9

In this chapter:
- *The Application Kit and Messages*
- *Application-Defined Messages*

Messages and Threads

Several years ago Be, Inc. set out to develop a new operating system—one they eventually dubbed the *Media OS*. The goal was to create an operating system that could keep up with the computationally intensive demands of media professionals who routinely worked with complex and resource-hungry graphics and audio files. From the start, Be knew that threads and messages would be of paramount importance. A multithreaded environment means that a single application can simultaneously carry out multiple tasks. On a single-processor machine, CPU idle time is reduced as the processor services one thread followed by another. On a multiprocessor machine, task time really improves as different CPUs can be dedicated to the servicing of different threads.

Threads can be considered roadways that allow the system to communicate with an object, one object to communicate with another object, and even one application to communicate with another application. Continuing with the analogy, messages are the vehicles that carry the information, or data, that is to be passed from one entity to another. Chapter 4, *Windows, Views, and Messages*, introduced messages, and seldom since then have a couple of pages passed without a direct or indirect reference to messages. In this chapter, I'll formally explain of how messages and threads work. In doing so, you'll see how your application can create its own messages and use them to let one object tell another object what to do. You'll see how sending a message can trigger an object to perform some desired action. You'll also see how a message can be filled with any manner of data before it's sent. Once received, the recipient object has access to any and all of the data held within the message.

The Application Kit and Messages

Servers are background processes that exist to serve the basic, low-level needs of applications. The BeOS end user makes indirect use of servers every time he or she runs a Be application. As a Be programmer, you make more direct use of servers by using the BeOS application programming interface. The classes of the API are organized into the software kits that have been discussed at length throughout this book. The most basic, and perhaps most important, of these kits is the Application Kit. Among the classes defined in this kit is the `BApplication` class. Your application begins by creating a `BApplication` object. When an instance of that class is created, your application connects to the Application Server, and can make use of all the services provided by that server. Tasks handled by the Application Server include the provision of windows, the handling of the interaction between these windows, and the monitoring and reporting of user events such as mouse button clicks. In short, the Application Server, and indirectly the classes of the Application Kit, allow the system to communicate with an application. This communication takes place via messages that travel by way of threads.

The classes of the Application Kit (shown in Figure 9-1) fall into the four categories listed below. Of these groupings, it's messaging that's the focus of this chapter.

Messaging
: The Application Server delivers system messages to an application. Additionally, an application can send application-defined messages to itself (the purpose being to pass information from one object to another). The Application Kit defines a number of message-related classes that are used to create, deliver, and handle these messages. Among these classes are: `BMessage`, `BLooper`, and `BHandler`.

BApplication class
: An application's single `BApplication` object is the program's interface to the Application Server.

BRoster class
: The system keeps a roster, or list, of all executing applications. Upon the launch of your application, a `BRoster` object is automatically created. In the event that your program needs to communicate with other running applications, it can do so by accessing this `BRoster` object.

BClipboard class
: The system keeps a single clipboard as a repository for information that can be shared—via copying, cutting, and pasting—between objects in an application and between distinct applications. Upon launch, your application automatically creates a `BClipboard` object that is used to access the systemwide clipboard.

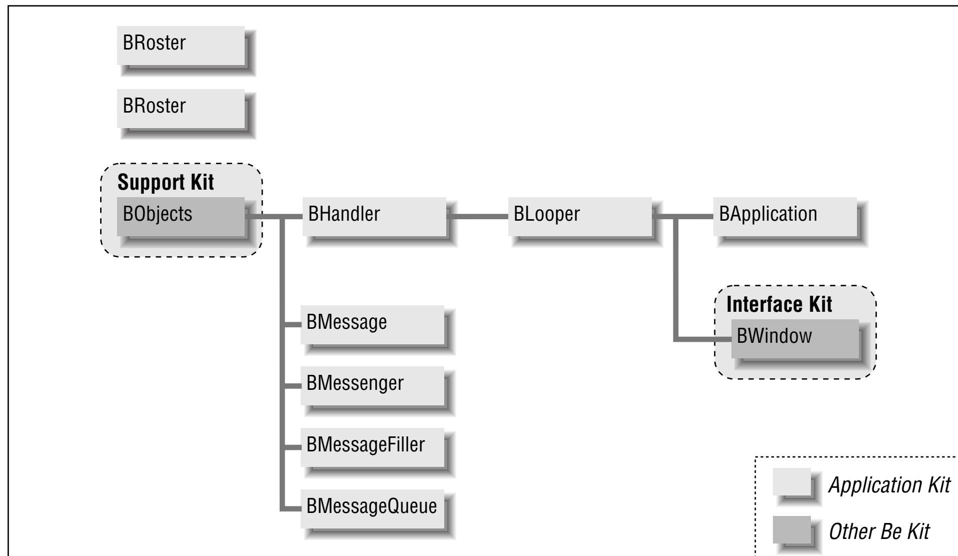


Figure 9-1. The inheritance hierarchy for the Application Kit

Messaging

The Application Kit defines the classes that allow an application to be multithreaded. While threads run independently, they do need a means of communicating with one another. So the Application Kit also defines classes that allow for the creation and delivery of messages.

The `BMessage` class is used to create message objects. A single message can contain as little or as much information as appropriate for its purpose. Once created within one thread, a message can be delivered to the same thread, a different thread in the same application, or to a thread in a different application altogether.

How a thread obtains a message and then handles that message is determined by the Application Kit classes `BLooper` and `BHandler`. A `BLooper` object runs a message loop in a thread. This message loop receives messages and dispatches each to a `BHandler` object. The handler object is responsible for handling the message as appropriate for the message type. Notice in Figure 9-1 that the `BLooper` class is derived from the `BHandler` class. This means that a looper object is also a handler object, and can thus pass a message to itself. While this may sound self-defeating, it actually serves as a quite useful mechanism for initiating and carrying out a task from within one object, such as a window (which, as shown in Figure 9-1, is both a looper and a handler). Throughout this chapter you'll see several examples of the creating of messages and the dispatching of these messages both by the object that created them and by other objects.

Because an application is multithreaded, more than one thread may attempt to access the same data. For read-only data (data that can't be written to or altered), that's not a problem. For read-write data, a problem could exist; if both accessing objects try to alter the same data at the same time, the result will be at best unpredictable and at worst disastrous. To prevent simultaneous data access, the BeOS allows for the locking of data. When one thread is about to access data, it can first lock it. While locked, other threads are prevented access. When a thread encounters locked data, it finds itself waiting in queue. Only after the thread that locked the data later unlocks it will other threads get a chance at access.

In many instances, the locking and unlocking of data is handled by the system. For instance, when a window receives a message (when a message enters the window thread's message loop), the `BWindow` object is locked until the message is handled. From Chapter 4 you know that a window object has a host of characteristics, such as size, that can be altered at runtime. If the window wasn't locked during message handling, and the message indicated that, say, the window should be resized, the possibility exists for a second such message to arrive at the same time and also attempt to change the values of the window object's screen coordinates.

Occasionally there'll be cases where your application is responsible for the locking and unlocking of data. For such occasions, the object to be locked will have `Lock()` and `Unlock()` member functions in its class definition. This chapter provides one such instance of manually locking and unlocking an object. If your program wants to add data to the clipboard (as opposed to the user placing it there by a copy or cut), it should first lock the clipboard (in case the user does in fact perform a copy or cut while your program is in the process of placing data on the clipboard!). This chapter's ClipboardMessage project shows how this is done.

Application Kit Classes

The previous section described the Application Kit classes directly involved with messages—the `BMessage`, `BLooper`, and `BHandler` classes. Other Application Kit classes, while not as important in terms of messaging, are still noteworthy. Be suggests that the collective message-related classes make up one of four Application Kit categories. The other three each contain a single class—`BApplication`, `BRoster`, and `BClipboard` class. Those three classes are discussed next.

BApplication class

By now you're quite familiar with the notion that every program must create a single instance of the `BApplication` class (or of an application-defined `BApplication`-derived class). The `BApplication` class is derived from the `BLooper` class, so an object of this class type runs its own message loop. A program's application object is connected to the Application Server, and system

messages sent to the program enter the application object's message loop. If a message is a system message (such as a B_QUIT_REQUESTED), it is eventually handled by a BApplication hook function (such as QuitRequested()). The BApplication class defines a MessageReceived() function that augments the BHandler version of this routine. If your program wants the application object to handle application-defined messages, it should override and augment the BApplication version MessageReceived(). To do that, your program defines a message constant, declares MessageReceived() in the BApplication-derived class declaration, and implements MessageReceived():

```
#define   MY_APP_DEFINED_MSG       'mymg'

class MyAppClass : public BApplication {

   public:
                        MyAppClass();
      virtual void      MessageReceived(BMessage* message);
};


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

      case MY_APP_DEFINED_MSG:
         // handle this type of message
         break;

      default:
         inherited::MessageReceived(message);
         break;
   }
}
```

Previous example projects haven't made direct use of MessageReceived() in the application object. This chapter's AlertMessage project (discussed in the "Message-posting example project" section) provides a specific example.

Like the BApplication class, the BWindow class is derived from BLooper. So, like an application object, a window object runs a message loop. And, again like an application object, a window object has a connection to the Application Server—so a window can be the recipient of system messages. Examples of these interface system messages include B_QUIT_REQUESTED, B_ZOOM, B_MOUSE_DOWN, B_KEY_DOWN, and B_WINDOW_RESIZED messages.

BRoster class

The system, of course, keeps track of all running applications. Some of the information about these processes is stored in a roster, or table, in memory. Much of this information about other executing applications is available to your executing application. Your program won't access this roster directly, though. Instead, it will rely on the `be_roster` global variable. When an application launches, an object of the `BRoster` class is automatically created and assigned to `be_roster`.

To garner information about or communicate via messages with another application, you simply refer to `be_roster` and invoke one of the `BRoster` member functions. Some of the important `BRoster` functions and their purposes include:

GetAppList()
: Returns an identifier for each running application.

GetAppInfo()
: Provides information about a specified application.

ActivateApp()
: Activates an already running application by bringing one of its windows to the front and activating it.

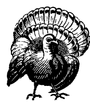
Broadcast()
: Broadcasts, or sends, a message to all currently running applications.

IsRunning()
: Determines if a specified application is currently running.

Launch()
: Locates an application on disk and launches it.

FindApp()
: Locates an application (as `Launch()` does), but doesn't launch it.

One of the ways `be_roster` identifies an application is by the program's signature (presenting you with another reason to make sure your application's signature is unique—as mentioned in Chapter 2, *BeIDE Projects*). The very simple RosterCheck example project in this chapter takes advantage of this in order to see how many instances of the RosterCheck program are currently running. RosterCheck allows itself to be launched more than once, but not more than twice.



When creating an application that is to allow for multiple instances of the program, you need make sure that the application flags field resource is set to multiple launch. Chapter 2 discusses this resource and how to set it. In short, you double-click on the project's resource file to open it, then click the Multiple Launch radio button in the Application Flags section.

The roster keeps track of each application that is running, including multiple instances of the same application. To check the roster and make use of the results, just a few lines of code in the application constructor are all that's needed:

```
MyHelloApplication::MyHelloApplication()
      : BApplication("application/x-dps-twoapps")
{
   BList  theList;
   long   numApps;

   be_roster->GetAppList("application/x-dps-twoapps", &theList);
   numApps = theList.CountItems();

   if (numApps > 2) {
      PostMessage(B_QUIT_REQUESTED);
      return;
   }

   BRect  aRect;

   aRect.Set(20, 30, 220, 130);
   fMyWindow = new MyHelloWindow(aRect);
}
```

When passed an application signature and a pointer to a `BList` object, the `BRoster` function `GetAppList()` examines the roster and fills in the list object with an item for each currently running application with the matching signature. To know what to do next, you need at least a passing familiarity with the `BList` class, a class not yet mentioned in this book.

The `BList` class is a part of the Support Kit, which defines datatypes, classes, and utilities any application can use. An instance of the `BList` class is used to hold a list of data pointers in an orderly fashion. Keeping data in a `BList` is handy because you can then use existing `BList` member functions to further organize or manipulate the data. The partial listing of the `BList` class hints at the things a list can do:

```
class BList {

public:
             BList(int32 itemsPerBlock = 20);
             BList(const BList&);
   virtual   ~BList();

   BList     &operator=(const BList &from);
   bool      AddItem(void *item);
   bool      AddItem(void *item, int32 atIndex);
   bool      AddList(BList *newItems);
   bool      AddList(BList *newItems, int32 atIndex);
   bool      RemoveItem(void *item);
   void      *RemoveItem(int32 index);
   bool      RemoveItems(int32 index, int32 count);
```

```
        bool       ReplaceItem(int32 index, void *newItem);
        void       MakeEmpty();

        void       SortItems(int (*cmp)(const void *, const void *));
        bool       SwapItems(int32 indexA, int32 indexB);
        bool       MoveItem(int32 fromIndex, int32 toIndex);

        void       *ItemAt(int32) const;
        void       *ItemAtFast(int32) const;
        void       *FirstItem() const;
        void       *LastItem() const;
        void       *Items() const;

        bool       HasItem(void *item) const;
        int32      IndexOf(void *item) const;
        int32      CountItems() const;
        bool       IsEmpty() const;

        ...
   }
```

The pointers that are stored in a list can reference any type of data, so the `BRoster` function `GetAppList()` stores a reference to each running application with the specified signature. After calling `GetAppList()` you can find out how many instances of the application in question are currently running—just invoke `CountItems()` to see how many items are in the list. That's exactly what I do in the RosterCheck project:

```
BList  theList;
long   numApps;

be_roster->GetAppList("application/x-dps-twoapps", &theList);
numApps = theList.CountItems();
```

After the above code executes, `numApps` holds the number of executing instances of the RosterCheck program (including the instance that's just been launched and is executing the above code). The following code limits the number of times the user can execute RosterCheck to two; if you try to launch RosterCheck a third time, the program will immediately quit:

```
if (numApps > 2) {
   PostMessage(B_QUIT_REQUESTED);
   return;
}
```

A more well-behaved version of RosterCheck would post an alert explaining why the program quit. It would also have some reason for limiting the number of instances of the program—my arbitrary limit of two exists so that I can demonstrate that the roster in general, and a `BRoster` member function in particular, work!

BClipboard class

The previous section described the system's application roster, the be_roster global object used to access the roster, and the BRoster class that defines the type of object be_roster is. The clipboard works in a similar vein: there's one system clipboard, it's accessed by a be_clipboard global object, and that object is of the Be class BClipboard.

Objects of some class types make use of be_clipboard without any intervention on your part. For instance, in Chapter 8, *Text*, you saw that a BTextView object automatically supports the editing functions cut, copy, paste, and select all. When the user cuts text from a BTextView object, the object places that text on the system clipboard. Because this clipboard is global to the system, the cut data becomes available to both the application from which the data was cut and any other application that supports the pasting of data.

As you may suspect, when editing takes place in a BTextView object, messages are involved. In particular, the BTextView object responds to B_CUT, B_COPY, B_PASTE, and B_SELECT_ALL messages. The B_CUT and B_COPY messages add to the clipboard the currently selected text in the text view object that's the focus view. The B_PASTE message retrieves text from the clipboard and pastes it to the insertion point in the text view object that's the focus view. If you want your program to manually force other text to be added to the clipboard, or if you want your program to manually retrieve the current text from the clipboard without pasting it anywhere, you can do so by directly accessing the clipboard.

To fully appreciate how to work with the clipboard, you'll want to read this chapter's "Working with BMessage Objects" section. In particular, the "Data, messages, and the clipboard" subsection discusses messages as they pertain to the clipboard, and the "Clipboard example project" subsection provides an example of adding text directly to the clipboard without any intervention on the part of the user.

Application-Defined Messages

Up to this point, you've dealt mostly with system messages—messages generated and dispatched by the system. The Message Protocols appendix of the Be Book defines all the system messages. In short, system messages fall into the following categories:

Application system messages

Such a message concerns the application itself, and is delivered to the BApplication object. The application handles the message by way of a hook function, as described in Chapter 4. B_QUIT_REQUESTED is one application message with which you're familiar.

Interface system messages

Such a message concerns a single window, and is delivered to a BWindow object. The window handles the message by way of a hook function, or, if the message affects a view in the window, passes it on to the BView object, which handles it by way of a hook function. A B_WINDOW_ACTIVATED message is an example of an interface message that would be handled by a window, while a B_MOUSE_DOWN message is an example of an interface message that would be passed on to a view (the view the cursor was over at the time of the mouse button click) for handling.

Standard messages

Such a message is produced by either the system or application, but isn't handled by means of a hook function. The editing messages covered in Chapter 8—B_CUT, B_COPY, B_PASTE, and B_SELECT_ALL—are examples of standard messages. When a user selects text in a BTextView object and presses Command-x, the affected window generates a B_CUT message that is sent to the text view object. That object automatically handles the text cutting by invoking the BTextView function Cut().

The system and standard messages are important to making things happen in your application—they allow the user to interact with your program. But these messages are only a part of the powerful Be messaging system. Your application is also free to define its own message constants, create messages of these application-defined types, add data to these messages, and then pass the messages on to other object or even other applications.

Message Handling

An application-defined message can be issued automatically in response to a user action such as a menu item selection or a control activation. Your application can also issue, or post, a message explicitly without any user intervention. Before going into the details of application-defined messages, a quick review of system messages will minimize confusion between how these different types of messages are handled.

System message handling

When an application receives a system message, it is dispatched by sending the message to the affected BHandler object. That object then invokes a hook function—a function specifically implemented to handle one particular type of system message.

A system message is the result of an action external to the application. The message is generated by the operating system, and is delivered to an application

object or a window object. That object, or an object the message is subsequently passed to, invokes the appropriate hook function.

As an example, consider a mouse button click. The click of a mouse button inspires the Application Server to generate a B_MOUSE_DOWN message. The server passes this message to the affected window (the window under the cursor at the time of the mouse button click). A BWindow object is a looper, so the window has its own thread that runs a message loop. From this loop, the message is dispatched to a handler, which in this example is the affected view (the view under the cursor at the time of the mouse button click). A BView object is a handler, so it can be the recipient of a passed message. A handler object in general, and a BView-derived object in particular, has its own hook functions (either inherited from the BView class or overridden). For a B_MOUSE_DOWN message, the pertinent function the view invokes is the BView hook function MouseDown(). Figure 9-2 illustrates system message dispatching for this situation.

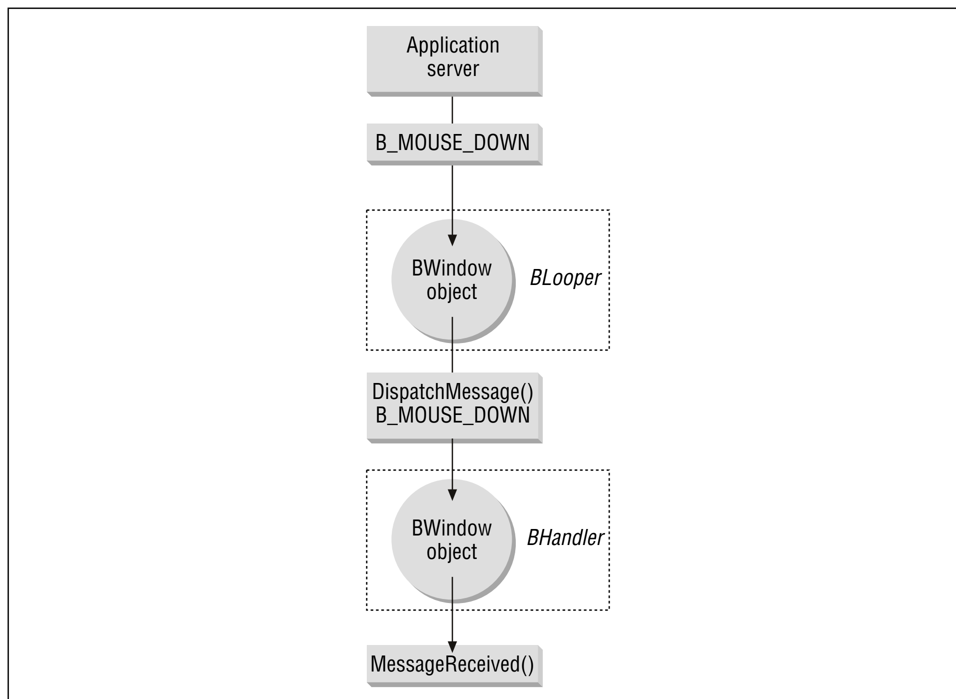


Figure 9-2. A message moves from the Application Server to a view

In Figure 9-2, you see that the window invokes a function named DispatchMessage(). This is a BLooper function that BWindow augments (overrides in order to add window-specific functionality, and then invokes the inherited version as well). DispatchMessage() is responsible for forwarding a system

message to the affected view. While your application can override `DispatchMessage()`, it should seldom need to. Similarly, while `DispatchMessage()` can be invoked directly, it's best to leave the timing of the call to the system. Leave it to the looper object (whether the application or a window) to automatically use this message-forwarding routine as it sees fit. In this example, `DispatchMessage()` will make sure that the `BView` object's version of the hook function `MouseDown()` is invoked.

Chapter 4 provided a variety of examples that demonstrated system message handling, including `B_MOUSE_DOWN` and `B_KEY_DOWN` messages. If you refer back to any of these examples, you'll see that each uses a hook function.

Application-defined message handling and implicitly generated messages

An application-defined message isn't handled by means of a hook function. The very fact that your application defines the message means that no pre-existing hook function could be included in whatever `BHandler`-derived class the recipient object belongs to. Instead, an application-defined message is always dispatched by way of a call to `MessageReceived()`. The looper object that receives the message passes it to a handler object, which uses its version of `MessageReceived()` to carry out the message's action. That leads to the distinction that a system message is usually handled by a hook function (some system-generated messages, such as the standard messages resulting from text edits, need to be handled by a version of `MessageReceived()`), while an application-defined message is always handled by a `MessageReceived()` function.

You've seen several examples of how an application works with application-defined messages—most notably in the chapters that deal with controls and menus (Chapter 6, *Controls and Messages*, and Chapter 7, *Menus*). For instance, a program that implements message handling through a menu item first defines a message constant:

```
#define   MENU_ADV_HELP_MSG     'help'
```

The program then includes this message constant in the creation of a new `BMessage` object—as is done here as part of the process of creating a new `BMenuItem`:

```
menu->AddItem(new BMenuItem("Advanced Help",
                            new BMessage(MENU_ADV_HELP_MSG)));
```

Finally, the message constant appears in a `case` section in the `BWindow` object's `MessageReceived()` function—as in this snippet:

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

```
            case MENU_ADV_HELP_MSG:
               OpenHelpWindow(MENU_ADV_HELP_MSG);
               break;

            // other case sections here

            default:
               BWindow::MessageReceived(message);
         }
      }
```

Like a system message, an application-defined message relies on the BLooper-inherited function DispatchMessage() to transfer the application-defined message from the looper to the handler. Again, your code shouldn't ever have to redefine DispatchMessage() or invoke it directly. As shown in Figure 9-3, in this example the BWindow object is both the looper and handler. The menu item–generated message is placed in the window's message loop, and the window object sends the message to itself and invokes the window's version of MessageReceived() via the BWindow version of DispatchMessage().

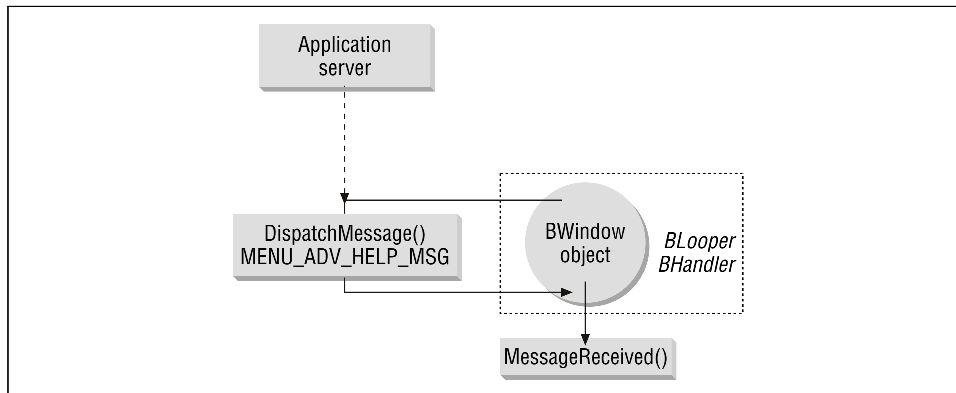


Figure 9-3. A message moves from a window back to that window

While the window generates the message and delivers it to itself, the Application Server may play a role in the act. This is most evident for a message generated by a menu item or control. In each case, the Application Server inserts when data into the message so the application knows at what instant the event (generally a mouse button click) that initiated the message occurred.

Application-defined message handling and explicitly generated messages

A user request, such as menu item selection or control activation, is one way an application-defined message gets generated and `MessageReceived()` gets invoked. In this case, the message is created and passed automatically. You may encounter other instances where it's appropriate for one object in a program to receive information, or take some action, based on circumstances other than a user action. To do that, your program can have an object (such as a window) create a message object, and then have that message posted to a looper object.

As an example, consider a window that needs to pass some information to the application. Perhaps the window is performing some lengthy task, and it wants the application to know when the task is completed. The window could create a `BMessage` object and send it to the application. In a simple case, the arrival of the message might be enough information for the application. However, a message can contain any amount of information, so a more sophisticated example might have the message holding information about the completed task, such as the length of time it took to execute the task.

When `PostMessage()` is called, the specified message is delivered to the looper the function is called upon. You've seen this in all of the example projects to this point. When the user clicks on a window's close button, the window's `QuitRequested()` hook function is invoked. In that function, the application object invokes `PostMessage()`. Here the application object acts as a looper to post the message, then acts as a handler to dispatch the message to its `MessageReceived()` function:

```
bool MyHelloWindow::QuitRequested()
{
   be_app->PostMessage(B_QUIT_REQUESTED);

   return(true);
}
```

A message posted to a looper via a call to `PostMessage()` gets delivered, or dispatched, via the `DispatchMessage()` function. When it comes time to send a message, the sender (the looper object) calls `PostMessage()`. `PostMessage()` in turn calls `DispatchMessage()`. In the above version of `QuitRequested()`, the message posted is a Be-defined message, but that needn't be the case—it could be an application-defined one. In such a case, an object such as a window would create the message using `new` and the `BMessage` constructor (as discussed ahead). If the message was to be delivered to the application, the message could then be posted just as it was in `QuitRequested()`. Figure 9-4 illustrates the process.

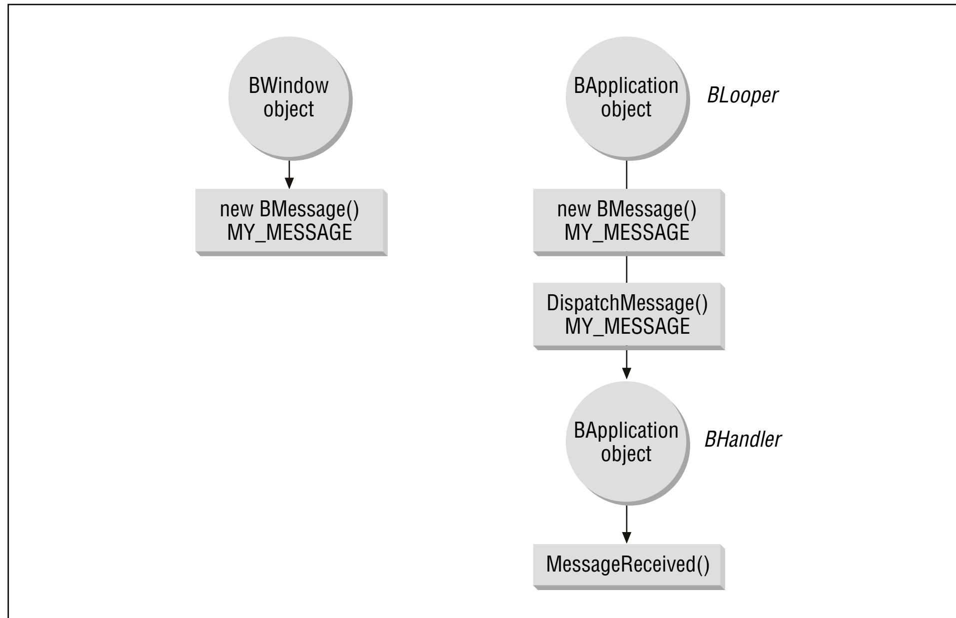


Figure 9-4. A message moves from a window to the application

Working with BMessage Objects

The preceding section served as an introduction to how an application might create a message object and send it to another object. That section was just that—an introduction. Here you'll see the details—and code—for creating, posting, and handling `BMessage` objects.

Creating a message

The `BMessage` constructor has a single parameter—a `uint32` value represents the new message object's command constant. System message command constants always begin with `B_`, as in `B_QUIT_REQUESTED` and `B_MOUSE_DOWN`, so to be quickly recognized as an application-defined message, your application-defined command constants should begin with any other combination of characters. Additionally, each system message's command constant is defined to be a four-character string that consists of only uppercase characters and, optionally, underscore characters. Defining an application-defined message by any other scheme (such as using all lowercase characters) ensures that the message won't be misinterpreted as a system message. Here's an example of the creation of a `BMessage` object:

```
#define    WAGER_MSG    'wger'

BMessage   firstRaceWagerMsg = new BMessage(WAGER_MSG);
```

The `BMessage` constructor sets the `what` data member of the new message object to the value of the command parameter. As you've seen, it's the value of `what` that's used by `MessageReceived()`:

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

      case WAGER_MSG:
         // handle message;

      ...
   }
}
```

A message always has a command constant, and it may include data. Regardless of whether a message holds data, it's posted to a looper, and dispatched to a handler, in the same way. The `firstRaceWagerMsg` consists of nothing more than a command constant, but it is nonetheless a complete message. So before increasing the complexity of message-related discussions by examining how data is added to and extracted from a message object, let's use the simple message to see how a message is posted to a looper and then dispatched to a handler.

Posting and dispatching a message

Once created, a message needs to be placed in the message loop of a looper's thread and then delivered to a handler. The looper is an object of the `BLooper` class or an object of a `BLooper`-derived class, such as the application object or a window object. The handler is an object of the `BHandler` class or an object of a `BHandler`-derived class, such as, again, the application object or a window object (refer back to Figure 9-1 to see the pertinent part of the BeOS API class hierarchy). A call to `PostMessage()` places a message in the queue of the looper whose `PostMessage()` function is called, and optionally specifies the handler to which the message is to be delivered. This `BLooper` member function has the following parameter list:

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

The first parameter, `message`, is the `BMessage` object to post. The second parameter, `handler`, names the target handler—the `BHandler` object to which the message is to be delivered. The `replyHandler`, which is initialized to `NULL`, is of interest only if the target handler object is going to reply to the message (more typically the target handler simply handles the message and doesn't return any type of reply). While the poster of the message and the target of the message don't have to be one and the same, they can be—as shown in this snippet (read the

"Menu Items and Message Dispatching" sidebar for a look at how previous example projects have been doing this):

```
#define    WAGER_MSG    'wger'

BMessage   firstRaceWagerMsg = new BMessage(WAGER_MSG);

theWindow->PostMessage(firstRaceWagerMsg, theWindow);
```

A posted message is placed in the looper's message queue, where it takes its place behind (possibly) other messages in the queue in preparation to be delivered to the target handler object. The looper object continually checks its queue and calls the `BLooper` function `DispatchMessage()` for the next message in the queue. When your posted message becomes the next in the queue, the looper invokes `DispatchMessage()` to pass the message to the target handler. The effect is for the posted message to reach the target handler's `MessageReceived()` function. If that routine has a `case` label that matches the message's `what` data member, the handler acts on the message. Since the above code names a window as both the looper and the target handler, the window must have a `MessageReceived()` function set up to take care of a message of type `WAGER_MSG` (if it doesn't, the program won't fail—the posted message simply isn't acted upon):

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

      case WAGER_MSG:
         // handle message;

      ...
   }
}
```

The `BLooper` class provides another way to call `PostMessage()`—a sort of shorthand method that in many cases saves you the (admittedly simple) step of creating a `BMessage` object. Instead of passing a `BMessage` object as the first `PostMessage()` parameter, simply pass the command constant that represents the type of message to be posted. Here's how a `WAGER_MSG` could be posted:

```
theWindow->PostMessage(WAGER_MSG, theWindow);
```

When a command constant is passed in place of a `BMessage` object, the `PostMessage()` function takes it upon itself to do the work of creating the `BMessage` object and initializing the new object's `what` data member to the value of the passed command constant. This method of invoking `PostMessage()` is acceptable only when the message to be created contains no data (other than the command constant itself). If a posted message object is to include additional data, then `PostMessage()` won't know how to add it to the newly created message

Menu Items and Message Dispatching

The code in this section shows that the poster of the message and the target of the message can be the same object. You've already seen this situation several times when working with menus, though the comparison may not be immediately noticeable. When a new menu item is created and added to a menu, a new BMessage object is created and associated with the new menu item:

```
#define    MENU_OPEN_MSG   'open'

BMenu      *menu;
BMenuItem  *menuItem;

menu = new BMenu("File");
menuItem = new BMenuItem("Open", new BMessage(MENU_OPEN_MSG));
menu->AddItem(menuItem);
```

When the user selects the Open menu item, the MENU_OPEN_MSG message is sent to the message loop of the window that holds the menu item. No call to PostMessage() is needed, as the system implicitly dispatches the message by way of a call to DispatchMessage(). By default, the BMenuItem constructor has made this same window the handler of this message, so the message typically gets dispatched to the MessageReceived() function of the window (though it could end up going to a hook function if the menu item message was a system message such as B_QUIT_REQUESTED):

```
void MyHelloWindow::MessageReceived(BMessage* message)
{

   switch(message->what)
   {
      case MENU_OPEN_MSG:
         // open a file;
         break;

      ...
   }
}
```

While the system takes care of menu handling without your code needing to include an explicit call to PostMessage(), the effect is the same.

While the target handler for a menu item-associated message is the window that holds the menu, you can change this default condition. A BMenuItem is derived from the BInvoker class (a simple class that creates objects that can be invoked to send a message to a target), so you can call the BInvoker function SetTarget() to make the change. After the following call, an Open menu item selection will send a MENU_OPEN_MSG to the application's version of MessageReceived() rather than to the window's version of this function:

```
menuItem->SetTarget(be_app);
```

object. Working with more complex messages—messages that hold data—is the subject of the next section.

Message-posting example project

The WindowMessage1 project demonstrates one way to stagger windows. Moving windows about the screen is a trivial task that doesn't necessarily require the use of messages. That's all the better reason to choose this chore for a message-related example—it lets me concentrate on working with messages rather than on solving a difficult problem!

A new WindowMessage1 window has a File menu that consists of a single item: a New item that creates a new window. The program begins by opening a single window near the upper-left corner of the screen. When the user chooses New from the File menu, all open windows jump 30 pixels down and 30 pixels to the right of their current locations. Thus, if a user chooses New a number of times (without moving the windows as they're created), the windows end up staggered (as shown in Figure 9-5) rather than piled up like cards in a deck.

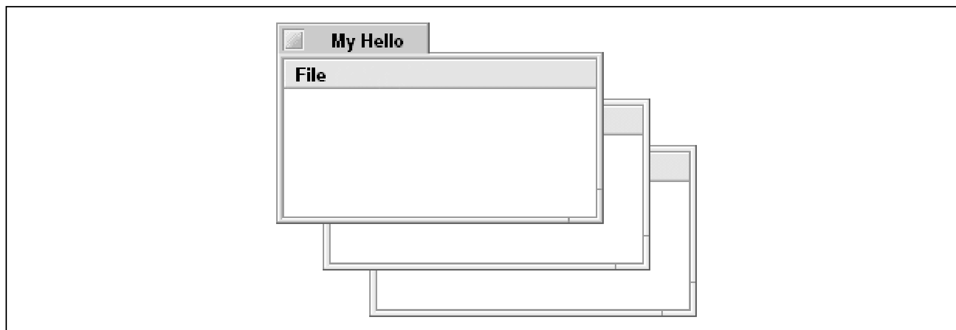


Figure 9-5. The staggered windows of the WindowMessage1 program

The WindowMessage1 project defines two application-defined message constants. A message of type `MENU_NEW_WINDOW_MSG` is implicitly generated whenever the user selects the New menu item. A message of type `MOVE_WINDOWS_MSG` is explicitly posted as a part of carrying out a New menu item selection:

```
#define   MENU_NEW_WINDOW_MSG    'nwwd'
#define   MOVE_WINDOWS_MSG       'anwd'
```

The `MyHelloWindow` constructor adds a menubar with the single menu to a new window. The `AddItem()` function that adds the menu item is responsible for associating a `BMessage` of type `MENU_NEW_WINDOW_MSG` with the menu item:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.OffsetTo(B_ORIGIN);
```

```
    frame.top += MENU_BAR_HEIGHT + 1.0;

    fMyView = new MyDrawView(frame, "MyDrawView");
    AddChild(fMyView);

    BMenu  *menu;
    BRect  menuBarRect;

    menuBarRect.Set(0.0, 0.0, 10000.0, MENU_BAR_HEIGHT);
    fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
    AddChild(fMenuBar);

    menu = new BMenu("File");
    fMenuBar->AddItem(menu);
    menu->AddItem(new BMenuItem("New Window",
                  new BMessage(MENU_NEW_WINDOW_MSG)));

    Show();
}
```

Each time the New menu item is selected, a copy of the menu item's message is created. A message object of this type consists of nothing more than the message constant `MENU_NEW_WINDOW_MSG`. The new message object is sent to the message's handler. By default, this handler is the window the menu item appears in. So it is the `MessageReceived()` function of the `MyHelloWindow` class that becomes responsible for handling the message generated by a New menu item selection:

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

      case MENU_NEW_WINDOW_MSG:
         be_app->PostMessage(MOVE_WINDOWS_MSG, be_app);
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

If I wanted the New menu item to simply create a new `MyHelloWindow`, I could do that with just a few lines of code. But besides creating a new window, the handling of this menu item choice might affect a number of existing windows. Keeping track of the windows that are currently open is the responsibility of the `BApplication` object, so I create a `MOVE_WINDOWS_MSG` and pass it to the application as a means of signaling the application to offset each open window. Including the message constant `MOVE_WINDOWS_MSG` in the call to `PostMessage()` tells this routine to create a new message object and assign the message constant `MOVE_WINDOWS_MSG` to the new message object's `what` data member. Since my

messages of type MOVE_WINDOWS_MSG won't contain any additional data, this message-creation shortcut is appropriate. The new message object is then posted to the application object (per the second PostMessage() parameter).

The MyHelloApplication class is to handle application-defined messages, so the class now needs to override MessageReceived(). Since the program allows multiple windows and doesn't keep constant track of which window is active, the MyHelloWindow data member fMyWindow that appears in similar examples has been eliminated:

```
class MyHelloApplication : public BApplication {

   public:
                        MyHelloApplication();
      virtual void      MessageReceived(BMessage* message);
};
```

The MyHelloApplication version of MessageReceived() uses the Chapter 4 method of repeatedly calling the BApplication function WindowAt() to gain a reference to each currently open window. Once found, a window is moved by invoking the BWindow function MoveBy(). After all existing windows have been moved, a new window is opened near the upper-left corner of the screen.

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

      case MOVE_WINDOWS_MSG:
         BWindow  *oldWindow;
         int32    i = 0;

         while (oldWindow = WindowAt(i++)) {
            oldWindow->MoveBy(30.0, 30.0);
         }

         BRect          theRect;
         MyHelloWindow  *newWindow;

         theRect.Set(20.0, 30.0, 220.0, 130.0);
         newWindow = new MyHelloWindow(theRect);
         break;

      default:
         inherited::MessageReceived(message);
         break;
   }
}
```

The BApplication function WindowAt() returns a BWindow object—so that's what I've declared oldWindow to be. The only action I take with the returned window is to call the BWindow function MoveBy(). If I needed to perform some

`MyHelloWindow`-specific action on the window (for instance, if the `MyHelloWindow` class defined a member function that needed to be invoked), then I'd first need to typecast `oldWindow` to a `MyHelloWindow` object.

Adding and retrieving message data

A number of `BMessage` member functions make it possible to easily add information to any application-defined message object. The prototypes for several of these routines are listed here:

```
status_t AddBool(const char  *name,
                 bool        aBool)

status_t AddInt32(const char  *name,
                  int32       anInt32)

status_t AddFloat(const char  *name,
                  float       aFloat)

status_t AddRect(const char  *name,
                 BRect       rect)

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

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

To add data to a message, create the message object and then invoke the `BMessage` function suitable to the type of data to add to the message object. The following snippet adds a pair of numbers, each stored as a 32-bit integer, to a message:

```
#define    HI_LOW_SCORE_MSG   'hilo'

BMessage  *currentScoreMsg = new BMessage(HI_LO_SCORE_MSG);
int32     highScore = 96;
int32     lowScore  = 71;

currentScoreMsg->AddInt32("High", highScore);
currentScoreMsg->AddInt32("Low", lowScore);
```

After the above code executes, a new message object exists—one that is referenced by the variable `currentScoreMsg`. This message has a `what` data member value of `HI_LO_SCORE_MSG`, and holds integers with values of 96 and 71.

For each `Add` function, the `BMessage` class defines a `Find` function. Each `Find` function is used to extract one piece of information from a message:

```
status_t  FindBool(const char *name,
                   bool       *value) const;
```

```
status_t  FindInt32(const char *name,
                    int32      *val) const;

status_t  FindFloat(const char *name,
                    float      *f) const;

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

status_t  FindString(const char *name,
                     const char **str) const;

status_t  FindPointer(const char *name,
                      void       **ptr) const;
```

To make use of data in a message, the originating object creates the message, invokes `Add` functions to add the data, and posts the message using `PostMessage()`. The receiving object invokes `Find` functions to extract any or all of the message's data from the object that receives the message.

Data added to a message always has both a name and a type. These traits alone are usually enough to extract the data—it's not your program's responsibility to keep track of data ordering in a message object (the exception being arrays, which are covered just ahead). To access the two integers stored in the previous snippet's `currentScoreMsg` message object, use this code:

```
int32  highestValue;
int32  lowestValue;

currentScoreMsg->FindInt32("High", &highestValue);
currentScoreMsg->FindInt32("Low", &lowestValue);
```

It's worthwhile to note that when adding data to a message, you can use the same name and datatype for more than one piece of information. For instance, two high score values could be saved in one message object as follows:

```
currentScoreMsg->AddInt32("High", 98);
currentScoreMsg->AddInt32("High", 96);
```

In such a situation, an array of the appropriate datatype (32-bit integers in this example) is set up and the values are inserted into the array in the order they are added to the message. As expected, array element indices begin at 0. There is a second version of each `Find` routine, one that has an index parameter for finding a piece of information that is a part of an array. For instance, the `FindInt32()` function used for accessing an array element looks like this:

```
status_t  FindInt32(const char *name,
                    int32      index,
                    int32      *val) const;
```

To access an array element, include the index argument. Here the value of 96 (the second element, with an index of 1) is being retrieved from the `currentScoreMsg` message:

```
int32  secondHighValue;

currentScoreMsg->FindInt32("High", 1, &secondHighValue);
```

Make sure to check out the `BMessage` class description in the Application Kit chapter of the Be Book. There you'll find descriptions for other `Add` and `Find` routines, such as `AddInt16()` and `FindPoint()`. You'll also see the other variants of each of the `Add` and `Find` routines I've listed. The Be Book also discusses the universal, or generic, `AddData()` member function. You can optionally use this routine in place of any of the type-specific functions (such as `AddInt32()` or `AddFloat()`) or for adding data of an application-defined type to a message object.

Message data example project

The WindowMessage2 project does the same thing as the WindowMessage1 project—it offsets all open windows when a new window is opened. Like WindowMessage1, this latest project uses messages to carry out its task. Let's look at the different approach used by the two projects.

Recall that when the WindowMessage1 program opened a new window, the active window created a single message and sent it to the application object's `MessageReceived()` function. It was then the responsibility of the application object to locate and move each window. The application did that by looping through the window list and calling `MoveBy()` for each window it encountered.

In the WindowMessage2 program, the active window's `MessageReceived()` function cycles through the window list. When a window is encountered, a reference to it is stored as data in a message, and that message is posted to the application. When the application object's `MessageReceived()` function gets the message, it retrieves the window reference and moves that one window. Thus the window that holds the selected New menu item may generate numerous messages (one for each window that's already open). The WindowMessage1 project may have acted a little more efficiently, but WindowMessage2 gives me the opportunity to post a slew of messages! It also gives me an excuse to store some data in each message—something the WindowMessage1 project didn't do.

WindowMessage2 defines the same two application-defined messages as the WindowMessage1 project—a `MENU_NEW_WINDOW_MSG` issued by a selection of the New menu item, and a `MOVE_WINDOWS_MSG` created by the window and sent to the application. This latest version of the `MyHelloWindow` constructor is identical to the version in the WindowMessage1 project—refer back to that example to see

the listing. The MyHelloWindow version of MessageReceived(), however, is different. Instead of simply creating a new MOVE_WINDOWS_MSG and sending it to the application, this function now repeatedly calls the BApplication function WindowAt(). For each open window, the loop creates a new message, adds a window reference to the message, and posts the message to the application:

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

      case MENU_NEW_WINDOW_MSG:

         BRect          theRect;
         MyHelloWindow  *newWindow;
         BWindow        *oldWindow;
         int32          i = 0;
         BMessage       *newWindowMsg;

         while (oldWindow = be_app->WindowAt(i++)) {
            newWindowMsg = new BMessage(MOVE_WINDOWS_MSG);
            newWindowMsg->AddPointer("Old Window", oldWindow);
            be_app->PostMessage(newWindowMsg, be_app);
         }

         theRect.Set(20.0, 30.0, 220.0, 130.0);
         newWindow = new MyHelloWindow(theRect);
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```

Each posted MOVE_WINDOWS_MSG message has the application as the designated handler. When a message reaches the application object, that object's MessageReceived() function calls FindPointer() to access the window of interest. The BMessage function name (FindPointer()), along with the data name ("Old Window"), indicates that the message object data should be searched for a pointer stored under the name "Old Window." Of course, in this example, that one piece of information is the only data stored in a MOVE_WINDOWS_MSG message, but the technique applies to messages of any size. A window object is a pointer, so the returned value can be used as is—a call to the BWindow function MoveBy() is all that's needed to relocate the window:

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

      case MOVE_WINDOWS_MSG:
         BWindow  *theWindow;
```

```
            message->FindPointer("Old Window", &theWindow);
            theWindow->MoveBy(30.0, 30.0);
            break;
    }
}
```

If you enable the debugger and run the program, you might be able to see multithreading in action. If you set a breakpoint in the `MyHelloApplication` version of `MessageReceived()`, you'll note that, as expected, the function gets called once for each already open window. You may be surprised to see the new window open before the last of the already opened windows is moved. With several windows open, a number of messages are posted to the application. One by one the application pulls these messages from its queue and handles each by moving one window. While that's going on, the code that creates the new window may very well execute.

A second message data example project

The previous two projects both relied on the user making a menu selection to trigger the posting of a message to the application object—it was a menu item-generated message handled in a window's `MessageReceived()` function that in turn created another message. While it may in fact be a menu item selection or other user action that causes your program to create still another message, this doesn't have to be the case. The stimulus may be an event unrelated to any direct action by the user that causes your program to create and post a message. Here, in the AlertMessage project, the launching of an application may result in that program creating a message.

All Be applications can be launched by either double-clicking on the program's icon or by typing the program's name from the command line. Like any of the examples in this book, the AlertMessage program can be launched by opening a terminal window: run the Terminal application from the Tracker's app menu, move to the directory that holds the AlertMessage program, and type the program name. Regardless of whether AlertMessage launches from the desktop or from the command line, a single window opens. If the program starts up from the command line, however, the option exists to choose the number of windows that will automatically open. To take advantage of this option, the user need simply follow the program name with a space and the desired number of windows. Figure 9-6 shows how I worked my way into the folder that holds my copy of AlertMessage, and how I then indicated that the program should start with three windows open.

The AlertMessage program allows at most five windows to be opened at application launch. If you launch AlertMessage from the command line and enter a value

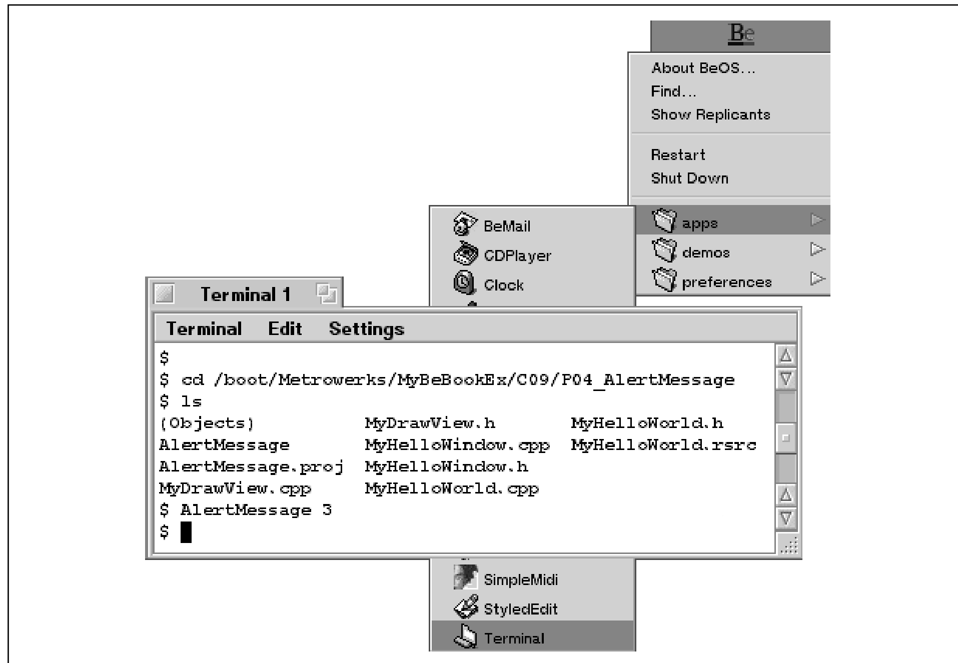


Figure 9-6. Launching the AlertMessage program from the command line

greater than 5, the program will execute, but only five windows will open. In such a case, the program gives the user an indication of what happened by displaying an alert like the one shown in Figure 9-7.

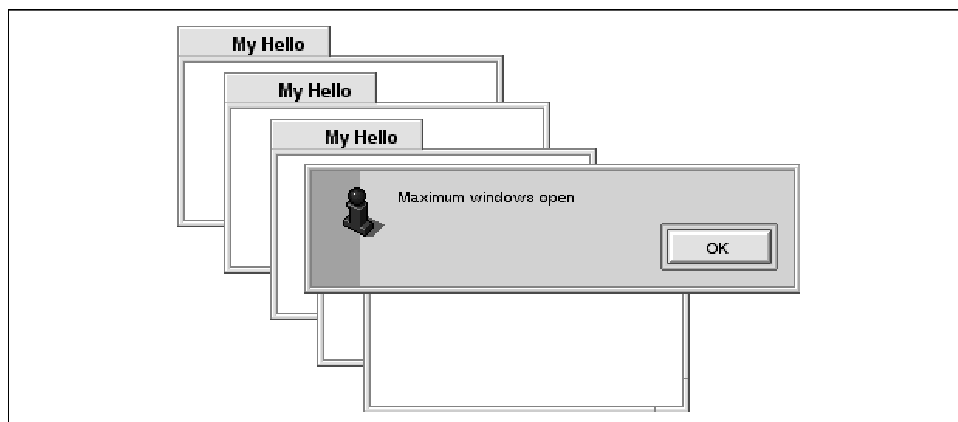


Figure 9-7. The windows of the AlertMessage program

The alert in Figure 9-7 is displayed thanks to a message the application posts to itself. When the AlertMessage program launches from the command line, a check

is made to see if the user-specified window value is greater than 5. If it is, an application-defined WINDOW_MAX_MSG is created:

```
#define   WINDOW_MAX_MSG  'wdmx'

BMessage  *maxWindowsMsg = new BMessage(WINDOW_MAX_MSG);
```

The WindowMessage2 project demonstrated how to add a pointer to a message. Here you see how to add a Boolean value and a string. The means are BMessage Add functions—data of other types is added in a similar manner:

```
bool        beepOnce = true;
const char  *alertString = "Maximum windows open";

maxWindowsMsg->AddBool("Beep", beepOnce);
maxWindowsMsg->AddString("Alert String", alertString);
```

The beepOnce variable will be used to specify whether or not a beep should accompany the display of the alert. The alertString holds the text to be displayed. Once created and set up, the message is posted to the application:

```
be_app->PostMessage(maxWindowsMsg, be_app);
```

PostMessage() specifies that the application be the message handler, so it's the application object's version of MessageReceived() that gets this WINDOW_MAX_MSG message:

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

      case WINDOW_MAX_MSG:
         bool        beepOnce;
         const char  *alertString;
         BAlert      *alert;
         long        result;

         beepOnce = message->FindBool("Beep");
         alertString = message->FindString("Alert String");

         if (beepOnce)
            beep();

         alert = new BAlert("MaxWindowAlert", alertString, "OK");
         result = alert->Go();

         break;
   }
}
```

MessageReceived() handles the message by first accessing its data. If beepOnce is true, a system beep is sounded. The text of the string alertString is used as

the text displayed in the alert (refer to Chapter 4 for information about alerts and the BAlert class).

AlertMessage is this book's first example that uses a command-line argument in the launching of a program, so a little extra explanation on how a program receives and responds to such input is in order.

Command-line arguments

An application message (a system message that affects the application itself rather than one particular window) is both received and handled by a program's BApplication object. A B_ARGV_RECEIVED message is such an application message. When a program is launched with one or more arguments from the command line, a B_ARGV_RECEIVED message is delivered to the application. Unlike most application messages, a B_ARGV_RECEIVED message holds data. In particular, it holds two pieces of data. The first, argc, is an integer that specifies how many arguments the program receives. The second, argv, is an array that holds the actual arguments. Because the program name itself is considered an argument, the value of argc will be one greater than the number of arguments the user typed. The array argv will thus always have as its first element the string that is the name of the program. Consider the case of the user launching the just-discussed AlertMessage program as follows:

```
$ AlertMessage 4
```

Here the value of argc will be 2. The string in argv[0] will be "AlertMessage" prefaced with the pathname, while the string in argv[1] will be "4". Because all arguments are stored as strings, you'll need to convert strings to numbers as necessary. Here I'm using the atoi() string-to-integer function from the standard C++ library to convert the above user-entered argument from the string "4" to the integer 4:

```
uint32 userNumWindows = atoi(argv[1]);
```

The fact that the program's path is included as part of the program name in the string argv[0] is noteworthy if you're interested in determining the program's name (remember—from the desktop the user is free to change the name of your application!). If the user is keeping the AlertMessage program in the computer's root directory, and launches it from the command line while in a subdirectory, the value of argv[0] will be "/root/AlertMessage". If your program is to derive its own name from argv[0], it should strip off leading characters up to and including the final "/" character.

When a program receives a B_ARGV_RECEIVED message, it dispatches it to its ArgvReceived() function. I've yet to discuss this BApplication member function because up to this point none of my example projects have had a provision for handling user input at application launch. The AlertMessage program does accept such input, so its application object needs to override this routine:

```
class MyHelloApplication : public BApplication {

   public:
                        MyHelloApplication();
      virtual void      MessageReceived(BMessage* message);
      virtual void      ArgvReceived(int32 argc, char **argv);
};
```

The program relies on a number of constants in opening each window. WINDOW_WIDTH and WINDOW_HEIGHT define the size of each window. WINDOW_1_LEFT and WINDOW_1_TOP establish the screen position of the first window. The two offset constants establish how each subsequent window is to be staggered from the previously opened window:

```
#define    WINDOW_WIDTH        200.0
#define    WINDOW_HEIGHT       100.0
#define    WINDOW_1_LEFT        20.0
#define    WINDOW_1_TOP         30.0
#define    WINDOW_H_OFFSET      30.0
#define    WINDOW_V_OFFSET      30.0
```

Regardless of whether the user launches AlertMessage from the desktop or from the shell, one window is always opened. The AlertMessage version of ArgvReceived() looks at the value the user typed in following the program name and uses that number to determine how many additional windows to open. ArgvReceived() thus opens the user-entered value of windows, less one. Before doing that, however, the user's value is checked to verify that it doesn't exceed 5—the maximum number of windows AlertMessage allows. If the value is greater than 5, ArgvReceived() creates a WINDOW_MAX_MSG, supplies this message with some data, and posts the message. After posting the message, the number of windows to open is set to the maximum of 5:

```
void MyHelloApplication::ArgvReceived(int32 argc, char **argv)
{
   uint32 userNumWindows = atoi(argv[1]);

   if (userNumWindows > 5) {
      bool        beepOnce = true;
      const char  *alertString = "Maximum windows open";
      BMessage    *maxWindowsMsg = new BMessage(WINDOW_MAX_MSG);

      maxWindowsMsg->AddBool("Beep", beepOnce);
      maxWindowsMsg->AddString("AlertString", alertString);
      be_app->PostMessage(maxWindowsMsg, be_app);
```

```
      userNumWindows = 5;
   }

   BRect          aRect;
   float          left = WINDOW_1_LEFT + WINDOW_H_OFFSET;
   float          right = left + WINDOW_WIDTH;
   float          top = WINDOW_1_TOP + WINDOW_V_OFFSET;
   float          bottom = top + WINDOW_HEIGHT;
   MyHelloWindow  *theWindow;
   uint32         i;

   for (i = 2; i <= userNumWindows; i++) {
      aRect.Set(left, top, right, bottom);
      theWindow = new MyHelloWindow(aRect);
      left   += WINDOW_H_OFFSET;
      right  += WINDOW_H_OFFSET;
      top    += WINDOW_V_OFFSET;
      bottom += WINDOW_V_OFFSET;
   }

}
```

As mentioned in the description of the AlertMessage project, a posted `WINDOW_MAX_MSG` is handled by the application object's `MessageReceived()` function. There the message data is accessed and an alert posted.

Adding data of any type to a message

The `BMessage` `Add` routines, such as `AddBool()` and `AddString()`, serve as a sort of shorthand notation for the more generic `BMessage` function `AddData()`. `AddData()` can be used to add data of any type to a message. Thus, `AddData()` can be used to add data of an application-defined type, or data of any of the types that can be added using a specific `Add` function. Here's the declaration for `AddData()`:

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

The `name` and `data` parameters serve the same purposes as their counterparts in the other `Add` routines—`name` serves as an identifier that's used when later accessing the data through the use of a `Find` function, while `data` holds the data itself. Unlike most `Add` functions, though, in `AddData()` the data parameter is a pointer to the data rather than the data itself.

Because `AddData()` can accept data of any type, you need to specify both the kind of data to add and the size, in bytes, of data that is to be added. Use the appropriate Be-defined type constant for the `type` parameter. The third column of

Table 9-1 lists these constants for commonly used Add routines—make sure to turn to the BMessage class description in the Application Kit chapter of the Be Book for more Add routines and corresponding type constants.

Table 9-1. BMessage Add Functions and Associated Be-Defined Type Constants

Add Member Function	Datatype Added	Datatype Constant
AddBool()	bool	B_BOOL_TYPE
AddInt32()	int32/uint32	B_INT32_TYPE
AddFloat()	float	B_FLOAT_TYPE
AddRect()	BRect object	B_RECT_TYPE
AddString()	Character string	B_STRING_TYPE
AddPointer()	Any type of pointer	B_POINTER_TYPE

The fixedSize and numItems parameters are useful only when adding data that is to become the first item in a new array (recall that adding data with the same name parameter automatically results in the data being stored in an array). Both these parameters help AddData() work with data more efficiently. If the array is to hold items that are identical in size (such as an array of integers), pass true for fixedSize. If you have an idea of how many items will eventually be in the array, pass that value as numItems. An inaccurate value for numItems just diminishes slightly the efficiency with which AddData() utilizes memory—it won't cause the routine to fail.

The just-described AlertMessage example project created a message object and added a bool value and a string to that message:

```
bool        beepOnce = true;
const char  *alertString = "Maximum windows open";
BMessage    *maxWindowsMsg = new BMessage(WINDOW_MAX_MSG);

maxWindowsMsg->AddBool("Beep", beepOnce);
maxWindowsMsg->AddString("AlertString", alertString);
```

Because AddBool() and AddString() are simply data-type "tuned" versions of AddData(), I could have added the data using two calls to AddData(). To do that, I'd replace the last two lines in the above snippet with this code:

```
maxWindowsMsg->AddData("Beep", B_BOOL_TYPE, &beepOnce, sizeof(bool));

maxWindowsMsg->AddData("AlertString", B_STRING_TYPE,
                       alertString, strlen(alertString));
```

AddData() accepts a pointer to the data to add, so the bool variable beepOnce is now prefaced with the "address of" operator. The string alertString is already in the form of a pointer (char *), so it can be passed as it was for AddString(). As shown in the above snippet, if you're adding a bool value, pass B_BOOL_TYPE

as the second `AddData()` parameter. You generally determine the size of the data to add through the standard library function `sizeof()` or, as in the case of a string, the `strlen()` routine.

Like the other `Add` functions, `AddData()` has a companion `Find` function—`FindData()`. Here's that routine's prototype:

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

`FindData()` searches a message for data that is of the type specified by the `type` parameter and that is stored under the name specified by the `name` parameter. When it finds it, it stores a pointer to it in the `data` parameter, and returns the number of bytes the data consists of in the `numBytes` parameter. An example of the use of `FindData()` appears next.

Data, messages, and the clipboard

Earlier in this chapter, I discussed the clipboard, but held off on presenting an example project. Here's why: the clipboard holds its data in a `BMessage` object, and the details of accessing message data weren't revealed until well past this chapter's first mention of the clipboard. Now that you've been introduced to the clipboard and have a background in `BMessage` basics, working with the clipboard will seem simple.

The clipboard is represented by a `BClipboard` object that includes a data member that is a `BMessage` object. Items on the clipboard are all stored as separate data in this single clipboard message object. This is generally of little importance to you because most program interaction with the clipboard is transparent. For instance, when you set up a Paste menu item, the `B_PASTE` message is associated with the menu item, and your work to support pasting is finished. Here's the pertinent code:

```
menu->AddItem(menuItem = new BMenuItem("Paste", new BMessage(B_PASTE), 'V'));
menuItem->SetTarget(NULL, this);
```

If your program has cause to add data to, or retrieve data from, the clipboard by means other than the standard Be-defined messages, it can. Only then is it important to understand how to interact with the clipboard's data.

Because the clipboard object can be accessed from any number of objects (belonging to your application or to any other running application), the potential for clipboard data to be accessed by two threads at the same time exists. Clipboard access provides a specific example of locking and unlocking an object, the topic discussed in this chapter's "Messaging" section. Before working with the clipboard, call the `BClipboard` function `Lock()` to prevent other access by other

threads (if the clipboard is in use by another thread when your thread calls `Lock()`, your thread will wait until clipboard access becomes available). When finished, open up clipboard access by other threads by calling the `BClipboard` function `Unlock()`:

```
be_clipboard->Lock();

// access clipboard data here

be_clipboard->Unlock();
```

The global clipboard is typically used to hold a single item—the most recent item copied by the user. Adding a new item generally overwrites the current item (which could be any manner of data, including that copied from a different application). If your thread is adding data to the clipboard, it should first clear out the existing clipboard contents. The `BClipboard` function `Clear()` does that. After adding its own data, your thread needs to call the `BClipboard` function `Commit()` to confirm that this indeed is the action to perform. So while the above snippet works fine for retrieving clipboard data, it should be expanded a bit for adding data to the clipboard:

```
be_clipboard->Lock();
be_clipboard->Clear();

// add clipboard data here

be_clipboard->Commit();
be_clipboard->Unlock();
```

To actually access the clipboard's data, call the `BClipboard` function `Data()`. This function obtains a `BMessage` object that you use to reference the clipboard's data. This next snippet shows that here you don't use `new` to create the message—the `Data()` function returns the clipboard's data-holding message:

```
BMessage  *clipMessage;

clipMessage = be_clipboard->Data();
```

At this point, clipboard data can be accessed using `BMessage` functions such as `AddData()` and `FindData()`. Here the text "Testing123" replaces whatever currently resides on the clipboard:

```
const char *theString = "Testing123";

be_clipboard->Lock();
be_clipboard->Clear();

BMessage  *clipMessage;

clipMessage = be_clipboard->Data();
```

```
clipMessage->AddData("text/plain", B_MIME_TYPE, theString,
strlen(theString));

be_clipboard->Commit();
be_clipboard->Unlock();
```

The clipboard exists for data exchange—including interapplication exchange. So you might not be surprised to see that MIME (Multipurpose Internet Mail Extensions) may be involved in clipboard usage. When you pass `AddData()` a `type` parameter of `B_MIME_TYPE`, you're specifying that the data to be added is of the MIME main type and subtype listed in the `name` parameter. For adding text, use `text` as the main type and `plain` as the subtype—resulting in "text/plain" as the first `AddData()` parameter.

To retrieve data from the clipboard, use the `BMessage` function `FindData()`. This snippet brings whatever text is currently on the clipboard into a string variable named `clipString`. It also returns the number of bytes of returned text in the variable `numBytes`:

```
be_clipboard->Lock();

BMessage    *clipMessage;
const char  *clipString;
ssize_t     numBytes;

clipMessage = be_clipboard->Data();
clipMessage->FindData("text/plain", B_MIME_TYPE, &clipString, &numBytes);

be_clipboard->Unlock();
```

Clipboard example project

The ClipboardMessage project provides a simple example of adding text to the clipboard. This project adds just a few changes to the Chapter 8 project TextViewEdit. Recall that TextViewEdit displayed a window that included a single menu with a Test item that sounds the system beep, and the four standard text-editing items. The window also included one `BTextView` object. Figure 9-8 shows that for the new ClipboardMessage project a new Add String menu item has been added. Choosing Add String clears the clipboard and places the text "Testing123" on it. Subsequent pastes (whether performed by choosing the Paste menu item or by pressing Command-v) place this string at the insertion point in the window's text view object.

The `MyHelloWindow` constructor associates a new application-defined message constant, `ADD_STR_MSG`, with the new Add String menu item. Except for the new `AddItem()` line before the call to `Show()`, the `MyHelloWindow` constructor is

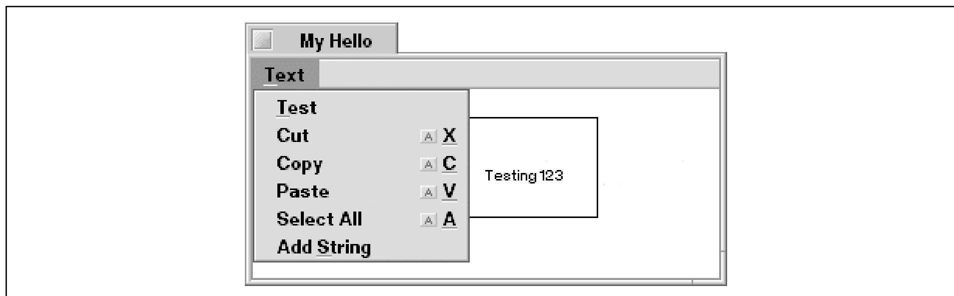


Figure 9-8. The window of the ClipboardMessage program

identical to the version used in the Chapter 8 TextViewEdit project on which this new project is based, so only a part of the constructor is shown here:

```
#define   ADD_STR_MSG      'adst'

MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   ...
   menu->AddItem(menuItem = new BMenuItem("Select All",
                                        new BMessage(B_SELECT_ALL), 'A'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Add String",
                                        new BMessage(ADD_STR_MSG)));

   Show();
}
```

The `MessageReceived()` function holds the new clipboard code. Selecting Add String locks and clears the clipboard, accesses the clipboard data-holding message, adds a string to the clipboard, commits that addition, then unlocks the clipboard for use by other threads. Here's `MessageReceived()` in its entirety (recall that the text-editing commands `B_CUT`, `B_COPY`, `B_PASTE`, and `B_SELECT_ALL` are standard messages that are automatically handled by the system):

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case ADD_STR_MSG:

         const char *theString = "Testing123";

         be_clipboard->Lock();
         be_clipboard->Clear();

         BMessage  *clipMessage;

         clipMessage = be_clipboard->Data();
```

```
         clipMessage->AddData("text/plain", B_MIME_TYPE, theString,
                                   strlen(theString));

         be_clipboard->Commit();
         be_clipboard->Unlock();

         break;

      case TEST_MSG:
         beep();
         break;

      default:
         BWindow::MessageReceived(message);
   }
}
```