**. If you supply a *signature* but leave *team* as 1, the messenger targets an app with that signature. (The app must already be running; in the case of multiple instances of a running app, the exact instance is indeterminate) If you supply a *team* but no *signature*, you target exactly that team, regardless of signature. By supplying both a *team* and a *signature*, you can specify a specific instance of an app. In this case, *team* must be an app that has the proper *signature*.

  Messages sent to a remote target are received and handled by the remote application's BApplication object.

The BMessenger doesn't own its target.

RETURN CODES

The constructor places an error code in *error* (if provided).

- **`B_OK`**. The target was properly set.
- **`B_BAD_VALUE`**. The application identified by *signature* couldn't be found, or both *handler* and *looper* are invalid.
- **`B_BAD_TEAM_ID`**. Invalid *team*.
- **`B_MISMATCHED_VALUES`**. *team* isn't a *signature* app, or *handler* is associated with a BLooper other than *looper*.
- **`B_BAD_HANDLER`**. *handler* isn't associated with a BLooper (

~BMessenger()

```
~BMessenger()
```

Frees the BMessenger; the target isn't affected.

Member Functions

IsTargetLocal() see Target()

IsValid()

```
bool IsValid(void) const
```

Returns **`true`** if the target looper, whether local or remote, still exists.

This function doesn't tell you whether the looper is actually ready to receive messages, or whether the handler (if it was specified in the constructor) exists. In other words, a valid BMessenger is no guarantee that a message will actually get to the target.

LockTarget() , LockTargetWithTimeout()

```
bool LockTarget(void) const

status_t LockTargetWithTimeout(bigtime_t timeout) const
```

These functions apply to local targets only.

These functions attempt to lock the target looper in the manner of the similarly named BLooper functions (see **`BLooper::LockTarget()`**). In addition to the error codes reported there, these functions return **`false`** and **`B_BAD_VALUE`** (respectively) if the target isn't local, or if the looper is otherwise invalid.

SendMessage()

```
status_t SendMessage(BMessage *message,
           BMessage *reply,
     bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT,
     bigtime_t replyTimeout = B_INFINITE_TIMEOUT) const

status_t SendMessage(BMessage *message,
           BHandler *replyHandler = NULL,
     bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT) const

status_t SendMessage(BMessage *message,
     BMessenger *replyMessenger,
     bigtime_t deliveryTimeout = B_INFINITE_TIMEOUT) const

status_t SendMessage(uint32 command, BMessage *reply) const
```

status_t **SendMessage(**uint32 *command*, BHandler **replyHandler* = NULL**)** const

Sends a copy of *message* (or a BMessage based on a *command* constant) to the object's target. The caller retains ownership of *message*. The function doesn't return until the message has been delivered; if you're sending a *message* (as opposed to a *command* constant) you can set a microsecond delivery timeout through *deliveryTimeout*.

The target can respond to the message:

- If you supply a *reply* BMessage, the response is synchronous, with an optional timeout (*replyTimeout*) that starts ticking after the original message has been delivered. If the response times out, or the target deletes the original message without responding, the *reply***>**what is set to **`B_NO_REPLY`**. The caller is responsible for allocating and freeing *reply*. *message* and *reply* can be the same object.

Use caution when requesting a synchronous reply: If you call **`SendMessage()`** from the target looper's thread, you'll deadlock (or, at best, time out).

- If you supply a reply target (*replyMessenger* or *replyHandler*), the response is asynchronous, and is sent to the reply target.
- If you supply neither a reply message nor a reply target, the target's response is sent to **`be_app_messenger`**.

RETURN CODES

`B_OK`. The message was delivered (and the synchronous reply was received, if applicable).

- **`B_TIMED_OUT`**. *deliveryTimeout* expired; the message never made it to the target.
- **`B_WOULD_BLOCK`**. You requested a 0 *deliveryTimeout*, and the target's message queue is full.
- **`B_BAD_PORT_ID`**. The messenger's target is invalid, or the reply port was deleted while waiting for a reply (synchronous response requests only).
- **`B_NO_MORE_PORTS`**. You asked for a synchronous reply, but there are no more reply ports.

If you specified a *handler* when you constructed your BMessenger, and if that handler has since changed loopers, **`SendMessage()`** won't deliver its message, but it doesn't complain (it returns **`B_OK`**).

Target() , IsTargetLocal() , Team()

BHandler ***Target(**BLooper ***looper***)** const

bool **IsTargetLocal(**void**)** const

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

`Target()` returns the BMessenger's handler (directly) and looper (by reference in *looper*). This function only works for local targets. If **`Target()`** returns **`NULL`**, it can mean one of four things:

- The target is remote; *looper* is set to **`NULL`**.
- The BMessenger hasn't been initialized; *looper* is set to **`NULL`**.
- The handler is the looper's preferred handler; *looper* will be valid.
- The handler has been deleted; *looper* will be valid given that it hasn't been deleted as well.

`IsTargetLocal()` returns true if the target is local. **`Team()`** returns a target's team.

Team() see Target()

Operators

= (assignment)

```
BMessenger &operator =(const BMessenger&)
```

Sets the left−side BMessenger's target to that of the right−side object.

== (equality)

```
bool operator ==(const BMessenger&) const
```

Two BMessengers are equal if they have the same target.

BPropertyInfo

Derived from: BFlattenable

Declared in: be/app/PropertyInfo.h

Library: libbe.so

Summary

BPropertyInfo is a simple class that manages scripting. A program describes its scripting interface to a BPropertyInfo object through an array of property_info structures, with each entry describing a piece of the scripting suite. The structure definition:

```
   struct
property_info
 {
      char *name;
      uint32 commands[10];
      uint32 specifiers[10];
      char *usage;
      uint32 extra_data;
   };
```

- *name* provides the name of the property this structure describes.
- *commands* is a zero−terminated array of commands understood by the property, i.e. **B_GET_PROPERTY**. If the first element is 0, it represents a wildcard matching all possible commands.
- *specifiers* is a zero−terminated array of the specifiers understood by the property, i.e. **B_DIRECT_SPECIFIER**. If the first element is 0, it represents a wildcard matching all possible specifiers.
- *usage* gives a human−readable string describing the property and its allowable commands and specifiers.
- *extra_data* is an area free for general use; the operating system does not touch its contents.

A BPropertyInfo is instantiated by passing a zero−terminated array of property_info to its constructor. A typical initialization of BPropertyInfo looks like:

```
   static property_info prop_list[] = {
      { "duck", {B_GET_PROPERTY, B_SET_PROPERTY, 0},
      {B_DIRECT_SPECIFIER, B_INDEX_SPECIFIER, 0}, "get or set duck"},
      { "head", {B_GET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "get head"},
      { "head", {B_SET_PROPERTY, 0}, {B_DIRECT_SPECIFIER, 0}, "set head"},
      { "feet", {0}, {0}, "can do anything with his orange feet"},
      0 // terminate list
   };

   BPropertyInfo prop_info(prop_list);
```

Since BPropertyInfo only stores a pointer to the array, it is important that the life span of the array is at least as long as that of the BPropertyInfo object.

Notice that BPropertyInfo doesn't impose any particular structure upon the array; in particular, not all commands and specifiers for a given property need be placed in a single entry in the array. You are free to organize your scripting suite in whatever manner is most convenient for your particular object.

BPropertyInfo is a descendant of BFlattenable, and can therefore be used to store a description of an object's supported scripting suite. This is particularly useful when overriding **`GetSupportedSuites()`**:

```
   status_t MyHandler::GetSupportedSuites(BMessage *msg)
   {
      msg->AddString("suites", "suite/vnd.Me-my_handler");
      BPropertyInfo prop_info(prop_list);
      msg->AddFlat("messages", &prop_info);
      return baseClass::GetSupportedSuites(msg);
   }
```

Naturally, BPropertyInfo is equally as useful in interpreting the results obtained from querying an object for its supported suites.

BPropertyInfo defines the **`FindMatch()`** method designed to simplify the implementation of **`ResolveSpecifier()`**. It returns the index of the property info matching the description given to it, or −1 if none match. This reduces **`ResolveSpecifier()`** in the simplest cases to:

```
   BHandler *MyHandler::ResolveSpecifier(BMessage *msg, int32 index,
      BMessage *spec, int32 form, const char *prop)
   {
      BPropertyInfo prop_info(prop_list);
      if (prop_info.FindMatch(msg, index, spec, form, prop) >= 0)
      return this;
      return baseClass::ResolveSpecifier(msg, index, spec, form, prop);
   }
```

Of course, for more complicated objects, **`ResolveSpecifier()`** may need to set the target handler to an object other than itself, so more processing may be required. In those cases, the object can use the index returned by **`FindMatch()`** to help it determine the target of the scripting message.

Constructor and Destructor

BPropertyInfo()

`BPropertyInfo(property_info *p = NULL, bool free_on_delete = false)`

Initializes the object with the specified zero−terminated array *p* of property_info. Passing **`true`** in *free_on_delete* instructs the object to free the memory associated with the property_info when the object is destroyed. BPropertyInfo does not copy the array, so it is important that the array is not deleted or otherwise destroyed while the BPropertyInfo is in use.

~BPropertyInfo()

`~BPropertyInfo()`

If *free_on_delete* set to **`true`** in the constructor, the destructor frees all memory associated with the property_info. Otherwise, does nothing.

Member Functions

AllowsTypeCode()

Implementation detail. See **`BFlattenable::AllowsTypeCode()`**.

FindMatch()

```
int32 FindMatch(BMessage *msg, int32 index, BMessage *spec, int32 form,
    const char *prop, void *data = NULL) const
```

Passed a property name in *prop*, a specifier in *form*, and a command in *msg−>what*, searches the property_info array for an item supporting the specified scripting request. If *index* is nonzero, then **`FindMatch()`** only searches those property_info structures with the wildcard command (first element of command array equal to 0). Otherwise, it searches through all available property_info structures for a match. If a match is found, it fills the memory at *data* with the contents of the extra_data field of the match and returns the index of the match in the array. Otherwise, it returns **`B_ERROR`**.

Flatten

Implementation detail. See **`BFlattenable::Flatten()`**.

FlattenedSize()

Implementation detail. See **`BFlattenable::FlattenedSize()`**.

IsFixedSize()

Implementation detail. See **`BFlattenable::IsFixedSize()`**.

TypeCode()

Implementation detail. See **`BFlattenable::TypeCode()`**.

PrintToStream()

`void PrintToStream(void) const`

Prints information about the BPropertyInfo to standard output.

PropertyInfo()

```
const property_info *PropertyInfo(void) const
```

Returns the property_info list associated with the object.

Unflatten()

Implementation detail. See **`BFlattenable::Unflatten()`**.

BRoster

Derived from: none

Declared in: be/app/Roster.h

Library: libbe.so

Summary

The BRoster object represents a service that keeps a roster of all applications currently running. It can provide information about any of those applications, activate one of them, 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 BRoster object is constructed and assigned to a global variable, **`be_roster`**. You always access the roster through this variable; you never have to instantiate a BRoster in application code.

The BRoster identifies applications in three ways:

- By **`entry_ref`** references to the executable files where they reside.
- By their signatures. The signature is a unique identifier for the application assigned as a file−system attribute or 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.
- At run time, by their **`team_id`**s. A team is a group of threads sharing an address space; every application is a team.

If an application is launched more than once, the roster will include one entry for each instance of the application that's running. These instances will have the same signature, but different team identifiers.

Constructor and Destructor

BRoster()

```
BRoster(void)
```

Sets up the object's connection to the roster service.

When an application constructs its BApplication object, the system constructs a BRoster object and assigns it to the **`be_roster`** global variable. A BRoster is therefore readily available from the time the application is initialized until the time it quits; you don't have to construct one. The constructor is public only to give programs that don't have BApplication objects access to the roster.

~BRoster()

```
~BRoster()
```

Does nothing.

Member Functions

ActivateApp()

```
status_t ActivateApp(team_id team) const
```

Activates the *team* application (by bringing one of its windows to the front and making it the active window). This function works only if the target application has a window on−screen. The newly activated application is notified with a **`B_APP_ACTIVATED`** message.

See also: **`BApplication::AppActivated()`**

AddToRecentDocuments() , GetRecentDocuments()

```
void AddToRecentDocuments(const entry_ref *document,
    const char *appSig = NULL) const

void GetRecentDocuments(BMessage *refList, int32 maxCount,
    const char *ofType = NULL,
    const char *openedByAppSig = NULL) const

void GetRecentDocuments(BMessage *refList, int32 maxCount,
    const char *ofTypeList[] = NULL, int32 ofTypeListCount,
    const char *openedByAppSig = NULL) const
```

AddToRecentDocuments() adds the document file specified by *document* to the list of recent documents. If you wish to record that a specific application used the document, you can specify the signature of that application using the *appSig* argument; otherwise you can specify **NULL**.

GetRecentDocuments() returns a list of the most recent documents. The BMessage *refList* will be filled out with information about the *maxCount* most recently used documents. If you want to obtain a list of documents of a specific type, you can specify a pointer to that MIME type string in the *ofType* argument. Likewise, if you're only interested in files that want to be opened by a specific application, specify that application's signature in *openedByAppSig*; if you don't care, pass **NULL**.

If you want to get a list of files of multiple types, you can specify a pointer to an array of strings in *ofTypeList*, and the number of types in the list in *ofTypeListCount*.

Specifying **NULL** for *ofType* will fetch all files of all types.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting list of files.

AddToRecentFolders() , GetRecentFolders()

```
void AddToRecentFolders(const entry_ref *folder,
    const char *appSig = NULL) const

void GetRecentFolders(BMessage *refList, int32 maxCount,
    const char *openedByAppSig = NULL) const
```

AddToRecentFolders() adds the folder specified by *folder* to the list of recent folders. If you wish to record that a specific application used the folder, you can specify the signature of that application using the *appSig* argument; otherwise you can use **NULL**.

GetRecentFolders() returns a list of the most recently−accessed folders. The BMessage *refList* will be filled out with information about the *maxCount* most recently used folders. If you're only interested in folders that were used by a specific application, specify that application's signature in *openedByAppSig*; if you don't care, pass **NULL**.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting list of folders.

Broadcast()

```
status_t Broadcast(BMessage *message) const

status_t Broadcast(BMessage *message, BMessenger reply_to) const
```

Sends the *message* to every running application, except to those applications (**B_ARGV_ONLY**) that don't accept messages. The message is sent asynchronously with a timeout of 0. As is the case for other message−sending functions, the caller retains ownership of the *message*.

This function returns immediately after setting up the broadcast operation. It doesn't wait for the messages to be sent and doesn't report any errors encountered when they are. It returns an error only if it can't start the broadcast operation. If successful in getting the operation started, it returns **B_OK**.

Replies to the broadcasted message will be sent via the *reply_to* BMessenger, if specified. If *reply_to* is absent, the replies will be lost.

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

FindApp()

```
status_t FindApp(const char *type, entry_ref *app) const

status_t FindApp(entry_ref *file, entry_ref *app) const
```

Finds the application associated with the MIME data *type* or with the specified *file*, and modifies the *app* `entry_ref` structure so that it refers to the executable file for that application. If the *type* is an application signature, this function finds the application that has that signature. Otherwise, it finds the preferred application for the type. If the *file* is an application executable, **`FindApp()`** merely copies the file reference to the *app* argument. Otherwise, it finds the preferred application for the file type.

In other words, this function goes about finding an application in the same way that **`Launch()`** finds the application it will launch.

If it can translate the *type* or *file* into a reference to an application executable, **`FindApp()`** returns **`B_OK`**. If not, it returns an error code, typically one describing a file system error.

See also: **`Launch()`**

GetAppInfo() , GetRunningAppInfo() , GetActiveAppInfo()

status_t **`GetAppInfo(`**const char **signature*, app_info **appInfo***`)`** const

status_t **`GetAppInfo(`**entry_ref **executable*, app_info **appInfo***`)`** const

status_t **`GetRunningAppInfo(`**team_id *team*, app_info **appInfo***`)`** const

status_t **`GetActiveAppInfo(`**app_info **appInfo***`)`** const

These functions return (in *appInfo*) information about a specific application. In all cases, the application must be running.

- **`GetAppInfo()`** finds an app that has the given *signature*, or that was launched from the *executable* file. If there's more than one such app, the function chooses one at random.
- **`GetRunningAppInfo()`** reports on the app that corresponds to the given *team* identifier.
- **`GetActiveAppInfo()`** reports on the currently active app.

If they're able to fill in the **`app_info`** structure with meaningful values, these functions return **`B_OK`**. **`GetActiveAppInfo()`** returns **`B_ERROR`** if there's no active application. **`GetRunningAppInfo()`** returns **`B_BAD_TEAM_ID`** if *team* isn't a valid team identifier for a running application. **`GetAppInfo()`** returns **`B_ERROR`** if the application isn't running.

The **`app_info`** structure contains the following fields:

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.

uint32 **`flags`**
A mask that contains information about the behavior of the application.

entry_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()`**.)

char **`signature`**[]
The signature of the application. (This will be the same as the *signature* passed to **`GetAppInfo()`**.)

The **`flags`** mask can be tested (with the bitwise **&** operator) against these two constants:

- **`B_BACKGROUND_APP`**. The application won't appear in the Deskbar's application list.
- **`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:

```
unit32 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()`, `BApplication::GetAppInfo()`

GetAppList()

void **GetAppList(**BList **teams***)** const

void **GetAppList(**const char **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:

```
BList *teams = new BList;
be_roster->GetAppList(teams);
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

GetRecentApps()

void **GetRecentApps(**BMessage **refList*, int32 *maxCount***)** const

`GetRecentApps()` returns a list of the most recently−launched applications. The BMessage *refList* will be filled out with information about the *maxCount* most recently−launched applications.

The resulting *refList* will have a field, "refs", containing the entry_refs to the resulting applications.

GetRecentDocuments() see AddToRecentDocuments()

GetRecentFolders() see AddToRecentFolders()

IsRunning() see TeamFor()

Launch()

status_t **Launch(**const char **type*,
BMessage **message* = NULL,
team_id **team* = NULL**)** const

status_t **Launch(**const char **type*,
BList **messages*,
team_id **team* = NULL**)** const

status_t **Launch(**const char **type*,
int *argc*,
char ***argv*,
team_id **team* = NULL**)** const

status_t **Launch(**const entry_ref **file*,
const BMessage **message* = NULL,
team_id **team* = NULL**)** const

status_t **Launch(**const entry_ref **file*,
const BList **messages*,
team_id **team* = NULL**)** const

```
status_t Launch(const entry_ref *file,
     int argc,
     const char * const char *argv,
     team_id *team = NULL) const
```

Launches the application associated with a MIME *type* or with a particular *file*. If the MIME *type* is an application signature, this function launches the application with that signature. Otherwise, it launches the preferred application for the type. If the *file* is an application executable, it launches that application. Otherwise, it launches the preferred application for the file type and passes the *file* reference to the application in a **B_REFS_RECEIVED** message. In other words, **Launch()** finds the application to launch just as **FindApp()** finds the application for a particular *type* or *file*.

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 caller retains ownership of the BMessage objects (and the container BList); they won't be deleted for you.

Sending an on−launch message is appropriate 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.

If the target application is already running, **Launch()** won't launch it again, unless it permits multiple instances to run concurrently (it doesn't wait for the messages to be sent or report errors encountered when they are). It fails for **B_SINGLE_LAUNCH** and **B_EXCLUSIVE_LAUNCH** applications that have already been launched. Nevertheless, it assumes that you want the messages to get to the application and so delivers them to the currently running instance.

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_OK**. If unsuccessful, it sets the *team* variable to 1 and returns an error code, typically one of the following:

- **B_BAD_VALUE**. The *type* or *file* is not valid, or 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_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.
- A file system error. The *file* or *type* can't be matched to an application.

See also: the BMessenger class, **GetAppInfo()**, **FindApp()**

StartWatching() , StopWatching()

```
status_t StartWatching(BMessenger target, uint32 events = B_REQUEST_LAUNCHED | B_REQUEST_QUIT) const

status_t StopWatching(BMessenger target) const
```

StartWatching() initiates the application event monitor, which is used for keeping track of events such as application launches. The caller specifies the events to monitor through the *events* argument; *target* is the BMessenger to which the corresponding notification messages are sent. The *events* flags and the corresponding messages are listed below:

B_REQUEST_LAUNCHED	**B_SOME_APP_LAUNCHED**
B_REQUEST_QUIT	**B_SOME_APP_QUIT**
B_REQUEST_ACTIVATED	**B_SOME_APP_ACTIVATED**

The fields in a notification message describe the application that was launched, quit, or activated:

"mime_sig"	**B_STRING_TYPE**	MIME signature
"team"	**B_INT32_TYPE**	**team_id**
"thread"	**B_INT32_TYPE**	**thread_id**
"flags"	**B_INT32_TYPE**	application flags
"ref"	**B_REF_TYPE**	executable's **entry_ref**

StopWatching() terminates the application monitor previously initiated for a given BMessenger.

StartWatching()

TeamFor() , IsRunning()

```
team_id TeamFor(const char *signature) const

team_id TeamFor(entry_ref *executable) const

bool IsRunning(const char *signature) const

bool IsRunning(entry_ref *executable) const
```

Both these functions query whether the application identified by its *signature* or by a reference to its *executable* file is running. **TeamFor()StartWatching()** 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 in the Application Kit. 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`** and **`B_KEY_DOWN`**), only those that objects in this kit handle are listed here. Those that designate interface messages are documented in the chapter on the Interface Kit.

Global Variables

be_app

Declared in: be/app/Application.h

```
BApplication *be_app
```

This variable provides global access to the BApplication object. It's initialized by the BApplication constructor.

See also: the BApplication class

be_app_messenger

Declared in: be/app/Application.h

```
BMessenger *be_app_messenger
```

This variable provides global access to a BMessenger object whose target is **`be_app`**. It's initialized by the BApplication constructor.

See also: the BApplication class

be_clipboard

Declared in: be/app/Clipboard.h

```
BClipboard *be_clipboard
```

This variable gives applications access to the system clipboardthe shared repository of data for cut, copy, and paste operations. It's initialized at startup.

See also: the BClipboard class

be_roster

Declared in: be/app/Roster.h

```
const BRoster *be_roster
```

This variable points to the application's global BRoster object. The BRoster keeps a roster of all running applications and can add applications to the roster by launching them. It's initialized when the application starts up.

See also: the BRoster class

Constants

Application Flags

Declared in: be/app/Roster.h

`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

Declared in: be/app/AppDefs.h

`B_QUIT_REQUESTED`
`B_READY_TO_RUN`
`B_APP_ACTIVATED`
`B_ABOUT_REQUESTED`
`B_QUIT_REQUESTED`
`B_ARGV_RECEIVED`
`B_REFS_RECEIVED`
`B_PULSE`

These constants represent the system messages that are recognized and given special treatment by BApplication and BLooper dispatchers. 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 30 of the BApplication class

Cursor Constants

Declared in: be/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()`**

filter_result Constants

Declared in: be/app/MessageFilter.h

`B_SKIP_MESSAGE`
`B_DISPATCH_MESSAGE`

These constants list the possible return values of a filter function.

See also: **`BMessageFilter::Filter()`**

Launch Constants

Declared in: be/app/Roster.h

`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

Looper Port Capacity

Declared in: be/app/Looper.h

`B_LOOPER_PORT_DEFAULT_CAPACITY`

This constant records the default capacity of a BLooper's port. The default is 100 slots; a greater or smaller number can be specified when constructing the BLooper.

See also: the BLooper constructor

Message Constants

Declared in: be/app/AppDefs.h

`B_REPLY`
`B_NO_REPLY`
`B_MESSAGE_NOT_UNDERSTOOD`
`B_SAVE_REQUESTED`
`B_CANCEL`
`B_SIMPLE_DATA`
`B_MIME_DATA`
`B_ARCHIVED_OBJECT`
`B_UPDATE_STATUS_BAR`
`B_RESET_STATUS_BAR`
`B_NODE_MONITOR`
`B_QUERY_UPDATE`
`B_CUT`
`B_COPY`
`B_PASTE`
`B_SELECT_ALL`

B_SET_PROPERTY
B_GET_PROPERTY
B_CREATE_PROPERTY
B_DELETE_PROPERTY
B_GET_SUPPORTED_SUITES

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

message_delivery Constants

Declared in: be/app/MessageFilter.h

B_ANY_DELIVERY
B_DROPPED_DELIVERY
B_PROGRAMMED_DELIVERY

These constants distinguish the delivery criterion for filtering a BMessage.

See also: the BMessageFilter constructor

message_source Constants

Declared in: be/app/MessageFilter.h

B_ANY_SOURCE
B_REMOTE_SOURCE
B_LOCAL_SOURCE

These constants list the possible constraints that a BMessageFilter might impose on the source of the messages it filters.

See also: the BMessageFilter constructor

Message Specifiers

Declared in: be/app/Message.h

B_NO_SPECIFIER
B_DIRECT_SPECIFIER
B_INDEX_SPECIFIER
B_REVERSE_INDEX_SPECIFIER
B_RANGE_SPECIFIER
B_REVERSE_RANGE_SPECIFIER
B_NAME_SPECIFIER
B_ID_SPECIFIER

B_SPECIFIERS_END	= 128

These constants fill the **what** slot of specifier BMessages. Each constant indicates what other information the specifer contains and how it should be interpreted. For example, a **B_REVERSE_INDEX_SPECIFIER** message has an "index" field with an index that counts backwards from the end of a list. A **B_NAME_SPECIFIER** message includes a "name" field that names the requested item.

Defined Types

app_info

Declared in: be/app/Roster.h

```
typedef struct {
      thread_id thread;
      team_id team;
      port_id port;
      uint32 flags;
      entry_ref ref;
      char signature[B_MIME_TYPE_LENGTH];
            app_info(void);
            ~app_info(void);
    } app_info
```

This structure is used by BRoster's **GetAppInfo()**, **GetRunningAppInfo()**, and **GetActiveAppInfo()** functions to report information about an application. Its constructor ensures that its fields are initialized to invalid values. To get meaningful values for an actual application, you must pass the structure to one of the BRoster functions. See those functions for a description of the various fields.

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

filter_result

Declared in: be/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

Declared in: be/app/MessageFilter.h

```
typedef enum { . . . } message_delivery
```

This type enumerates the delivery criteria for filtering a message.

See also: the BMessageFilter constructor

message_source

Declared in: be/app/MessageFilter.h

```
typedef enum { . . . } message_source
```

This type enumerates the source criteria for filtering a message.

See also: the BMessageFilter constructor

The Application Kit: Master Index