Table of Contents

In this chapter:
- *Features of the BeOS*
- *Structure of the BeOS*
- *Software Kits and Their Classes*
- *BeOS Programming Fundamentals*
- *BeOS Programming Environment*

1 BeOS Programming Overview

A few years back, the Macintosh operating system was considered innovative and fun. Now many view it as dated and badly in need of a rewrite rather than a simple upgrade. Windows 95 is the most popular operating system in the world—but this operating system is in many ways a copy of the Mac OS, less the Mac's character. Many programmers and computer enthusiasts enjoy the command-line interface power of Unix—but Unix isn't nearly intuitive enough for the average end user. What users really want is an operating system that has an easy-to-use graphical user interface, takes advantage of the power of today's fast microprocessor chips, and is unencumbered with the burdens of backward compatibility. Enter Be, Inc., and the BeOS—the Be operating system.

In this introductory chapter, you'll learn about the features of the BeOS from a programmer's perspective. In particular, you'll read about the terminology relating to the Be operating system. You'll also get an overview of the layout of the application programming interface, or API, that you'll be using to aid you in piecing together your programs. After the overview, you'll look at some of the fundamentals of writing applications for the BeOS. No attempt will be made to supply you with a full understanding of the concepts, techniques, and tricks of programming for this operating system—you've got the whole rest of the book for that! Instead, in this chapter I'll just give you a feel for what it's like to write a program for the BeOS. Finally, this chapter concludes with a first look at Metrowerks CodeWarrior—the integrated development environment you'll be using to develop your own applications that run on the BeOS.

Features of the BeOS

With any new technology comes a plethora of buzzwords. This marketing hype is especially true in the computer industry—innovative software and hardware seem to appear almost daily, and each company needs some way to ensure that the public views their product as the best. Unsurprisingly, the BeOS is also accompanied by a number of buzzwords—multithreaded, multiprocessor support, and preemptive multitasking being a few. What may be surprising is that this nomenclature, when applied to BeOS, isn't just hype—these phrases really do define this exciting operating system!

Multithreaded

A *thread* is a path of execution—a part of a program that acts independently from other parts of the program, yet is still capable of sharing data with the rest of program. An OS that is *multithreaded* allows a single program to be divided into several threads, with each thread carrying out its own task. The processor devotes a small amount of time first to one thread and then to another, repeating this cycle for as long as it takes to carry out whatever task each thread is to perform. This parallel processing allows the end user to carry out one action while another is taking place. Multithreading doesn't come without a price—though fortunately in the BeOS this price is a rather small one. A program that creates multiple threads needs to be able to protect its data against simultaneous access from different threads. The technique of locking information when it is being accessed is one that is relatively easy to implement in BeOS programs.

The BeOS is a multithreaded operating system—and a very efficient one. While programmers can explicitly create threads, much of the work of handling threads is taken care of behind the scenes by the operating system itself. For instance, when a window is created in a program, the BeOS creates and maintains a separate thread for that one window.

Multiprocessor Support

An operating system that uses multithreading, designed so that threads can be sent to different processors, is said to use *symmetric multiprocessing*, or SMP. On an SMP system, unrelated threads can be sent to different processors. For instance, a program could send a thread that is to carry out a complex calculation to one processor and, at the same time, send a thread that is to be used to transfer a file over a network to a second processor. Contrasting with symmetric multiprocessing (SMP) is *asymmetric multiprocessing*, or AMP. A system that uses AMP sends a thread to one processor (deemed the master processor) which in turn parcels out subtasks to the other processor or processors (called the slave processor or processors).

The BeOS can run on single-processor systems (such as single-processor Power Macintosh computers), but it is designed to take full advantage of machines that have more than one processor—it uses symmetric multiprocessing. When a Be program runs on a multiprocessor machine, the program can send threads to each processor for true parallel processing. Best of all, the programmer doesn't need to be concerned about how to evenly divide the work load. The Be operating system is responsible for distributing tasks among whatever number of processors are on the host machine—whether that be one, two, four, or more CPUs.

The capability to run different threads on different processors, coupled with the system's ability to assign threads to processors based on the current load on each processor, makes for a system with very high performance.

Preemptive Multitasking

An operating system that utilizes *multitasking* is one that allows more than one program to run simultaneously. If that operating system has *cooperative multitasking*, it's up to each running program to yield control of system resources to allow the other running applications to perform their chores. In other words, programs must cooperate. In a cooperative multitasking environment, programs can be written such that they don't cooperate graciously—or even such that they don't cooperate at all. A better method of implementing multitasking is for an operating system to employ preemptive multitasking. In a *preemptive multitasking* environment the operating system can, and does, preempt currently running applications. With preemptive multitasking, the burden of passing control from one program to another falls on the operating system rather than on running applications. The advantage is that no one program can grab and retain control of system resources.

If you haven't already guessed, the BeOS has preemptive multitasking. The BeOS microkernel (a low-level task manager discussed later in this chapter) is responsible for scheduling tasks according to priority levels. All tasks are allowed use of a processor for only a very short time—three-thousandths of a second. If a program doesn't completely execute a task in one such *time-slice*, it will pick up where it left off the next time it regains use of a processor.

Protected Memory

When a program launches, the operating system reserves an area of RAM and loads a copy of that program's code into this memory. This area of memory is then devoted to this application—and to this application only. While a program running under any operating system doesn't intentionally write to memory locations reserved for use by other applications, it can inadvertently happen (typically when the offending program encounters a bug in its code). When a program writes outside of its own address space, it may result in incorrect results or an aborted program. Worse still, it could result in the entire system crashing.

An operating system with *protected memory* gives each running program its own memory space that can't be accessed by other programs. The advantage to memory protection should be obvious: while a bug in a program may crash that program, the entire system won't freeze and a reboot won't be necessary. The BeOS has protected memory. Should a program attempt to access memory outside its own well-defined area, the BeOS will terminate the rogue program while leaving any other running applications unaffected. To the delight of users, their machines running BeOS rarely crash.

Virtual Memory

To accommodate the simultaneous running of several applications, some operating systems use a memory scheme called virtual memory. Virtual memory is memory other than RAM that is devoted to holding application code and data. Typically, a system reserves hard drive space and uses that area as virtual memory. As a program executes, the processor shuffles application code and data between RAM and virtual memory. In effect, the storage space on the storage device is used as an extension of RAM.

The BeOS uses virtual memory to provide each executing application with the required memory. For any running application, the system first uses RAM to handle the program's needs. If there is a shortage of available physical memory, the system then resorts to hard drive space as needed.

Less Hindered by Backward Compatibility

When a company such as Apple or Microsoft sets about to upgrade its operating system, it must take into account the millions of users that have a large investment in software designed to run on the existing version of its operating system. So no matter how radical the changes and improvements are to a new version of an operating system, the new OS typically accommodates these users by supplying backward compatibility.

Backward compatibility—the ability to run older applications as well as programs written specifically for the new version of the OS—helps keep the installed base of users happy. But backward compatibility has a downside: it keeps an upgrade to an operating system from reaching its true potential. In order to keep programs that were written for old technologies running, the new OS cannot include some new technologies that would "break" these existing applications. As a new operating system, the BeOS had no old applications to consider. It was designed to take full advantage of today's fast hardware and to incorporate all the available modern programming techniques. As subsequent releases of the BeOS are made available, backward compatibility does become an issue. But it will be quite a while before original applications need major overhauling (as is the case for, say, a Macintosh application written for an early version of the Mac OS).

Structure of the BeOS

Be applications run on hardware driven by either Intel or PowerPC microprocessors (check the BeOS Support Guides page at *http://www.be.com/support/guides/* for links to lists of exactly which Intel and PowerPC machines are currently supported). Between the hardware and applications lies the BeOS software. As shown in Figure 1-1, the operating system software consists of three layers: a microkernel layer that communicates with the computer's hardware, a server layer consisting of a number of servers that each handle the low-level work of common tasks (such as printing), and a software kit layer that holds several software kits—shared libraries (known as dynamically linked libraries, or DLLs, to some programmers) that act as a programmer's interface to the servers and microkernel.

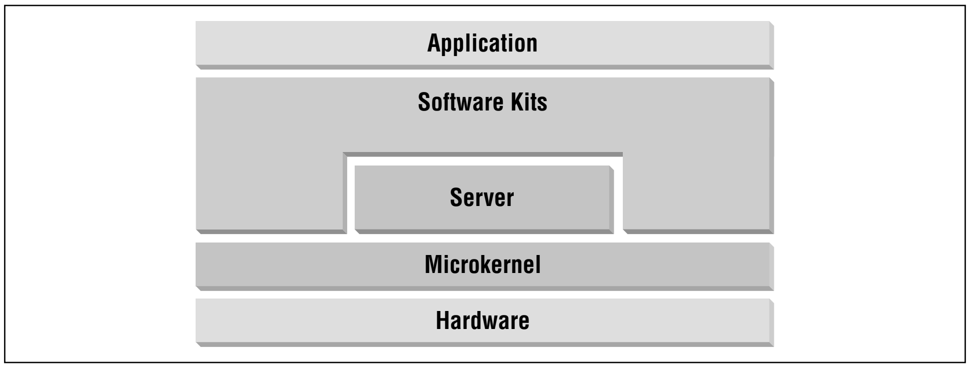


Figure 1-1. The layers of the BeOS reside between applications and hardware

Microkernel

The bottom layer consists of the microkernel. The microkernel works directly with the hardware of the host machine, as well as with device drivers. The code that makes up the microkernel handles low-level tasks critical to the control of the computer. For instance, the microkernel manages access to memory. The kernel also provides the building blocks that other programs use: thread scheduling, the file system tools, and memory-locking primitives.

Servers

Above the microkernel lies the server layer. This layer is composed of a number of servers—processes that run in the background and carry out tasks for applications that are currently executing. For example, the purpose of the Input Server is to handle access to all the various keyboards, mice, joysticks, and other input devices that may be connected to a machine running the BeOS. Another server is the Application Server, a very important server that handles the display of graphics and application communication. As a programmer you won't work directly with servers; instead, you'll rely on software kits to access the power of the server software.

Kits

Above the server layer is the software kit layer. A kit consists of a number of object-oriented classes that a programmer makes use of when writing a BeOS program. Collectively the classes in the software kits comprise the BeOS API. You know that the abbreviation *API* stands for *application programming interface*. But what does the application interface to? Other software. For Be applications, the kits are the interface to the various servers. For instance, the Application Kit holds several classes used by programmers in your position who are trying to create tools for users. The programmer writes code that invokes methods that are a part of the classes of the Application Kit, and the Application Kit then communicates with the Application Server to perform the specified task. A couple of the other servers you'll encounter in your Be programming endeavors are the Print Server and the Media Server.

Some kits don't rely on servers to carry out microkernel-related operations—the chores they take care of may be simple and straightforward enough that they don't need their own server software. Instead, these kits directly invoke microkernel code. As you can see in Figure 1-1, an application relies directly on the software kits and indirectly on the servers and microkernel.

As you become more proficient at BeOS programming, you'll also become more intimate with the classes that comprise the various software kits. Now that you know this, you'll realize that it is no accident that the majority of this book is devoted to understanding the purpose of, and working with, the various BeOS kits.

This book is tutorial in nature. Its purpose is to get you acquainted with the process of developing applications that run on the BeOS and to provide an overview of the BeOS API. Its purpose *isn't* to document the dozens of classes and hundreds of member functions that make up the BeOS API. After—or while—reading this book, you may want such a reference. If you do, consider the books *Be Developer's Guide* and *Be Advanced Topics*, also by O'Reilly & Associates.

Software Kits and Their Classes

The application programming interface of the BeOS is object-oriented—the code that makes up the software kits is written in C++. If you have experience programming in C++ on any platform, you're already at the midpoint in your journey to becoming adept at BeOS programming. Now you just need to become proficient in the layout and use of the classes that make up the software kits.

Software Kit Overview

The BeOS consists of about a dozen software kits—the number is growing as the BeOS is enhanced. Don't panic, though—you won't be responsible for knowing about all the classes in all of the kits. Very simple applications require only the classes from a very few of the kits. For instance, an application that simply displays a window that holds text uses the Application Kit and the Interface Kit. A more complex application requires more classes from more kits. Presentation software that stores sound and video data in files, for example, might require the use of classes from the Storage Kit, the Media Kit, and the Network Kit—as well as classes from the two previously mentioned kits. While it's unlikely that you'll ever write a program that uses all of the BeOS kits, it's a good idea to at least have an idea of the purpose of each.

> The kits of the BeOS are subject to change. As the BeOS matures, new functionality will be added. This functionality will be supported by new classes in existing kits and, perhaps, entirely new software kits.

Application Kit

The Application Kit is a small but vitally important kit. Because every application is based on a class derived from the `BApplication` class that is defined in this kit, every application uses the Application Kit.

The Application Kit defines a messaging system (described later in this chapter) that makes applications aware of events (such as a click of a mouse button by the user). This kit also give applications the power to communicate with one another.

Interface Kit

The Interface Kit is by far the largest of the software kits. The classes of this kit exist to supply applications with a graphical user interface that fully supports user interaction. The definition of windows and the elements that are contained in windows (such as scrollbars, buttons, lists, and text) are handled by classes in this kit. Any program that opens at least one window uses the Interface Kit.

Storage Kit
: The Storage Kit holds the classes that store and update data on disks. Programs that work with files will work with the Storage Kit.

Support Kit
: As its name suggests, the contents of the Support Kit support the other kits. Here you'll find the definitions of datatypes, constants, and a few classes. Because the Support Kit defines many of the basic elements of the BeOS (such as the Boolean constants `true` and `false`), all applications use this kit.

Media Kit
: The Media Kit is responsible for the handling of real-time data. In particular, this kit defines classes that are used to process audio and video data.

Midi Kit
: The Midi Kit is used for applications that process MIDI (Musical Instrument Digital Interface) data.

Kernel Kit
: The Kernel Kit is used by applications that require low-level access to the BeOS microkernel. This kit defines classes that allow programmers to explicitly create and maintain threads.

Device Kit
: The Device Kit provides interfaces to hardware connectors (such as the serial port), and is necessary only for programmers who are developing drivers.

Network Kit
: The Network Kit exists to provide TCP/IP services to applications.

OpenGL Kit
: The OpenGL Kit provides classes that allow programmers to add 3D capabilities to their programs. The classes aid in the creation and manipulation of three-dimensional objects.

Translation Kit
: The Translation Kit is useful when a program needs to convert data from one media format to another. For instance, a program that can import an image of one format (such as a JPEG image) but needs to convert that image to another format might make use of this kit.

Mail Kit
: The Mail Kit assists in adding Internet email services (such as sending messages using Simple Mail Transfer Protocol (SMTP) to an application).

Game Kit

The Game Kit—which is under development as of this writing—consists of two major classes that support game developers.

BeOS Naming Conventions

Some of the many classes that make up the BeOS are discussed a little later. As they're introduced, you'll notice that each starts with an uppercase letter "B," as in `BMessage`, `BApplication`, and `BControl`. This is no accident, of course—the software of the kits follows a naming convention.

The BeOS software kits consist of classes (which contain member functions and data members), constants, and global variables. The BeOS imposes a naming convention on each of these types of elements so that anyone reading your code can readily distinguish between code that is defined by the BeOS and code that is defined by your own program. Table 1-1 lists these conventions.

Table 1-1. BeOS Naming Conventions

Category	Prefix	Spelling	Example
Class name	B	Begin words with uppercase letter	`BRect`
Member function	none	Begin words with uppercase letter	`OffsetBy()`
Data member	none	Begin words (excepting the first) with uppercase letter	`bottom`
Constant	B_	All uppercase	`B_LONG_TYPE`
Global variable	be_	All lowercase	`be_clipboard`

Classes of the BeOS always begin with an uppercase "B" (short for "BeOS", of course). Following the "B" prefix, the first letter of each word in the class name appears in uppercase, while the remainder of the class name appears in lowercase. Examples of class names are `BButton`, `BTextView`, `BList`, and `BScrollBar`.

Member functions that are defined by BeOS classes have the first letter of each word in uppercase and the remainder of the function name in lowercase. Examples of BeOS class member function names are `GetFontInfo()`, `KeyDown()`, `Frame()`, and `Highlight()`.

Data members that are defined by BeOS classes have the first letter of each word in uppercase and the remainder of the data member name in lowercase, with the exception of the first word—it always begins in lowercase. Examples of BeOS class data member names are `rotation` and `what`.

I've included only a couple of examples of data member names because I had a hard time finding any! Be engineers went to great lengths to hide data members. If you peruse the Be header files you'll find a number of data members—but most are declared private and are used by the classes themselves rather than by you, the programmer. You'll typically make things happen in your code by invoking member functions (which themselves may access or alter private data members) rather than by working directly with any data members.

Constants defined by BeOS always begin with an uppercase "B" followed by an underscore. The remainder of the constant's name is in uppercase, with an underscore between words. Examples include: `B_WIDTH_FROM_LABEL`, `B_WARNING_ALERT`, `B_CONTROL_ON`, and `B_BORDER_FRAME`.

The BeOS software includes some global variables. Such a variable begins with the prefix "be_" and is followed by a lowercase name, as in: `be_app`, `be_roster`, and `be_clipboard`.

Software Kit Inheritance Hierarchies

The relationships between classes of a software kit can be shown in the *inheritance hierarchy* for that kit. Figure 1-2 shows such an inheritance hierarchy for the largest kit, the Interface Kit.

The kits that make up the BeOS don't exist in isolation from one another. A class from one kit may be derived from a class defined in a different kit. The `BWindow` class is one such example. Kits serve as logical groupings of BeOS classes—they make it easier to categorize classes and conceptualize class relationships.

Figure 1-2 shows that the object-oriented concept of inheritance—the ability of one class to inherit the functionality of another class or classes—plays a very large role in the BeOS. So too does multiple inheritance—the ability of a class to inherit from multiple classes. In the figure, you see that almost all of the Interface Kit classes are derived from other classes, and that many of the classes inherit the contents of several classes. As one example, consider the six control classes pictured together in a column at the far right of Figure 1-2. An object of any of these classes (such as a `BButton` object) consists of the member functions defined in that class as well as the member functions defined by all of the classes from which it is directly and indirectly derived: the `BControl`, `BInvoker`, `BView`, `BHandler`, and

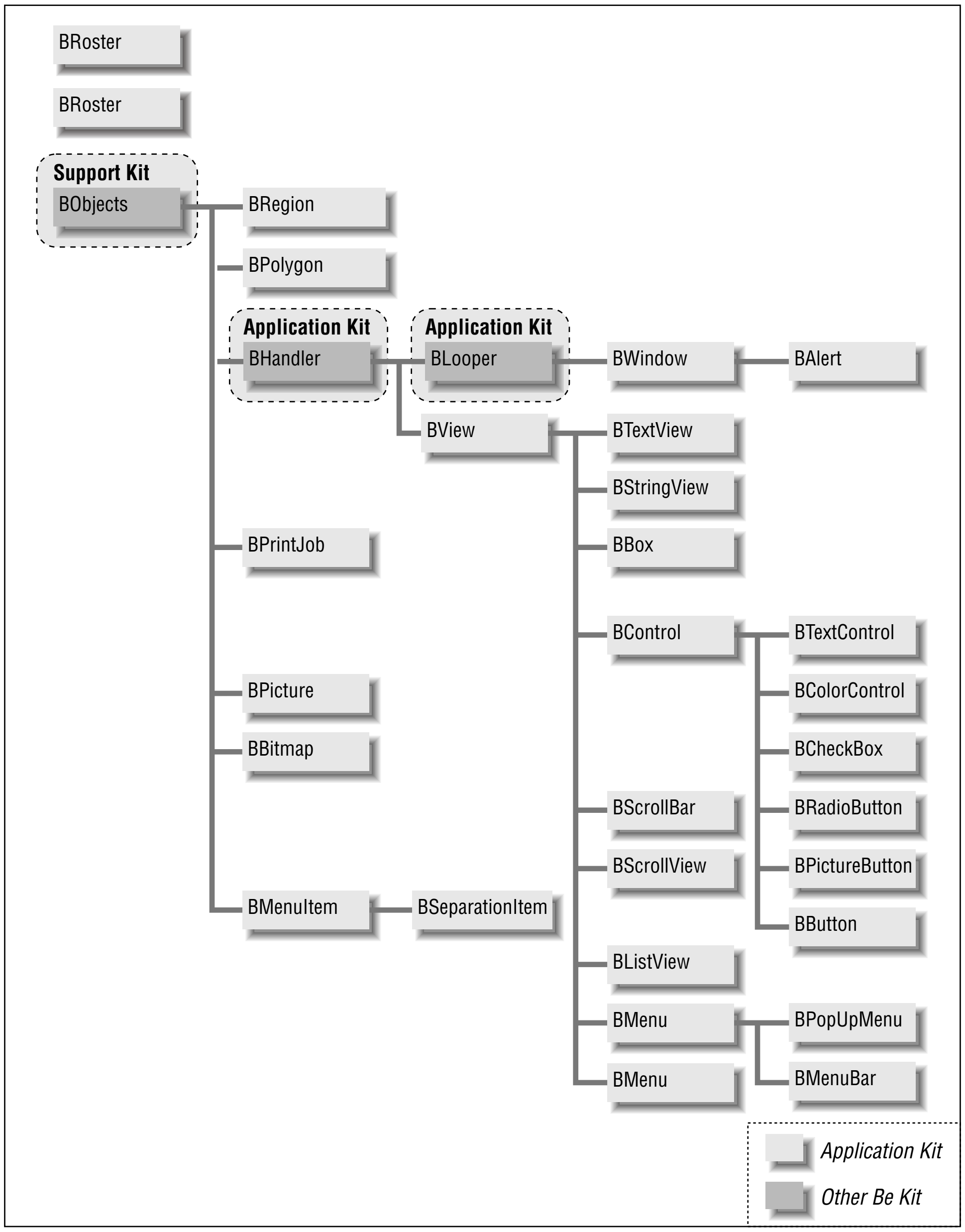


Figure 1-2. The inheritance hierarchy for the Interface Kit

`BArchivable` classes. Figure 1-3 isolates the discussed classes for emphasis of this point. This figure shows that in inheritance hierarchy figures in this book, a class pictured to the left of another class is higher up in the hierarchy. In Figure 1-3, `BView` is derived from `BHandler`, `BControl` is derived from `BView`, and so forth.

Understanding the class hierarchies of the BeOS enables you to quickly determine which class or classes (and thus which member functions) you will need to use to implement whatever behavior you're adding to your program. Obviously, knowledge of the class hierarchies is important. Don't be discouraged, though, if the hierarchies shown in Figures 1-2 and 1-3 don't make complete sense to you. This chapter only provides an overview of the object-oriented nature of the BeOS. The remainder of the book fills in the details of the names, purposes, and uses of the important and commonly used classes.

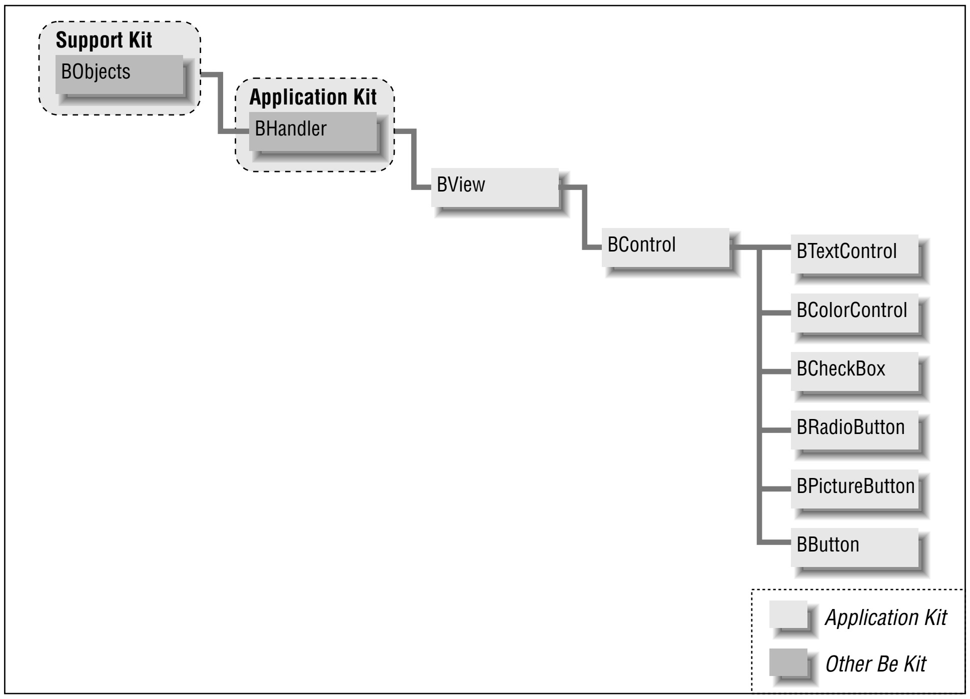


Figure 1-3. The Interface Kit classes that contribute to the various control classes

The `BControl` class defines member functions that handle the needs common to any type of control. For instance, a control should be able to have two states: enabled or disabled. An enabled control is active, or usable by the user. A disabled control is inactive—and has a greyed-out look to let the user know it is unusable. To give controls the ability to implement this behavior, the `BControl` class includes the `SetEnabled()` member function. This routine is used to enable or disable a control—any kind of control. Individual types of controls will have some needs that aren't common to all other types of controls and thus can't be implemented by the `BControl` class. For example, different controls (such as buttons and checkboxes) have different looks. To make it possible for each control type to be able to draw itself, each control class defines its own constructor to initialize the control and a `Draw()` member function to handle the drawing of the control.

Not all BeOS classes are derived from other classes—there are a few classes that don't rely on inheritance. Two examples, both of which happen to be in the Interface Kit, are the `BRect` and `BPoint` classes. The `BRect` class is used to create objects representing rectangles. A rectangle is an easily defined, two-dimensional shape that's considered a basic datatype. As such, it doesn't need to inherit the functionality of other classes. The `BPoint` class is not a derived class for the same reason.

BeOS Programming Fundamentals

In the previous section, you gained an understanding of how the BeOS is composed of numerous interrelated classes that are defined in software kits. Together these classes form an application framework from which you build your Be applications. Your program will create objects that are based on some of the BeOS classes. These objects will then communicate with one another and with the operating system itself through the use of messages. In this section, you'll look at a few of the most important of these classes, and you'll see how they're used. You'll also see how messages play a role in a BeOS program. To make the transition from the theoretical to the practical, I'll supply you with a few C++ snippets—as well as the code for a complete Be application. In keeping with the introductory nature of this chapter, I'll make this first application a trivial one.

Messages, Threads, and Application Communication

Earlier in this chapter, you read that the BeOS is a multithreaded operating system. You also read that the term *multithreaded* isn't just bandied about by BeOS advocates for no good reason—it does in fact play a key role in why the BeOS is a powerful operating system. Here, you'll get an introduction as to why that's true. In Chapter 4, *Windows, Views, and Messages*, I'll have a lot more to say about multithreading.

Applications and messages

A Be application begins with the creation of an object of a class type derived from the `BApplication` class—a class defined in the Application Kit. Figure 1-4 shows how the BApplication class fits into the inheritance hierarchy of the Application Kit. Creating an application object establishes the application's main thread, which serves as a connection between the application and the Application Server. Earlier in this chapter, you read that a BeOS server is software that provides services to an application via a software kit. The Application Server takes care of many of the tasks basic to any application. One such task is reporting user actions to applications. For instance, if the user clicks the mouse button or presses a key on the keyboard, the Application Server reports this information to executing applications. This information is passed in the form of a message, and is received by an application in its main thread. A *message* is itself an object—a parcel of data that holds details about the action being reported. The ability of the operating system to determine the user's actions and then use a separate thread to pass detailed information about that action to a program makes your programming job easier.

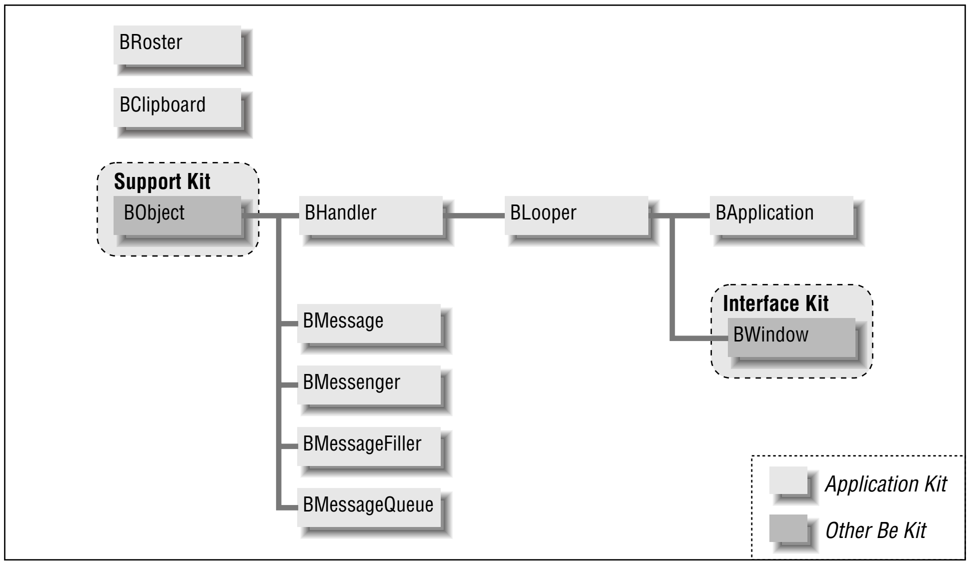


Figure 1-4. The inheritance hierarchy for the Application Kit

An application's code can explicitly define BMessage objects and use them to pass information. What I've discussed above, however, are *system messages* that originate from within the BeOS itself. The movement of the mouse, the pressing of a keyboard key, a mouse button click in a window's close button, and a mouse button click and drag in a window's resize knob are all examples of system messages. Each type of system message has a command constant associated with it. This constant names the type of event the message represents. Examples of command constants are B_KEY_DOWN, B_MOUSE_DOWN, and B_WINDOW_RESIZED.

Message loops and message handling

The BeOS defines classes that allow the creation of objects that can work with messages. The Application Kit defines two such classes: the `BLooper` class and the `BHandler` class. The `BLooper` class is used to create an object that exists in its own thread. The purpose of this thread is to run a *message loop*. As messages reach a message loop thread, they are placed in a queue. From this queue the thread extracts and dispatches messages one after another.

A message is always dispatched to an object of the `BHandler` class. The job of the `BHandler` object is to handle the message it receives. How it handles a message is dependent on the type of message it receives.

As shown back in Figure 1-4, the `BLooper` class is derived from the `BHandler` class. This means that an object of the `BLooper` class (or of a class derived from `BLooper`) can have both a message loop that dispatches messages and can receive these messages itself for handling. Because the `BApplication` class and the `BWindow` class are derived from the `BLooper` class, such is the case for the application itself and any of its windows. Just ahead you'll read a little more on how an application and windows can in fact watch for and respond to messages.

To summarize, a `BLooper` object has a thread that runs a message loop that dispatches messages, and a `BHandler` object receives and handles these dispatched messages. Because the `BLooper` class is derived from the `BHandler` class, a `BLooper` object can dispatch and receive and handle messages. A `BHandler` object can only receive and handle messages. From that description it might seem that all objects that deal with messages might as well be `BLooper` objects. After all, the `BLooper` class provides more functionality. As you read more about messaging, you'll see why that path isn't the one to take. Each `BLooper` object creates a new thread and dominates it with a message loop—the thread shouldn't be used for any other purpose. A `BHandler` object, on the other hand, doesn't create a thread. While having multiple threads in a program can be advantageous, there's no benefit to creating threads that go unused.

Defining and Creating Windows

At the heart of the graphical user interface of the Be operating system is the window. Be applications are window-based—windows are used to accept input from the user by way of menus and controls such as buttons, and to display output to the user in the form of graphics and text. The Interface Kit—the largest of the kits—exists to enable programmers to provide their Be applications with a graphical user interface that includes windows. It is classes of the Interface Kit that you'll be using when you write a program that displays and works with windows.

The BWindow class

Almost all Be applications display at least one window and therefore use the BWindow class—one of the dozens of classes in the Interface Kit. If you look in the *Window.h* header file that is a part of the set of header files used in the compilation of a Be program, you'll find the declaration of the BWindow class. I've included a partial listing (note the ellipses) of this class below. Here you can see the names of a dozen of the roughly one hundred member functions of that class. Looking at the names of some of the member functions of the BWindow class gives you a good indication of the functionality the class supplies to BWindow objects.

```
class BWindow : public BLooper {

public:
                      BWindow(BRect frame,
                              const char *title,
                              window_type type,
                              uint32 flags,
                              uint32 workspace = B_CURRENT_WORKSPACE);
...
virtual               ~BWindow();

virtual  void         Quit();
         void         Close();

virtual  void         DispatchMessage(BMessage *message, BHandler *handler);
virtual  void         MessageReceived(BMessage *message);
virtual  void         FrameMoved(BPoint new_position);
...
virtual  void         Minimize(bool minimize);
virtual  void         Zoom(BPoint rec_position, float rec_width, float rec_
height);
...
         void         MoveBy(float dx, float dy);
         void         MoveTo(BPoint);
         void         MoveTo(float x, float y);
         void         ResizeBy(float dx, float dy);
         void         ResizeTo(float width, float height);
virtual  void         Show();
virtual  void         Hide();
         bool         IsHidden() const;
...
         const char   *Title() const;
         void         SetTitle(const char *title);
         bool         IsFront() const;
         bool         IsActive() const;

...
}
```

If you're interested in viewing the entire BWindow class declaration, you can open the *Window.h* header file. The path that leads to the *Window.h* file will most likely be *develop/headers/be/interface*. There's a good chance that your development environment resides in your root directory, so look for the *develop* folder there. You can open any header file from the Edit text editor application or from the BeIDE. The Metrowerks CodeWarrior BeIDE programming environment is introduced later in this chapter and discussed in more detail in Chapter 2, *BeIDE Projects*.

Deriving a class from BWindow

A Be program that uses windows could simply create window objects using the BWindow class. Resulting windows would then have the impressive functionality provided by the many BWindow member functions, but they would be very generic. That is, while they could be moved, resized, and closed (BWindow member functions take care of such tasks), they would have no properties that made them unique from the windows in any other application. Instead of simply creating a BWindow object, programs define a class derived from the BWindow class. This derived class, of course, inherits the member functions of the BWindow class. Additionally, the derived class defines new member functions and possibly overrides some inherited member functions to give the class the properties that windows of the application will need. The following snippet provides an example:

```
class SimpleWindow : public BWindow {

public:
                    SimpleWindow(BRect frame);

virtual  bool       QuitRequested();
};
```

From the BeOS naming conventions section of this chapter, you know that the name of a class that is a part of the BeOS API (such as BWindow) always starts with an uppercase "B." As long as my own classes (such as SimpleWindow) don't start with an uppercase "B," anyone reading my code will be able to quickly spot classes that are of my own creation.

The SimpleWindow constructor

The SimpleWindow class declares a constructor and one member function. The definition of the constructor follows.

```
SimpleWindow::SimpleWindow(BRect frame)
    : BWindow(frame, "A Simple Window", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

This constructor makes use of a technique common in Be applications: the constructor for the class derived from the `BWindow` class invokes the `BWindow` class constructor. Calling the `BWindow` class constructor is important because the `BWindow` constructor arguments provide important information to the window object that is to be created. In Chapter 5, *Drawing*, I discuss the four `BWindow` constructor parameters in detail. In this introduction, it will suffice for me to say that the four parameters specify the following for a newly created window object:

- The frame, or content area of the window (the size and screen placement of the window)
- The name of the window (as it will appear in the window's tab)
- The type of the window (the look and feel of the window)
- The behavior of the window (whether it has a resize knob, and so forth)

Recall from your C++ background that when the definition of a constructor is followed by a single colon and the base class constructor, the effect is that the base class constructor gets invoked just before the body of the derived class constructor executes.

In this example, the `BWindow` constructor's first argument comes from the sole argument passed to the `SimpleWindow` constructor. A hardcoded string serves as the second argument to the `BWindow` constructor. The third and fourth arguments are constants defined in the *Window.h* header file.

Notice that the body of the `SimpleWindow` constructor is empty. This tells you that the only chore of the `SimpleWindow` constructor is to invoke the `BWindow` constructor. You have to call the `BWindow` constructor; this function creates a new window and spawns a new thread of execution in which the window runs, and starts up a message loop in that same thread. In a Be program, each window exists in its own thread and each window is notified of system messages that involve the window. You'll be pleased to find that the work of maintaining a window's thread and of keeping a window informed of system messages (such as a mouse button click in the window) is taken care of by the operating system. You'll be even more pleased to find that for some system messages, even the window's response to the message is handled by the BeOS. For instance, you needn't write any code that watches for or handles the resizing of a window.

A window can watch for and respond to messages because the `BWindow` class inherits from both the `BLooper` and `BHandler` classes (see Figure 1-4). A window is thus a window (from `BWindow`), an object that includes a message loop (from `BLooper`), and an object that responds to messages (from `BHandler`). This pertains to `BWindow` objects and, of course, objects created from classes derived from the `BWindow` class—such as objects of my `SimpleWindow` class type.

The SimpleWindow QuitRequested() member function

The `SimpleWindow` class declares one member function. Here's the definition of `QuitRequested()`:

```
bool SimpleWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}
```

`QuitRequested()` is actually a member function of the `BLooper` class. Because my `SimpleWindow` class is derived from the `BWindow` class, which in turn is derived from the `BLooper` class, this member function is inherited by the `SimpleWindow` class. By redeclaring `QuitRequested()`, `SimpleWindow` is overriding this function.

If I had opted *not* to override the `QuitRequested()` member function in the `SimpleWindow` class, it would be the `BLooper` version of this function that would execute upon a user mouse button click in a window's close button. Like my `SimpleWindow` version of `QuitRequested()`, the version of `QuitRequested()` defined by the `BLooper` class returns a value of `true`. The effect is for an object of `BLooper` type to kill the thread it is running in and delete itself. That sounds much like what I'd like to do in response to the user's attempt to close a window—kill the thread in which the window is running. And it is. But in my trivial example program, I'll only be allowing a single window to appear on the screen. When the user closes that window, I'll want to terminate the application, not just the window. That's the action I've added to the `QuitRequested()` function with this line of code:

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

A mouse button click in a window's close button generates a system message that gets passed to the window. The window is a type of `BLooper`, so it captures messages in its message loop. A window is also a type of `BHandler`, so it can handle this message (as opposed to having to pass it to some other type of object for handling). It handles the message by invoking `QuitRequested()`. If my `SimpleWindow` class didn't override the `BLooper` version of this function, the `BLooper` version would be executed and the window would close—but the application wouldn't quit. That's because the BLooper version only kills its own thread in order to delete itself. Because SimpleWindow does override QuitRequested(), it is the SimpleWindow version of this function that instead gets invoked. The SimpleWindow version posts a B_QUIT_REQUESTED message to the application to tell the application to also quit. The notation used in the above line (be_app->PostMessage()) is new to you, so it's worthy of examination.

You already know that a window is a type of BLooper, but there is another very important type of BLooper: the application itself. An application is always represented by an application object—an object of the BApplication class that is defined in the Application Kit (refer back to Figure 1-4 if you need to verify the relationship between the BLooper class and the BWindow and BApplication classes). The PostMessage() routine is a member function of the BLooper class. A BLooper object can invoke this function to place a message in the queue of its own message loop.

As you'll see ahead, be_app is a global variable that represents the application object. This variable is always available for use by your code. The above line of code invokes the application object's version of the PostMessage() function. The message the application object places in its message loop is one that tells itself to quit.

The variable be_app is a pointer to an object—the use of the membership access operator (->) to invoke PostMessage() tells you that. As is often the case in object-oriented programming, a pointer to an object is simply referred to as the object itself. So in this book, as well as in other Be documentation, you'll read about the "application object" in discussions that include mention of be_app.

After the call to PostMessage() places a request to kill the application thread in the application object's message queue, the SimpleWindow version of QuitRequested() returns a value of true. Remember, QuitRequested() won't be called by my own code—it will be invoked by the system in response to the mouse button click in a window's close button. By returning a value of true, QuitRequested() is telling the system that the requested service should be carried out. The system will then kill the window thread to dispose of the window.

Previously I mentioned that the BeOS took care of system messages involving a window. I gave the example of window resizing being handled by the operating system. Yet here I'm discussing how my own code is being used to handle the system message that gets generated by a click in a window's close button. It's important to restate what I just discussed. It wouldn't be necessary to include any window-closing code in my `SimpleWindow` class if my goal was only to have a mouse button click in the close button result in the closing of the window. The `QuitRequested()` function defined in `BLooper` would take care of that by killing the window's thread. I, however, also want the program to terminate when a window's close button is clicked. To get that extra action, I need to override `QuitRequested()`.

In summary, a mouse button click in a window's close box automatically causes `QuitRequested()` to execute. If a window class doesn't override this function, the window closes but the application continues to run. If the window class does override this function, what happens is determined by the code in this new version of the function. In my `SimpleWindow` class example, I choose to have this function tell the application to quit and tell the window to close.

Creating a window object

Declaring a class and defining its constructor and member functions only serves to specify how objects of this class type will look and behave—it doesn't actually create any such objects. To create and display a window object you'll first declare a variable that will be used to point to the object:

```
SimpleWindow  *aWindow;
```

Before going ahead and allocating the memory for a new window object, your code should declare and set up a rectangle object that will serve to establish the size and screen placement of the new window:

```
BRect   aRect;

aRect.Set(20, 20, 200, 60);
```

The above snippet first declares and creates a rectangle object. The `BRect` class was briefly mentioned earlier in this chapter—it is discussed at length in Chapter 6, *Controls and Messages*. Next, the `Set()` member function of the `BRect` class is called to establish the dimensions of the rectangle. The `Set()` function's four parameters specify the left, top, right, and bottom coordinates, respectively.

The above call to `Set()` establishes a rectangle that will be used to create a window with a left side 20 pixels in from the left of the screen and a top 20 pixels down from the top of the screen. While the window that will use this rectangle would seem to have a width of 180 pixels (200–20) and a height of 40 pixels (60–20), it will actually have a width of 181 pixels and a height of 41 pixels. This apparent one-pixel discrepancy is explained in the screen and drawing coordinates section of Chapter 5.

With the window's bounding rectangle established, it's time to go ahead and create a new window. This line of code performs that feat:

```
aWindow = new SimpleWindow(aRect);
```

To dynamically allocate an object, use the `new` operator. Follow `new` with the constructor of the class from which the object is to be created. If you glance back at the section that describes the `SimpleWindow` constructor, you'll be reminded that this function has one parameter—a `BRect` object that defines the size of the window and gets passed to the `BWindow` constructor.

After allocating memory for a `SimpleWindow` object, the system returns a pointer to this memory. That pointer is stored in the `aWindow` variable. Until this new window is deleted, it can be accessed via this pointer. This line of code provides an example:

```
aWindow->Show();
```

By default, a newly created window is not visible. To display the window, your code should call the `BWindow` member function `Show()` as I've done in the above line.

Let's end this section by pulling together the code that's just been introduced. Here's a look—with comments—at how a window is typically created in a Be application:

```
SimpleWindow  *aWindow;             // declare a pointer to a SimpleWindow
                                    // object
BRect          aRect;               // declare a rectangle object

aRect.Set(20, 20, 200, 60);         // specify the boundaries of the
                                    // rectangle
aWindow = new SimpleWindow(aRect);  // create a SimpleWindow object

aWindow->Show();                    // display the newly created window
```

You may have noticed that I used the `new` operator to create a window object, but created a rectangle object without a `new` operator. In Be programs, objects can always be, and sometimes are, allocated dynamically. That is, the `new` operator is used to set aside memory in the program's heap—as I've done with the window. Some objects, however, are allocated statically. That is, an object variable (rather than a pointer) is declared in order to set aside memory on the stack—as I chose to do with the rectangle. Creating an object that resides on the stack is typically done for objects that are temporary in nature. In the above snippet, the rectangle object fits that bill—it exists only to provide values for the window's dimensions. After the creation of the window, the rectangle object is unimportant.

If you aren't familiar with the term *heap,* I should explain that it is an area in a program's address space that exists to hold objects that are created dynamically during the execution of a program. An object can be added or deleted from the heap without regard for its placement in the heap, or for the other contents of the heap. The *stack*, on the other hand, is used to store objects in a set order—objects are stacked one atop the other. Objects can only be added and removed from the top of the stack.

Defining an Application

Every Be program must create an object that represents the application itself. This one object is the first one created when a program launches and the last one deleted when the program quits. One of the primary purposes of the application object is to make and maintain a connection with the Application Server. It is the Application Server that takes care of the low-level work such as handling interactions between windows and monitoring input from data entry sources such as the keyboard and mouse.

The BApplication class

To create the application object, you first define a class that is derived from the `BApplication` class and then create a single instance of that class (an *instance* being nothing more than another name for an *object*). From the *Application.h* header file, here's a partial listing of the `BApplication` class:

```
class BApplication : public BLooper {

public:
                    BApplication(const char * signature);
virtual             ~BApplication();
...
virtual  thread_id  Run();
virtual  void       Quit();
...
         void       ShowCursor();
         void       HideCursor();
...
}
```

Referring back to Figure 1-4, you can see that the `BApplication` class is both a type of `BLooper` and a type of `BHandler`. This means that an application object is capable of having a message loop, and is capable of handling messages in that loop. As it turns out, the application object runs the application's main message loop. It is this loop that receives messages that affect the application.

Deriving a class from BApplication

Every application defines a single class derived from the `BApplication` class. A program that will be communicating with other programs may define a number of member functions to handle this interapplication communication. A simpler application might define nothing more than a constructor, as shown here:

```
class SimpleApplication : public BApplication {

public:
                SimpleApplication();
};
```

The SimpleApplication constructor

When a Be program starts, it's common practice for the program to open a single window without any help from the user. Because the `SimpleApplication()` constructor executes at program launch (that's when the application object is created), it would make sense to let this constructor handle the job of creating and displaying a window. Here's a look at how the constructor does that:

```
SimpleApplication::SimpleApplication()
    : BApplication("application/x-vnd.dps-simpleapp")
{
    SimpleWindow  *aWindow;
    BRect          aRect;

    aRect.Set(20, 20, 200, 60);
    aWindow = new SimpleWindow(aRect);

    aWindow->Show();
}
```

Just as my `SimpleWindow` class invoked the constructor of the class it was derived from (the `BWindow` class), so does my `SimpleApplication` class invoke the constructor of the class it is derived from (the `BApplication` class). Invoking the `BApplication` constructor is necessary for a few reasons. The `BApplication` constructor:

- Connects the application to the Application Server
- Provides the application with a unique identifying signature for the program
- Sets the global variable `be_app` to point to the new application object

The connecting of an application to the Application Server has already been mentioned. This connection allows the server to send messages to the application. The application signature is a MIME (Multipurpose Internet Mail Extensions) string. The phrase *application/x-vnd.* should lead off the signature. Any characters you want can follow the period, but convention states that this part of the MIME string consist of an abbreviation of your company's name, a hyphen, and then part or all of the program name. In the above example, I've used my initials (*dps*) as the company name. I've elected to name my program SimpleApp, so the MIME string ends with *simpleapp*. The assignment of an application's signature is described at greater length in Chapter 2. The global variable `be_app` was introduced in the earlier discussion of windows. This variable, which is always available for your program's use, always points to your program's `BApplication` object.

In the "Creating a window object" section that appears earlier, you saw five lines of code that demonstrated how a window object could be created and how its window could be displayed on the screen. If you compare the body of the `SimpleApplication()` constructor to those five lines, you'll see that they are identical.

Creating an application object

After defining a class derived from the `BApplication` class, it's time to create an application object of that class type. You can create such an object dynamically by declaring a pointer to the class type, then using the `new` operator to do the following: allocate memory, invoke a constructor, and return a pointer to the allocated memory. Here's how that's done:

```
SimpleApplication  *myApplication;

myApplication = new SimpleApplication();
```

After the above code executes, `myApplication` can be used to invoke any of the member functions of the `BApplication` class (from which the `SimpleApplication` class is derived). In particular, soon after creating an application object, your Be program will invoke the `BApplication Run()` member function:

```
myApplication->Run();
```

The `Run()` function kicks off the message loop in the application's main thread, and then it begins processing messages. Not only is a call to this function important, it's necessary; an application won't start running until `Run()` is invoked.

A program's application object is typically declared in `main()`, and is accessed by the global variable `be_app` outside of `main()`. So there's really no need to have the application object reside in the heap—it can be on the stack. Here's how the creation of the application object looks when done using a variable local to `main()`:

```
SimpleApplication  myApplication;

myApplication.Run ();
```

This second technique is the way the application object will be created in this book, but you should be aware that you may encounter code that uses the first technique.

The main() Routine

The preceding pages introduced the C++ code you'll write to create an application object and start an application running. One important question remains to be answered, though: where does this code go? Because the application object must be created immediately upon application launch (to establish a connection to the Application Server), it should be obvious that this code must appear very early in the program. A C++ program always begins its execution at the start of a routine named `main()`, so it shouldn't come as a surprise that it is in `main()` that you'll find the above code. Here's a look at a `main()` routine that is typical of a simple Be application:

```
int  main()
{
    SimpleApplication   myApplication;

    myApplication.Run();

    return(0);
}
```

To start a program, call `Run()`. When the user quits the program, `Run()` completes executing and the program ends. You'll notice in the above snippet that between `Run()` and `return`, there is no code. Yet the program won't start and then immediately end. Here's why. The creation of the application object (via the declaration of the `BApplication`-derived object `myApplication`) initiates the program's main thread. When `Run()` is then called, the `Run()` function takes control of this main application thread. `Run()` sets up the main message loop in this main thread, and controls the loop and thread until the program terminates. That is, once called, `Run()` executes until the program ends.

The SimpleApp Example Program

The preceding pages have supplied you with all the code you need to write a Be application—albeit a very simple one. Because this same code (or slight variations of it) will appear as a part of the source code for every Be program you write, I've gone to great lengths to explain its purpose. In trying to make things perfectly clear, I'll admit that I've been a bit verbose—I've managed to take a relatively small amount of starter code and spread it out over several pages. To return your focus to just how little code is needed to get a Be program started, I've packaged the preceding snippets into a single source code listing. When compiled and linked, this source code becomes an executable named SimpleApp. When launched, the SimpleApp program displays a single, empty window like the one shown in Figure 1-5.

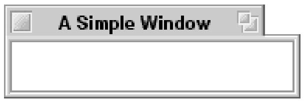


Figure 1-5. The window that results from running the SimpleApp program

The SimpleApp source code listing

Presented next, in its entirety, is the source code for a Be application named SimpleApp. As mentioned, all of the code you're about to see has been presented and discussed earlier in this chapter.

```
#include <Window.h>
#include <Application.h>

class SimpleWindow : public BWindow {

public:
                    SimpleWindow(BRect frame);
    virtual  bool   QuitRequested();
};


SimpleWindow::SimpleWindow(BRect frame)
    : BWindow(frame, "A Simple Window", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}


bool SimpleWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}


class SimpleApplication : public BApplication {

public:
                SimpleApplication();
};


SimpleApplication::SimpleApplication()
    : BApplication("application/x-vnd.dps-simpleapp")
{
    SimpleWindow  *aWindow;
    BRect          aRect;
```

```
        aRect.Set(20, 20, 200, 60);
        aWindow = new SimpleWindow(aRect);

        aWindow->Show();
    }


    main()
    {
        SimpleApplication   myApplication;

        myApplication.Run();

        return(0);
    }
```

What the SimpleApp program does

When you launch SimpleApp you'll see the window pictured in Figure 1-5. You can click the mouse button while the cursor is positioned over the zoom button in the window's tab to expand the window to a size that fills most of your monitor. Click the mouse button with the cursor again positioned over the window's zoom button and the window will return to its previous, much smaller, size. If you click and hold the mouse button while the cursor is positioned over the window's tab, you can drag the window about the monitor. Most important to this discussion is that the SimpleApp source code includes no code to handle such tasks. The zooming and moving of windows is handled by the system, not by the SimpleApp code. This simple demonstration emphasizes the power of the BeOS system software—it is the system software code (rather than the application code) that supplies much of the functionality of a program.

What the SimpleApp program doesn't do

There are a number of things SimpleApp doesn't do—things you'd expect a "real" Be application to do. Most notable of these omissions are menus, support of input by way of controls in the window, and support of output via drawing or writing to the window. Of course these omissions will be rectified in the balance of this book. Starting, in fact, with the next chapter.

BeOS Programming Environment

The programming tool you'll be using to create your Be applications is the BeIDE. This piece of software is an integrated development environment (IDE) and it runs on the Be operating system (so the origin of the name BeIDE is pretty evident!).

The development of a new program entails the creation of a number of files which, collectively, are often referred to as a *project*. Taking a look at an existing project is a good way to get an overview of the files that make up a project, and is also of benefit in understanding how these same files integrate with one another. In Chapter 2 I do just that. There you'll see the HelloWorld example that's the mainstay of getting introduced to a new programming language or platform. In that chapter, you'll also see how an existing project (such as HelloWorld) can be used as the basis for an entirely new program. As a prelude to Chapter 2's in-depth coverage of this project, take a look at Figures 1-6 and 1-7.

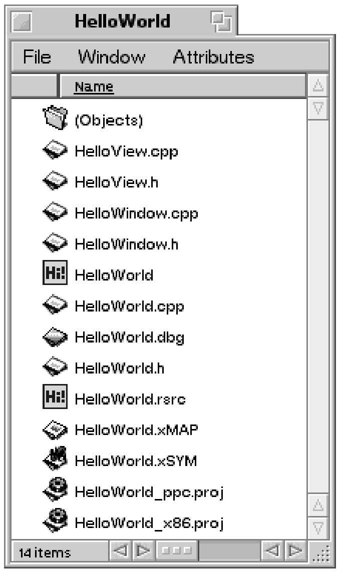


Figure 1-6. The files used in the development of the HelloWorld application

The */boot/apps/Metrowerks* folder holds the BeIDE itself, along with other folders that hold supporting files and projects. Figure 1-6 shows the contents of a folder that holds two projects, both of which are used for building a standalone HelloWorld application. The project named *HelloWorld_ppc.proj* is used to build a Be application that executes on a PowerPC-based machine running the BeOS, while the project named *HelloWorld_x86.proj* is used to build a Be application that executes on an Intel-based PC. In Figure 1-6 you see that a project consists of a number of files. The filename extensions provide a hint of the types of files that make up any one project. A file with an extension of:

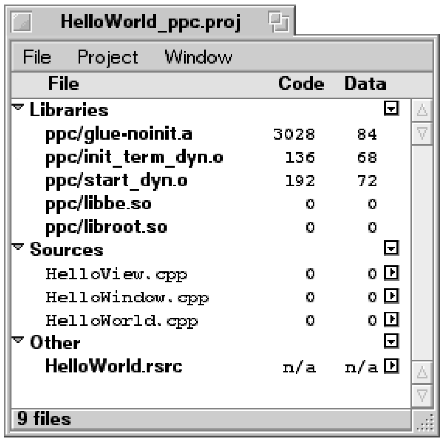
.cpp
Is a C++ source code file

.h Is a header file that holds definitions used by certain C++ source code files

.rsrc
Is a resource file that holds resources that get merged with compiled source code

.proj
Is a project file that is used to organize the files used by the project

Also shown in the HelloWorld folder in Figure 1-6 is a makefile—appropriately named *makefile*. The BeIDE programming environment supports creation of programs from the command line. That is, you can supply the BeIDE compiler and linker with information by editing a makefile and then running that file from the BeOS Terminal application. In this book I'll forego the command-line approach to application development and instead rely on the BeIDE's project-based model. As you'll see in Chapter 2, creating a project file to serve as a means of organizing the files used in a project takes full advantage of the Be graphical user interface. Figure 1-7 shows the window that appears when you use the BeIDE to open the project file for the HelloWorld project.



File	Code	Data
Libraries		
ppc/glue-noinit.a	3028	84
ppc/init_term_dyn.o	136	68
ppc/start_dyn.o	192	72
ppc/libbe.so	0	0
ppc/libroot.so	0	0
Sources		
HelloView.cpp	0	0
HelloWindow.cpp	0	0
HelloWorld.cpp	0	0
Other		
HelloWorld.rsrc	n/a	n/a

Figure 1-7. The project window for the HelloWorld project

You use a project file as a sort of command center for one project. From this one window, you add and remove source code files, libraries, and resource files from the project. You can also double-click on a source code filename in the project window to open, and then edit, that file. Using menu items from the File, Project, and Window menus in the menubar of the project window, you can perform a myriad of commands—including compiling source code and building an application.

In this chapter:
- ***Development Environment File Organization***
- ***Examining an Existing BeIDE Project***
- ***Setting Up a New BeIDE Project***
- ***HelloWorld Source Code***

2
BeIDE Projects

The BeOS CD-ROM includes the BeIDE—Be's integrated development environment (IDE) that's used for creating Be applications. This programming environment consists of a number of folders and files, the most important of which are mentioned in this chapter. In the early stages of your Be programming studies, the folder of perhaps the most interest is the one that holds sample code. Within this folder are a number of other folders, each holding a Be-supplied project. A project is a collection of files that, when compiled, results in a single Be application. The best way to understand just what a project consists of is to take a long look at an existing Be project. That's exactly what I do in this chapter.

After examining an existing project, you'll of course want to create your own. A large part of this chapter is devoted to the steps involved in doing that. Here you'll see how to organize classes into header files and source code files, and how the resource file fits into the scheme of things.

Development Environment File Organization

You'll find that an overview of how the many BeIDE items are organized will be beneficial as you look at existing BeIDE example projects and as you then start to write your own BeOS program.

The BeIDE Folders

When the BeIDE is installed on your hard drive, the folders and files that make up this programming environment end up in a pair of folders named *develop* and *apps* on your boot drive.

The /boot/develop folder

In the *develop* folder you'll find folders that hold header files, libraries, and developer tools. Figure 2-1 shows the contents of the *develop* folder (on a PowerPC-based machine—a BeOS installation on an Intel-based machine results in one additional folder, the *tools* folder). This figure also shows the *apps* folder. The *apps* folder holds over a dozen items, though in Figure 2-1 you just see a single item (the *Metrowerks* folder, discussed later).

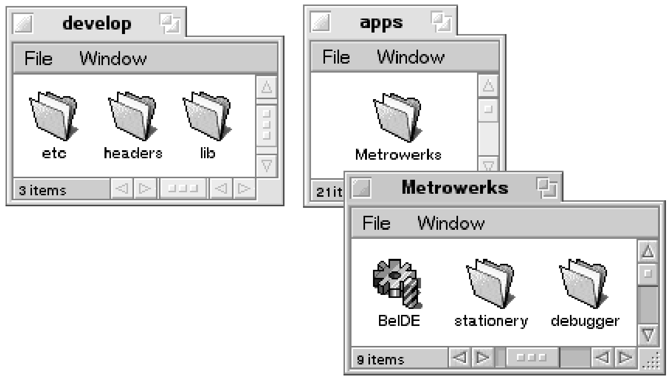


Figure 2-1. Some of the key folders and files used in BeOS programming

In the *develop* folder the *lib* folder holds a number of library files that can be linked to your own compiled code. The act of creating a BeIDE project (discussed later) automatically handles the adding of the basic libraries (*libroot.so* and *libbe.so* at this writing) to the project. As a novice Be programmer, this automatic adding of libraries to a new project is beneficial—it shields you from having to know the details of the purpose of each library. As you become proficient at programming for the BeOS, though, you'll be writing code that makes use of classes not included in the basic libraries—so you'll want to know more about the libraries included in the *develop/lib* folder. Of course you could simply add libraries wholesale to a project to "play it safe," but that tack would be a poor one—especially for programmers developing BeOS applications that are to run on Intel machines. On Intel, all libraries in a project will likely be linked during the building of an application—even if the program uses no code from one or more of the project's libraries. The resulting application will then be unnecessarily large, or will include dependencies on libraries that are not needed.

The *develop* folder *headers* holds the header files that provide the BeIDE compiler with an interface to the software kits. Within the *headers* folder is a folder named *be*. Within that folder you'll find one folder for each software kit. In any one of these folders are individual header files, each defining a class that is a part of one kit. For instance, the `BWindow` class is declared in the *Window.h* header file in the *interface* folder. The complete path to that file is */boot/develop/headers/be/interface/Window.h*.

The *etc* folder in the *develop* folder contains additional developer tools. As of this writing, the primary component in this folder is files used by programmers who prefer a makefile alternative to BeIDE projects. To build an application without creating a BeIDE project, copy the *makefile* template file from this folder to the folder that holds your source code files. Then edit the copied makefile to include the names of the files to compile and link. In this book, I'll focus on the BeIDE project model, rather than the makefile approach, for creating an application.

The *tools* folder in the *develop* folder is found only on Intel versions of the BeOS. This folder contains the x86 (Intel) compiling and linking tools and the debugger.

The /boot/apps/Metrowerks folder

Of most interest in the */boot/apps* folder is the *Metrowerks* folder. The BeIDE was originally an integrated development environment that was created and distributed by a company named Metrowerks. Be, Inc. has since taken over development and distribution of the BeIDE. Though Be now owns the BeIDE, installation of the environment still ends up in a folder bearing Metrowerks' name.

In the *Metrowerks* folder can be found the BeIDE application itself. The BeIDE is the Be integrated development environment—to develop an application, you launch the BeIDE and then create a new project or open an existing one.

Also in the *Metrowerks* folder are a number of subdirectories that hold various supporting files and tools. The *plugins* folder holds BeIDE plugins that enhance the capabilities of the BeIDE. The *stationery* folder contains the basic stationery used in the creation of a new BeIDE project (stationery being a file that tells the BeIDE which files (such as which libraries) to include, and what compiler and linker settings to use in a new project). The *tools* folder contains the compiler and linker (on the PowerPC version of the BeOS) or links to the compiler and linker (on the Intel version of the BeOS). On the PowerPC version of the BeOS, you'll find a couple of other folders in the Metrowerks folder: the *debugger* folder (which holds the PowerPC debugger, of course) and the *profiling* folder (which holds some PowerPC profiling tools).

The sample-code folder

Included on the BeOS CD-ROM, but not automatically placed on your hard drive during the installation of the BeOS, is the *sample-code* folder. If you elected to have optional items included during the BeOS installation, this folder may be on your hard drive. Otherwise, look in the *optional* folder on the BeOS CD-ROM for the *sample-code* folder and manually copy it to your hard drive.

The *sample-code* folder holds a number of Be-provided projects. Each project, along with the associated project files, is kept in its own folder. A Be application starts out as a number of files, including source code files, header files, and a resource file (I have much more to say about each of these file types throughout this chapter).

Examining an Existing BeIDE Project

The development of a new program entails the creation of a number of files collectively called a *project*. Taking a look at an existing project is a good way to get an overview of the files that make up a project, and is also of benefit in understanding how these same files integrate with one another. Because my intent here is to provide an overview of what a project consists of (as opposed to exploring the useful and exciting things that can be accomplished via the code within the files of a project), I'll stick to staid and familiar ground. On the next several pages I look at the HelloWorld project.

You've certainly encountered a version of the HelloWorld program—regardless of your programming background. The Be incarnation of the HelloWorld application performs as expected—the phrase "Hello, World!" is written to a window. Figure 2-2 shows what is displayed on your screen when the HelloWorld program is launched.

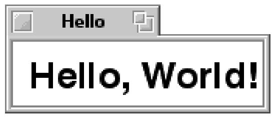


Figure 2-2. The window displayed by the HelloWorld program

You may encounter a number of versions of the HelloWorld project—there's one in the *sample-code* folder, and you may uncover other incarnations on Be CD-ROMs or on the Internet. So that you can follow along with me, you might want to use the version I selected—it's located in its own folder in the Chapter 2 folder of example projects. Figure 2-3 shows the contents of this book's version of the *HelloWorld* folder.

As shown in Figure 2-3, when developing a new application, the general practice is to keep all of the project's files in a single folder. To organize your own projects, you may want to create a new folder with a catchy name such as *myProjects* and store it in the */boot/home* folder—as I've done in Figure 2-3. To

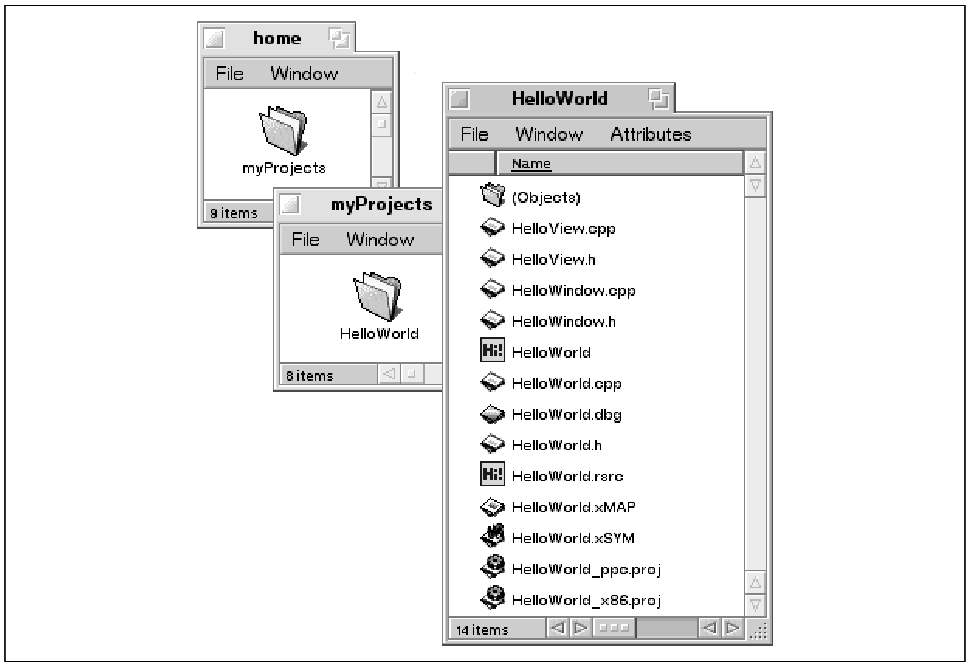


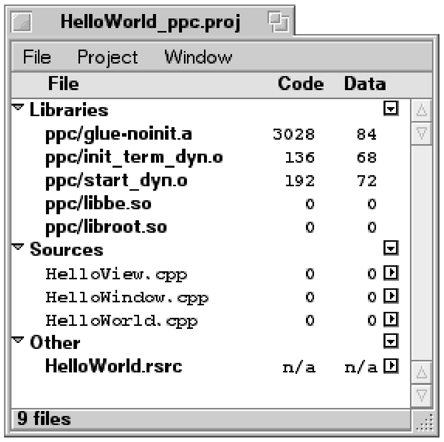
Figure 2-3. The files used in the development of a Be application

begin experimenting, you can copy this book's *HelloWorld* example folder to your own project folder. That way you're sure to preserve the original, working version of this example.

Project File

A Be application developed using the BeIDE starts out as a *project file*. A project file groups and organizes the files that hold the code used for one project. By convention, a project file's name has an extension of *.proj*. It's general practice to give the project file the same name the application will have, with the addition of an underscore and then *ppc* for a PowerPC-based project or an underscore and then *x86* for an Intel-based project. In Figure 2-3, you can see that for the HelloWorld project there are two versions of the project file: *HelloWorld_ppc.proj* and *HelloWorld_x86.proj*.

To open a project file, you can either double-click on its icon or start the BeIDE application and choose Open from the File menu. In either case, select the project that's appropriate for the platform you're working on. When a project file is opened, its contents are displayed in a *project window*. As shown in Figure 2-4, a project window's contents consist of a list of files.



HelloWorld_ppc.proj

File Project Window

File	Code	Data
Libraries		
ppc/glue-noinit.a	3028	84
ppc/init_term_dyn.o	136	68
ppc/start_dyn.o	192	72
ppc/libbe.so	0	0
ppc/libroot.so	0	0
Sources		
HelloView.cpp	0	0
HelloWindow.cpp	0	0
HelloWorld.cpp	0	0
Other		
HelloWorld.rsrc	n/a	n/a

9 files

Figure 2-4. The project window for the PowerPC version of the HelloWorld project

The files listed in a project window are the files to be compiled and linked together to form a single executable. This can be a combination of any number of source code, resource, and library files. The HelloWorld project window holds three source code files and one resource file, each of which is discussed in this chapter. The project window also lists one or more libraries. The number of libraries varies depending on whether you're working on a PowerPC version or an Intel version of a project. Figure 2-4 shows the PowerPC version of the HelloWorld project. In this project, the *glue-noinit.a*, *init_term_dyn.o*, and *start_dyn.o* libraries collectively make up the Be runtime support library that handles the dynamic linking code used by any Be application. An Intel project doesn't list these libraries—they're linked in automatically. The *libroot.so* library handles library management, all of the Kernel Kit, and the standard C library. The *libnet.so* library handles networking, while the *libbe.so* library is a shared library that contains the C++ classes and the global C functions that encompass many of the other kits. An Intel project lists only the *libbe.so* library—the other two libraries are always automatically linked in. The Be kits hold the software that make up much of the BeOS, so this library is a part of the Be operating system rather than a file included with the BeIDE environment.

Library filenames will be prefaced with an indicator as to the project's target platform (the platform on which the resulting application is to run). Figure 2-4 shows a project targeted for the PowerPC (Power Macintosh or BeBox) platform.

Project activity is controlled from the Project menu located in the project window menubar. In Figure 2-5, you see that this menu is used to add files to and remove files from a project. From this menu, you can compile a single file, build an application, and give a built application a test run. In the "Setting Up a New BeIDE Project" section, you'll make use of several of the menu items in the Project menu.

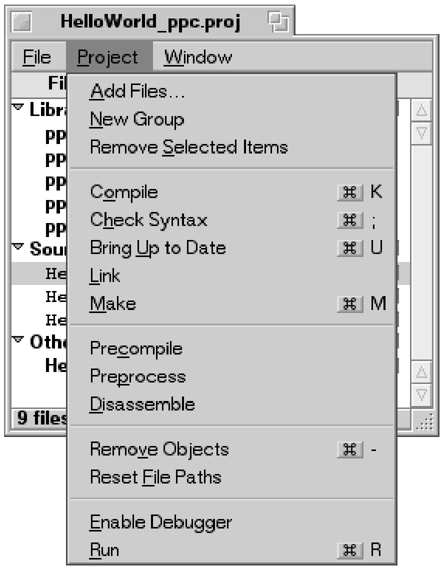


Figure 2-5. The Project menu in the menubar of a BeIDE project window

Of the many items in the Project menu, the Run/Debug item is the most important. Figure 2-5 shows that this bottom menu item is named Run—but this same item can instead take on the name Debug. When the menu item just above this one says Enable Debugger, then the Run/Debug item is in the Run mode. When the menu item just above instead says Disable Debugger, then the Run/Debug item is in the Debug mode. In either case, choosing Run or Debug causes all of the following to happen:

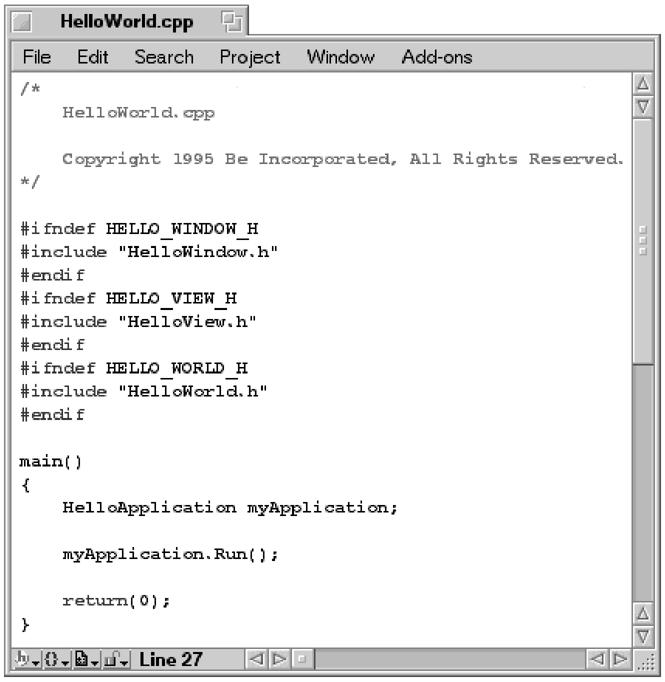
- Compile all project files that have been changed since the last compilation (which may be none, some, or all of the files in the project)
- Link together the resulting object code
- Merge the resource code from any resource files to the linked object code to make (build) an executable (an application)
- Launch the resulting application in order for you to test it (if no compile or link errors occurred)

If the Run/Debug menu is in Debug mode, then the last step in the above list takes place in the debugger. That is, the application is launched in the appropriate debugger (MWDebug-Be for PowerPC projects and bdb for Intel projects). Many of the other items in the Project menu carry out a subset of the operations that are collectively performed by the Run/Debug item.

If you haven't compiled any Be source code yet, go ahead and give it a try now. Open the HelloWorld project file. To avoid the debugger during this first test, make sure the Project menu item just above the Run/Debug item says Enable Debugger (select the item if it doesn't say that). Now choose Run from the Project menu to compile the project's code and run the resulting HelloWorld application.

Source Code and Header Files

The BeOS is a C++ application framework, so your source code will be written in C++ and saved in source code files that have an extension of *.cpp*. To open an existing source code file that is a part of a project, double-click on the file's name in the project window. That's what I did for the *HelloWorld.cpp* file that's part of the HelloWorld project—the result is shown in Figure 2-6.



```
/*
    HelloWorld.cpp

    Copyright 1995 Be Incorporated, All Rights Reserved.
*/

#ifndef HELLO_WINDOW_H
#include "HelloWindow.h"
#endif
#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif
#ifndef HELLO_WORLD_H
#include "HelloWorld.h"
#endif

main()
{
    HelloApplication myApplication;

    myApplication.Run();

    return(0);
}
```

Figure 2-6. The source code window for the HelloWorld.cpp source code file

Most of your code will be kept in source code files. Code that might be common to more than one file may be saved to a header file with an extension of *.h*. While you can keep a project's code in as few or as many source code and header files as desired, you'll want to follow the way Be does things in its examples.

Project file organization convention

Be example projects organize source code into files corresponding to a convention that's common in object-oriented programming. The declaration, or specifier, of an application-defined class exists in its own header file. The definitions, or implementations, of the member functions of this class are saved together in a single source code file. Both the header file and the source code file have the same name as the class, with respective extensions of *.h* and *.cpp*.

There's one notable exception to this naming convention. A project usually includes a header file and source code file with the same name as the project (and thus the same name as the application that will be built from the project). The header file holds the definition of the class derived from the `BApplication` class. The source code file holds the implementations of the member functions of this `BApplication`-derived class, as well as the `main()` function.

File organization and the HelloWorld project

Now let's take a look at the HelloWorld project to see if it follows the above convention. Because this example is based on a project from Be, Inc., you can guess that it does, but you'll want to bear with me just the same. The point here isn't to see if the HelloWorld project follows the described system of organizing files, it's to examine an existing project to clarify the class/file relationship.

Back in Figure 2-4 you saw that the HelloWorld project window displays the names of three source code files: *HelloView.cpp*, *HelloWindow.cpp*, and *HelloWorld.cpp*. While it's not obvious from the project window, there is also a header file that corresponds to each of these source code files (opening a source code file and looking at its `#include` directives reveals that information). According to the previous discussion, you'd expect that the *HelloView.h* file holds a listing for a class named `HelloView`. Here's the code from that file:

```
// ----------------------------------------------------------
// HelloView.h

class HelloView: public BView {

public:
                HelloView(BRect frame, char *name);
virtual  void   AttachedToWindow();
virtual  void   Draw(BRect updateRect);
};
```

Looking at the code in the *HelloView.cpp* file, we'd expect to see the implementations of the three member functions declared in the `HelloView` class definition. And we do:

```
// -----------------------------------------------------------
// HelloView.cpp

#include "HelloView.h"

HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void HelloView::AttachedToWindow()
{
    ...
}


void HelloView::Draw(BRect updateRect)
{
    ...
}
```

As you can see from the *HelloView.cpp* listing, I'm saving a little ink by not showing all of the code in the project's files. Nor do I describe the code I do show. Here I'm only interested in demonstrating the relationship between a project's files and the classes defined by that project. I do, however, take care of both of those omissions at the end of this chapter in the "HelloWorld Source Code" section.

I said I wouldn't discuss the HelloWorld source code here. Out of decency to the very curious, though, I will make a few quick comments. You're familiar with the SimpleApp example that was introduced in Chapter 1, *BeOS Programming Overview*. That example defined two classes. One was named `SimpleWindow` and was derived from the `BWindow` class. It was used to display a window. The second class was named `SimpleApplication` and was derived from the `BApplication` class. Every Be program needs to define such a class. The HelloWorld example discussed here defines similar classes named `HelloWindow` and `HelloApplication`. It also defines a third class named `HelloView`, which is derived from the `BView` class. Before writing or drawing to a window, a program must define a view—an area in the window to which drawing should be directed. The SimpleApp program didn't draw to its window, so it didn't need a class derived from the `BView` class.

The second source code file shown in the project window in Figure 2-4 is *Hello-Window.cpp*. This file has a corresponding header file named *HelloWindow.h*. In this file we expect to find the declaration of a class named `HelloWindow`—and we do:

```
// -----------------------------------------------------------
// HelloWindow.h

class HelloWindow : public BWindow {

public:
               HelloWindow(BRect frame);
virtual  bool  QuitRequested();
};
```

The *HelloWindow.cpp* file contains the source code for the two `HelloWindow` member functions, `HelloWindow()` and `QuitRequested()`:

```
// -----------------------------------------------------------
// HelloWindow.cpp

#include "HelloWindow.h"

HelloWindow::HelloWindow(BRect frame)
    : BWindow(frame, "Hello", B_TITLED_WINDOW,
              B_NOT_RESIZABLE | B_NOT_ZOOMABLE)
{
    ...
}

bool HelloWindow::QuitRequested()
{
    ...
}
```

Earlier I stated that the header file that bears the name of the project should hold the declaration of the project's application class—the class derived from the `BApplication` class. Here you see that the *HelloWorld.h* header file does indeed hold this declaration:

```
// -----------------------------------------------------------
// HelloWorld.h

class HelloApplication : public BApplication {

public:
          HelloApplication();
};
```

The source code file with the name of the project should hold the code for the implementation of the member functions of the application class as well as the `main()` function. *HelloWorld.cpp* does hold the following code.

```
// ----------------------------------------------------------
// HelloWorld.cpp

#include "HelloWindow.h"
#include "HelloView.h"
#include "HelloWorld.h"

int  main()
{
    ...
    ...
}

HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    ...
    ...
}
```

While I omitted the code that makes up the body of each member function of each class in the HelloWorld project, you may still have picked up on similarities between the HelloWorld source code and the source code of the Chapter 1 example, SimpleApp. In the section "HelloWorld Source Code" I point out all of the similarities and discuss the differences.

In looking at existing source code, you may encounter a `BApplication` constructor argument that's four characters between single quotes rather than a long, double-quoted MIME string. The four-character method is the old way of supplying a signature to an application, and is dated. The newer MIME string format is discussed in more detail later in this chapter.

Resources and the Resource File

It's nice to sum up a programming concept in a single sentence, as in "a pointer is a reference to a specific area in memory." Unfortunately, such conciseness isn't always possible. Such is the case with the subject of resources. I'll begin with a short summation—"a resource is code that represents one element of a program"—but adding clarity to that vague explanation necessitates a couple of paragraphs.

The "element of a program" I speak of is usually thought of as one part, or entity, of a program's graphical user interface. For instance, some operating systems make it easy to represent a window or menu as a resource. But a resource doesn't have to represent something graphical. For instance, an application's signature—a short, unique string that helps the operating system differentiate the application from all other applications—is anything but graphical. Yet it can be a resource. While an application's signature isn't graphical in nature, the way in which it can be created and edited can be thought of as graphical. For instance, one could imagine a simple editor that had a Create Application Signature menu item which, when selected, displayed a text box in which a short string was typed. The editor would then be responsible for saving these typed characters as a resource. So it turns out that rather than *representing* something that is itself graphical, a resource is usually something that can be *created and edited* graphically.

Being graphically editable is typically one trait that makes an element a candidate to be represented by a resource. Since some programmer will have to design a special editor that is capable of graphically editing a resource, another requirement is that the element be something common to most or all programs.

You've just read that different program elements exist as resources for a variety of reasons. An application's icon is a good example. First, an icon is a small picture, so it of course is an entity that lends itself to being easily edited graphically. Second, all applications have an icon that is used to represent the application on the desktop, so it made sense for someone to expend the effort to create an editor capable of editing icons. Finally, the BeOS needs the data that defines an application's icon even when the application isn't running, so that it can display the icon on the desktop at all times.

There are different types of resources, and the BeOS keeps track of these different types by using a different 32-bit integer for each resource type. As a convenience to programmers, a four-character constant is often used to define this integer. Consider the icon that represents an application on the desktop. The data that defines this icon exists as a resource, and its type is 'ICON.' Most programmers find it easier to remember the four-character constant 'ICON' than the numerical value this constant represents.

While a resource type is surrounded in single quotes in this book and in Be documentation as well, the quotes aren't a part of the resource type—a resource type is simply the four characters (or an actual 32-bit numerical value). The quotes are used only to make it obvious that a resource type is being discussed. This is important because a resource type can be in lowercase, and it can include a space or spaces. Placing an icon type in quotes sets it apart from the rest of the text that appears with it.

Application-information resource

There's one other reason that a certain part of a program will exist as a resource—a reason unrelated to the graphical nature of the element or its ability to be edited graphically. Because of the way in which resources are stored in an executable, resource information is available to the BeOS even when the application isn't running. The BeOS needs some information about an application in order to be able to effectively communicate with it. This information can be kept together in a single resource of type 'APPI' (for "application information") in the application. An 'APPI' resource consists of the following pieces of information about an application:

Launch Behavior
: The launch characteristics of a Be application can fall into one of three categories. *Single launch* is the typical behavior—no matter how many times a user double-clicks on the application's icon, only one instance of the executable is loaded into memory and executed (that is, double-clicking on an application's icon a second time has no effect). It's possible for two versions of a single launch application to end up in memory if the user makes a duplicate of the original executable and then double-clicks on each. *Exclusive launch* is a behavior that restricts this from occurring. Under no circumstance can two versions of a program execute at the same time. *Multiple launch* is a behavior that allows any number of instances of a single copy of a program to execute simultaneously.

Background App
: An application can forego a user interface and run in the background only. If an application is marked as a background app, it behaves in this way and won't be named in the Deskbar.

Argv Only
: An application can be excluded from receiving messages from the BeOS (refer to Chapter 1 for an introduction to messages). Marking an application as *argv only* means that the only information the application receives comes from the `argc` and `argv` command-line arguments that can be optionally passed to the program's `main()` routine.

Signature
: Each application has a string that lets the BeOS view the application as unique from all others. Obviously, no two applications should share the same signature. For your own test programs, the signature you choose isn't too important. Should you decide to distribute one of your applications to the Be community, though, you'll want to put a little effort into selecting a signature. Be's recommended format is "application/x-vnd.VendorName-ApplicationName".

Replacing VendorName with your company's name should provide a unique signature for your application.

Here I'll look at 'APPI' information for an existing project that already includes a resource file. In this chapter's "Setting Up a New BeIDE Project" section you'll find information on creating a resource file and editing its 'APPI' information. To view the 'APPI' information in a project's resource file, double-click on its name in the project window. That launches the FileTypes application (which can also be launched by choosing it from the preferences folder in the Be menu) and opens two windows. Figure 2-7 shows the main window of FileTypes.

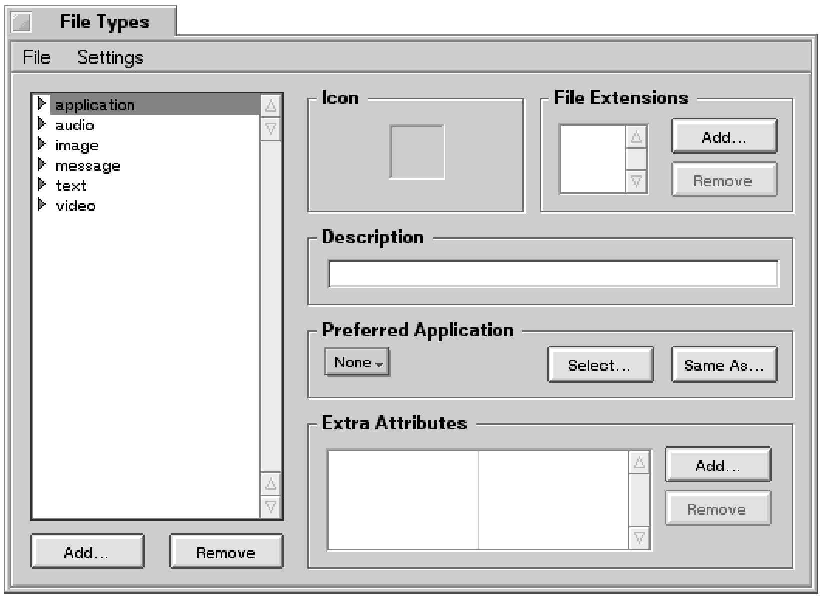
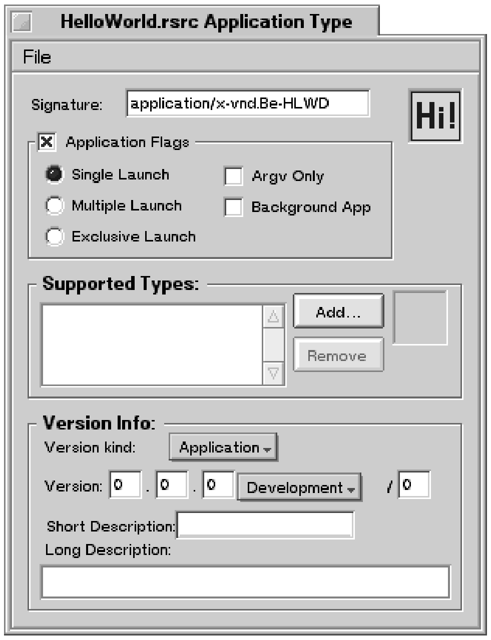


Figure 2-7. The main FileTypes window

To view or edit an application's 'APPI' resource information, work in the second of FileTypes' two windows. Figure 2-8 shows this window for the HelloWorld application.

The application's launch behavior is determined based on which of the three Launch radio buttons is on—Single Launch, Multiple Launch, or Exclusive Launch (only one can be on at any given time).

Whether or not the application is a background app is determined by the status of the Background App checkbox. Whether or not the application is capable of receiving messages is determined by the status of the Argv Only checkbox. While these two items appear grouped together in the Application Flags area, they aren't related. Neither, either, or both can be checked at the same time.



HelloWorld.rsrc Application Type

File

Signature: application/x-vnd.Be-HLWD

Hi!

[✓] Application Flags

● Single Launch ☐ Argv Only

○ Multiple Launch ☐ Background App

○ Exclusive Launch

Supported Types:

Add...

Remove

Version Info:

Version kind: Application

Version: 0 . 0 . 0 Development / 0

Short Description:

Long Description:

Figure 2-8. Viewing the 'APPI' information for the HelloWorld application

An application's signature is based on the MIME string you enter in the signature edit box of the FileTypes window. If a signature appears here, the string passed to the `BApplication` constructor will be ignored (refer to Chapter 1). If no signature appears here, the string passed to the `BApplication` constructor will be used. Thus, by entering a string in the FileTypes window, you're making the `BApplication` constructor argument immaterial. Figure 2-8 shows the signature for the HelloWorld application used throughout this chapter.

If you make any changes to a project's resource file, save them by choosing Save from the File menu of FileTypes (the File menu's other item, Save into Resource File, is discussed in the "Setting Up a New BeIDE Project" section of this chapter).

Icon resource

An icon could be described within source code (and, in the "old days," that was in fact how icons were described), but the specification of individual pixel colors in source code is difficult and tedious work. Rather than attempting to specify the colors of each pixel of an icon from within source code, a BeOS application's icon can be created using a special graphical editor built into the FileTypes application.

The graphical editor in FileTypes is used in a manner similar to the way you use a graphics paint program—you select a color from a palette of colors and then use a pencil tool to click on individual pixels to designate that they take on that color. See Figure 2-9 to get an idea of what the graphical editor in FileTypes looks like.

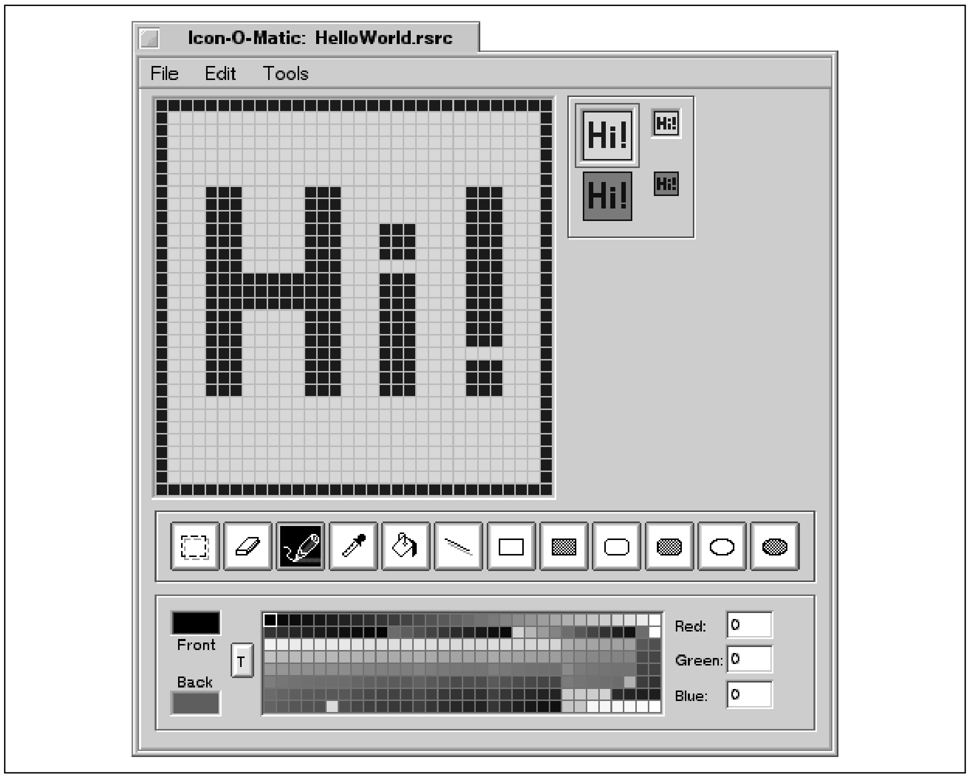


Figure 2-9. The icon editing window and palettes displayed by FileTypes

In FileTypes you simply draw the icon. You can then save the icon to a project's resource file so that each time an application is built from the project, your icon is merged with the application (and becomes the icon viewed on the desktop by the user). To view or edit the icon stored in the resource file of an existing project, you first double-click on the resource filename in the project window to open the resource file. After FileTypes opens the resource file, double-click on the small icon box located at the upper right of the FileTypes window; you'll see the window shown in Figure 2-9.

Setting Up a New BeIDE Project

In the previous section, you read that an application starts as a Be project. The Be project consists of a project file, source code files, header files, libraries, and a resource file. The project file itself doesn't hold any code; it serves as a means to organize all the other files in the project. The project file also serves as the project "command center" from which you compile code and build and test the executable. A close look at the HelloWorld project clarified many of these concepts.

When you set out to develop your own application, you'll find that you may be able to save some effort if you don't start from scratch, but instead duplicate a folder that holds the files of an existing project. Consider this scenario: I want to create a very simple children's game—perhaps a tic-tac-toe game. I know that the HelloWorld project results in a program that displays a single window and draws to it. That represents a good part of what my game will need to do, so it makes sense for me to base my game on the HelloWorld project, and then modify and add to the HelloWorld source code as needed. If your program will be a complex one, or one for which you can't find a similar "base" program to start with, this approach won't be as fruitful. In such cases you'll want to start with a new project.

In this section, I'll discuss each step of the process of setting up a new project first in general terms. I'll also carry out each step using the HelloWorld project to provide a specific example. So you see, I had good reason for devoting the previous several pages to a look at the HelloWorld project. While I use the small HelloWorld project for simplicity, the steps discussed on the following pages apply just as well to projects of a much larger scale.

In the above paragraphs, I refer to using code written by others. Before doing that you'll of course want to make sure that you're allowed to do so! The BeOS CD-ROM comes with a number of example projects that fall into the category of projects that are available for your own personal use. The source code that makes up the example projects is copyright Be, Inc., but Be, Inc. has granted unrestricted permission for anyone to use and alter any of this source code. I've taken advantage of this fact and used these projects as the basis for the numerous examples that appear in this book. In turn, you're free to use without restrictions the example code in this book for your own projects.

The following is an overview of the major steps you'll carry out each time you create a new project. While on the surface it may appear that performing these steps involves a lot of work, you'll find that after you've set up a few new projects the process becomes quite routine, and doesn't take much time at all. All of the steps are discussed in the sections that follow this list.

1. Find an existing project that is used to build a program that has similarities to the program you're to develop.
2. Duplicate the existing project folder and its contents.

3. Open the new project folder and change the names of the project, source code, header, and resource files to names that reflect the nature of the project you're working on.
4. Open the newly renamed project and drag the renamed source code files and resource file from the folder and drop them into the project window.
5. Remove the now obsolete source code and resource filenames from the project window.
6. Edit the name of the constants in the `#ifndef` directive in the header files and the `#includes` in the source files.
7. Test the project's code by building an application (here you're verifying that the original source code is error-free before you start modifying it)
8. If there are library-related errors, create a new project (which will automatically include the most recent versions of each library) and add the source code files and resource file to the new project.
9. If there are compilation errors, correct the source code that caused the errors.
10. Open the header files and change application-defined class names in the header files to names that make sense for the project you're working on.
11. Change all usage of application-defined class names in the source files to match the changes you made in the header files.
12. Open the resource file using FileTypes and modify any of the 'APPI' resource information and the icon.
13. Set the name for the executable to be built.
14. Build a new application from the modified BeIDE project.

No new functionality will have been added to the program that gets built from the new project—it will behave identically to the program that results from the original project. So why go through the above busy-work? Executing the above steps results in a new project that includes source code files that define and use classes with new names—names that make sense to you. This will be beneficial when you start the real work—implementing the functionality your new program requires.

The above list isn't an iron-clad set of steps you must follow. Other programmers have their own slightly (or, perhaps, very) different guidelines they follow when starting a new project. If you're new to the BeIDE, the BeOS, or both, though, you might want to follow my steps now. As you get comfortable with working in a project-based programming environment, you can vary the steps to match your preferred way of doing things.

Selecting and Setting Up a Base Project

As mentioned, you'll get off to the best start in your programming endeavor by finding a project that matches the following criteria:

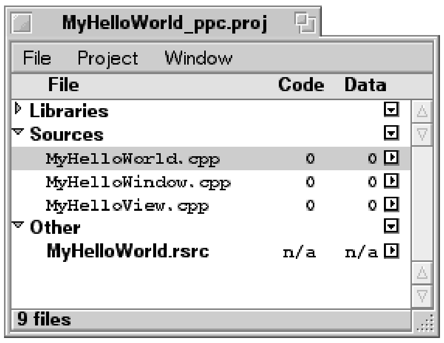
- The application that is built from the original project has several features common to the program you're to develop.
- You have access to the project and all its source code and resource files.
- It's made clear that the project's source code can be modified and redistributed, or you have the developer's permission to do so.

Once you've found a project that meets the above conditions, you've performed Step 1 from the previous section's numbered list.

Step 2 involves creating a copy of the project folder and its contents. After doing that, rename the new folder to something appropriate for the project you're embarking upon. Usually a project folder has the same name that the final application that gets built from the project will have. Here I'm making a new project based on the HelloWorld project only for the sake of providing a specific example, so after duplicating the HelloWorld folder, I'll simply change the name of the folder from *HelloWorld copy* to *MyHelloWorld* (each of these folders can be found in the Chapter 2 examples folder available on the O'Reilly web site).

Step 3 is the renaming of the project-related files. Double-click on the new folder to reveal its contents. Click on the name of any one of the header files and type a new name for the file. For my new MyHelloWorld project I'll rename the *HelloView.h*, *HelloWindow.h*, and *HelloWorld.h* header files to *MyHelloView.h*, *MyHelloWindow.h*, and *MyHelloWorld.h*, respectively. Next, rename the source code files (so, for example, *HelloWorld.cpp* becomes *MyHelloWorld.cpp*) and the resource file (here, from *HelloWorld.rsrc* to *MyHelloWorld.rsrc*). Now rename the project file. Again, choose a name appropriate to the project. Typically, the project file has the same name the application will have, with an extension of *x86.proj* or *ppc.proj* added. I'll change the PowerPC version of the HelloWorld project by updating the project filename from *HelloWorld_ppc.proj* to *MyHelloWorld_ppc.proj*.

Steps 4 and 5 are performed to get the project to recognize the newly named files. After the name changes are made, double-click on the project file to open it. The project window will appear on your screen. If you renamed a file from the desktop, the project file that includes that file will list it by its original, now invalid, name. That necessitates adding the file by its new name and removing the original file from the project. To add the newly named files, select them from the desktop (click on each) and drag and drop them into the project window. In the project window, drag each to its appropriate group (for instance, place *MyHelloWorld.cpp* in the Sources group). To remove the original files from the project, select each and choose Remove Selected Items from the Project menu. For the MyHelloWorld project, the resulting project window looks like the one shown in Figure 2-10.



File	Code	Data
Libraries		
Sources		
MyHelloWorld.cpp	0	0
MyHelloWindow.cpp	0	0
MyHelloView.cpp	0	0
Other		
MyHelloWorld.rsrc	n/a	n/a

9 files

Figure 2-10. The MyHelloWorld project window with renamed files in it

Testing the Base Project

The new project now has newly named files in it, but these files hold the code from the original project. Before adding new functionality to the code, verify that it compiles without error (this is Step 7 from the previous list). Before compiling the code, though, perform Step 6—update the `#includes` at the top of each source code file so that they match the new names of the header files. For example, near the top of *MyHelloView.cpp* is this `#include`:

```
#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif
```

I've changed the *HelloView.h* header file to *MyHelloView.h*, so this `#include` needs to be edited. While I'm doing that, I'll change `HELLO_VIEW_H` to the more appropriate `MY_HELLO_VIEW_H` (though I could leave this as is—it's simply a constant I define in the *MyHelloView.h* header file).

```
#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif
```

Because I changed the name of the constant `HELLO_VIEW_H` in the *MyHelloView.cpp* source file, I need to change this same constant in the *MyHelloView.h* header file. Originally the header file contained this code:

```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H
```

Now that code should look like this:

```
#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H
```

Finally, test the code by choosing Run from the Project menu. Later, if you experience compilation errors after you've introduced changes to the original code, you'll know that the errors are a direct result of your changes and not related to problems with the original code.

If the building of an application is successful, Steps 8 and 9 are skipped. If the attempt to build an application results in library-related errors (such as a library "file not found" type of error), you're probably working with a project created under a previous version of the BeIDE. The easiest way to get the proper libraries into a project is to follow Step 8—create a new project based on one of the Be-supplied project stationeries. A new BeIDE project file can be created by choosing New Project from the File menu of an existing project. When you do that, the New Project dialog box appears to let you choose a project stationery from which the new project is to be based. The project stationery is nothing more than a template that specifies which libraries and project settings are best suited for the type of project you're creating. Here are definitions of the more important stationeries:

BeApp
: Stationery that links against the standard libraries applications need.

EverythingApp
: Stationery that links against all of the Be libraries.

KernelDriver
: A basic template for writing a kernel driver.

SharedLib
: Stationery used to create a basic library or add-on, this links against the basic Be libraries.

BeSTL
: Stationery used to create an application that includes the C++ standard libraries (including STL and the basic iostream functions).

In Figure 2-11, I'm choosing the *BeApp* project stationery under the *ppc* heading in order to create a project that's set up to generate a Be application targeted for the PowerPC platform (the list of project stationeries you see may differ depending on the version of the BeOS you're using and the processor (PowerPC or Intel) in your machine). Note that when creating a new project you'll want to uncheck the Create Folder checkbox in the New Project dialog box and specify that the new project end up in the folder that holds all the renamed files.

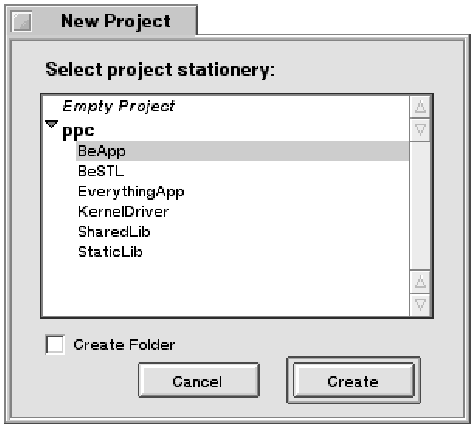


Figure 2-11. Choosing a stationery on which to base a new project

The act of creating a new project doesn't provide you with any source code files or a resource file—you'll need to repeat Step 4. That is, drag and drop the necessary files from the desktop folder to the new project window.

If the building of an application results in compilation errors, now's the time to correct them. This is Step 9. Only after you successfully build an application does it make sense to start making changes to the project's source code.

Preliminary Code Changes

You'll of course be making changes to the source code in the source code files. The most interesting of these changes will be the ones that turn the original code into code that results in an application that is distinctly your own. First, however, you need to make some preliminary source code changes. You've changed the names of the files, including the header files, so you'll need to search for and change header file references in the source code files. You'll also want to change the names of the classes already defined to match the names you've given to the header files.

Header file code changes

Step 10 in the list of new project setup tasks is the changing of application-defined class names in the header files. Begin the changes by opening any one of the header files. The quickest way to do that is to click on the small arrow icon to the right of one of the source code filenames in the project window. Doing that displays a menu that lists the header files included in the selected source code file. To open a header file, simply select it from this popup menu. With a header file open, make the following changes to the code.

- Add the new name of the file to the file's description comment section.
- If you haven't already done so, rename the application-defined constant that is used in the `#ifndef` and `#define` preprocessor directives.
- Rename the file's application-defined class.
- Rename the class constructor to match the new name you've give to the application-defined class.

The above steps are carried out in the same way regardless of the project you start with. To provide a specific example of how these changes are implemented, I'll change the *HelloView.h* header file from the HelloWorld project. The following listing shows the original version of the *HelloView.h* file. Refer to it in the discussion that follows the listing.

```
// --------------------------------------------------------------
    HelloView.h
    Copyright 1995 Be Incorporated, All Rights Reserved.

#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H

#ifndef _VIEW_H
#include <View.h>
#endif

class HelloView : public BView {

public:
               HelloView(BRect frame, char *name);
virtual  void  AttachedToWindow();
virtual  void  Draw(BRect updateRect);
};

#endif

// --------------------------------------------------------------
```

The first change to the header file, the altering of the file's descriptive comment, needs little discussion. You may want to leave the original name intact as a courtesy to the original author. I've done that in the *MyHelloView.h* file (the listing of which follows this discussion).

The `HELLO_VIEW_H` constant is defined to eliminate the possibility of the code in this header file being included more than once in the same source code file. Earlier I changed its name to `MY_HELLO_VIEW_H` to reflect the new name I've given to this header file (*MyHelloView.h*):

```
#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H
```

The `_VIEW_H` constant is a BeOS-defined constant (it's used to ensure that the BeOS header file *View.h* doesn't get included multiple times) so it can be left as is. If you aren't clear on the usage of the `#ifndef` preprocessor directive in Be files, refer to the "Header Files and Preprocessor Directives" sidebar.

The original class defined in the *HelloView.h* file was named `HelloView`. I've renamed that class to `MyHelloView` to match the name I've given the header file. I also changed the name of the class constructor from `HelloView()` to `MyHelloView()`. All of lines of code that include changes are shown in bold in the following listing of *MyHelloView.h*:

```
// ----------------------------------------------------------
    MyHelloView.h
    Dan Parks Sydow
    1999

    Based on source code from:
    HelloView.h
    Copyright 1995 Be Incorporated, All Rights Reserved.
// ----------------------------------------------------------

#ifndef MY_HELLO_VIEW_H
#define MY_HELLO_VIEW_H

#ifndef _VIEW_H
#include <View.h>
#endif

class MyHelloView : public BView {

public:
                MyHelloView(BRect frame, char *name);
virtual  void  AttachedToWindow();
virtual  void  Draw(BRect updateRect);
};

#endif

// ----------------------------------------------------------
```

Notice that I've stopped short of making any substantial changes to the application-defined class. While you may be tempted to "get going" and start adding new member functions to a class, it's best to wait. First, make all the name changes in the header files. Then update the source code files so that any usage of application-defined classes reflects the new names (that's discussed next). Finally, verify that everything works by compiling the code. Only after you're satisfied that all the preliminaries are taken care of will you want to start making significant changes to the code.

For the MyHelloWorld project, I'd repeat the above process on the *MyHelloWindow.h* and *MyHelloWorld.h* header files.

Header Files and Preprocessor Directives

In a large project that consists of numerous source code files and header files, the potential for a source code file to include the same header file more than once exists. Consider three files from a large project: source file *S1.cpp* and header files *H1.h* and *H2.h*. If *S1.cpp* includes both *H1.h* and *H2.h*, and *H1.h* includes *H2.h*, then *S1.cpp* will include the code from *H2.h* twice (once directly and once indirectly via *H1.h*). Such an event results in a compilation error that, given the dependency of the files, can be difficult to remedy.

To coordinate the inclusion of header file code in source code files, Be projects typically use the "if not defined" (`#ifndef`) preprocessor conditional directive to define a constant in a header file and to check for the definition of that constant in a source code file. This is a standard technique used by many C and C++ programmers—if you're familiar with it, you can feel free to skip the remainder of this sidebar.

A header file in a Be project begins by checking to see if a particular constant is defined. If it isn't defined, the header file defines it. Because all the remaining code in the header file lies between the `#ifndef` and the `#endif` directives, if the constant is already defined, the entire contents of the header file are skipped. Here's that code from the HelloWorld project's *HelloView.h* header file:

```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H
// class declaration
#endif
```

Before including a header file, a source code file checks to see if the constant defined in the header file is in fact defined. If it isn't defined, the source code file does include the header file. Here's that code from the *HelloView.cpp* source code file:

```
#ifndef HELLO_VIEW_H
#include "HelloView.h"
```

#endif source code file code changes

After you make the changes to the header files, it's time to update the source code files—this is Step 10 in the list of new project setup steps. Begin by double-clicking on one of the source code filenames in the project window. Then make the following changes to the code:

- Add the new name of the file to the file's description comment section.
- Rename the application-defined constant that is used in the `#ifndef` and `#define` preprocessor directives to match the renamed constant in the corresponding header file.
- Rename any header filenames that follow `#include` directives to match the renamed header files.
- Rename all occurrences of any application-defined classes to match the renamed classes in the other project header files.

As with header files changes, the source code file changes listed above apply to any project. Again, I'll use a file from the HelloWorld project to provide a specific example. The following listing shows the original version of the *HelloView.cpp* file:

```
// ----------------------------------------------------------
    HelloView.cpp
    Copyright 1995 Be Incorporated, All Rights Reserved.
// ----------------------------------------------------------

#ifndef HELLO_VIEW_H
#include "HelloView.h"
#endif

HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void HelloView::AttachedToWindow()
{
    ...
}


void HelloView::Draw(BRect updateRect)
{
    ...
}

// ----------------------------------------------------------
```

Next you'll see the edited version of *HelloView.cpp*—it's now the code for a file named *MyHelloView.cpp*. Because there are no occurrences of any application-defined class names in the code in the bodies of any of the member functions, I've omitted that code for brevity.

```
// ----------------------------------------------------------
    MyHelloView.cpp
    Dan Parks Sydow
    1999

    Based on source code from:
    HelloView.cpp
    Copyright 1995 Be Incorporated, All Rights Reserved.
// -----------------------------------------------------------

#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif

MyHelloView::MyHelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
    ...
}


void MyHelloView::AttachedToWindow()
{
    ...
}


void MyHelloView::Draw(BRect updateRect)
{
    ...
}

// -----------------------------------------------------------
```

These steps need to be repeated for each source code file in the project. As you do that, note that a source code file may include occurrences of application-defined classes other than the class defined in the corresponding header file. Consider this snippet from the original *HelloWorld.cpp* file:

```
HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    HelloWindow  *aWindow;
    HelloView    *aView;
    BRect         aRect;
```

Here you see that the constructor for the application-defined `HelloApplication` class declares variables of two different application-defined classes: `HelloWindow` and `HelloView`. After renaming the *HelloWorld.cpp* file to *MyHelloWorld.cpp*, the changes to the above snippet would turn it into code that looks like this:

```
MyHelloApplication::MyHelloApplication()
    : BApplication("application/x-vnd.dps-myworld")
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;
```

The astute reader will have noticed that I slipped an extra change in the above snippet. If you didn't spot it, look at the parameter to the `BApplication()` constructor. Later in this chapter you'll see that I also use the FileTypes application to place this same signature in the project's resource file.

A quicker way to make the changes

You may have found all this talk about manually editing header files and source code files a little disconcerting. Before your level of frustration rises too high, it's time for me to make a confession. There is a shortcut to poring over page after page of code to track down occurrences of class names that need to be changed—a shortcut I didn't mention at the onset of the "Preliminary Code Changes" section. The BeIDE has a search and replace utility that makes finding and replacing names (such as constants and class names) a snap.

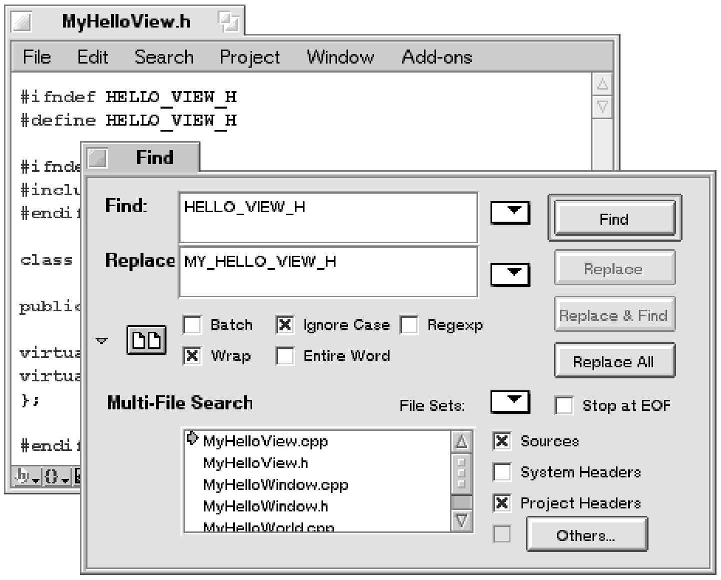
Before discussing how to quickly make global changes to all of the files in a project, I'll answer the one question that's on your mind: Why did I bother with the drawn-out explanation of exactly what changes to make if everything could be done with a few mouse clicks? I went laboriously through all the changes so that you'd know what you're changing and why.

Now, here's a revised method for changing the names of constants, files, and classes that appear in the header files and source code files of a project:

1. Choose Open from the BeIDE main menu in the dock and open a header file.
2. Choose Find from the Search menu of the opened header file.
3. Set up the Find window to search the project's source code files and header files.
4. Enter a name to change in the Find box and the name to replace it with in the Replace box of the Find window.
5. Click on the Find button in the Find window.
6. After verifying that the found text should indeed be replaced, click on the Replace & Find button in the Find window.
7. Repeat Step 6 until all occurrences of the name have been found and changed.
8. Repeat Steps 4 though 8 for each name to be changed.

You can speed things up by using the Replace All button once in place of the Replace & Find button several times. However, it's safer to use the Replace & Find button so that you can take a quick look at each name the BeIDE is changing.

If you want to perform a multiple-file search, the file icon located on the left side of the Find window must be displaying two files and the Find window must be expanded as shown in Figure 2-12. If the file icon is displaying a single file, click on the icon (it toggles between one and two files). If the Find window isn't expanded (the Multi-File Search section won't be visible), click on the small blue arrow located to the left of the file icon (the arrow expands and collapses the Find window).



```
#ifndef HELLO_VIEW_H
#define HELLO_VIEW_H

#ifnde
#inclu
#endi

class

publi

virtua
virtua
};

#endi
```

Find: HELLO_VIEW_H
Replace MY_HELLO_VIEW_H

☐ Batch ☒ Ignore Case ☐ Regexp
☒ Wrap ☐ Entire Word

Multi-File Search File Sets: ☐ Stop at EOF

MyHelloView.cpp
MyHelloView.h
MyHelloWindow.cpp
MyHelloWindow.h
MyHelloWorld.cpp

☒ Sources
☐ System Headers
☒ Project Headers

Figure 2-12. The BeIDE Find window set up to search all of a project's files

With the Find window set for multiple-file searching, specify which files are to be included in a search. Check the Sources checkbox to designate that all of the source code files listed in the project window are to be searched. To specify that the header files also be included in the search, you'll be tempted to check the Project Headers checkbox. At this point, however, the BeIDE doesn't know that the project source code files will be including the renamed header files—so it won't know to add the desired files to the Multi-File Search list. Instead, click on the Others button. In the Select a File to Search window that appears, Shift-click on the names of each of the renamed header files. That is, hold down the Shift key and successively click the mouse button while the cursor is over each header filename in the window's list of files. Click the Add button, then the Done button.

To quickly change a name, enter the name to change in the Find box, enter the new name in the Replace box, and click the Replace All button. The BeIDE will search all the project files for the name to find and replace each occurrence with the new name. For my MyHelloWorld project I began by searching for the constant `HELLO_VIEW_H` and replacing it with the new constant `MY_HELLO_VIEW_H`.

You can see in Figure 2-12 that the Find window is set up to perform that act. When done (it will only take a moment), enter another name and another new name in the Find and Replace boxes and again click on the Replace All button. Repeat these steps until all of the necessary changes have been made.

A click on the Replace All button quickly searches all of the files in a project and replaces all hits with new text. That makes the BeIDE Find facility a very powerful utility—*too* powerful if you aren't sure what you're changing. Hence my reasoning for describing this technique after the discussion on just what needs to be changed in the files of a project. Now, if you make a mistake that results in an error when you compile the project's code, you'll know where to look and what to look for to remedy the errors.

Testing the changes

After making all the changes to a project, compile the code by choosing Make from the Project menu in the project window's menubar. This menu item compiles all *touched* files and then builds a new version of the application. A touched file is one that has been edited since the last time the file was compiled. Before moving on to the "real" code changes—the possibly extensive changes and additions you'll be making to implement your program's functionality—you'll want to verify that the "cosmetic" changes you've just made didn't introduce any errors.

Editing the Resource File

You'll want to give your application a unique signature and its own icon. The 'APPI' and 'ICON' resources are both located in the project's resource file, so the FileTypes application is involved in both these acts.

Changing the signature

To edit the signature to be merged with the application built from the new project, double-click on project's resource file. This launches the FileTypes application and opens a window that holds 'APPI' information (refer back to Figure 2-8 for an example). To complete Step 12 of the new project setup process, modify the 'APPI' information as needed. Most likely this will involve nothing more than entering a new, unique signature to be assigned to the application that makes use of the resource file. For the MyHelloWorld project, I changed the signature in the *MyHelloWorld.rsrc* file to `application/x-vnd.dps-myworld`.

After changing the signature, you can choose Save from the File menu of the FileTypes window to save the change—but don't close the window just yet.

Changing the icon

To modify the icon housed in the resource file, double-click on the icon box located in the upper right corner of the Application Type window. Then draw away. That, in brief, is the rest of Step 12.

The Tracker—the software that allows you to work with files and launch applications in the BeOS—displays icons in one of two sizes: 32 pixels by 32 pixels or 16 pixels by 16 pixels. The larger size is the more commonly used, but you've seen the smaller size on occasion. For example, the active application displays its smaller icon in the area you click on to reveal the items in the main menu. For each icon resource, you'll draw both the larger and smaller variants. At the top right of the window back in Figure 2-9, you'll see the editing area for both variants of the icon in the *HelloWorld.rsrc* file. Clicking on either icon displays an enlarged, editable view of that icon in the editing area.

In the top right area of the window, you'll see a total of four actual-size icons. You can refer to these views to see how the 32-by-32 version and the 16-by-16 version will look on the desktop. There are two views of each sized icon to demonstrate how each looks normally and when selected (a selected icon has extra shading added by the BeOS).

You'll treat the large and small versions of an icon separately—editing one version has no effect on the other version. IconWorld will assign the larger version a resource type of 'ICON' and the smaller version a resource type of 'MICN' (for "mini-icon"). Because each icon has both a large and small version, IconWorld will save the two versions of the icon as a single unit.

You'll of course want to replace the original icon with one of your own creation. To do that, use the drawing tools and color palette to erase the old icon and draw a new one. Creating an icon in FileTypes is a lot like creating a small picture in a paint program, so you're on familiar ground here. If you aren't artistically inclined, at least draw a simple icon that distinguishes your program from the other program icons on your desktop. When done, click on the smaller editing area and then draw a smaller version of the icon. That's what I did for the MyHelloWorld example I'm working on. I show off my creative abilities back in Figure 2-9.

If you're developing a program that will be distributed to others (rather than one that will be for your personal use only), you can find or hire someone else to draw a 32-pixel-by-32-pixel and a 16-pixel-by-16-pixel picture for you at a later time.

When you're done with the new icon, choose Save from the File menu of the Icon-O-Matic window, then close the Icon-O-Matic window. Now choose Save from the File menu of the FileTypes window. Close the FileTypes window. This completes Step 12.

Setting Project Preferences

Besides keeping track of the files that hold what will become a program's executable code, a project also keeps track of a variety of project settings. The font in which source code is displayed and the types of compilation errors to be displayed are just two examples of project settings. There's a good chance that most of the copied project's settings will be fine just as they are. You will, however, want to make one quick change. Step 13 specifies that you need to set the name of the application about to be built—you do that in the Settings window.

To display the project Settings window, choose Settings from the Window menu in the project file. On the left side of the window that appears is a list that holds a variety of settings categories. Click on the xxx Project item under the Project heading in this list (where xxx specifies the target, such as PPC for PowerPC). The Settings window responds by displaying several project-related items on the right side of the window.

From the displayed items you'll set the name that will be assigned to the application that gets built from the project. The MyHelloWorld project is used to build a program that gets the name MyHelloWorld. Note that while the file type may look like a program signature (they're both MIME strings), the two aren't the same. The file type specifies the general type of file being created, such as a Be application. The file signature (assigned in the FileTypes window, as discussed earlier) gives an application a unique signature that differentiates the file from all other applications.

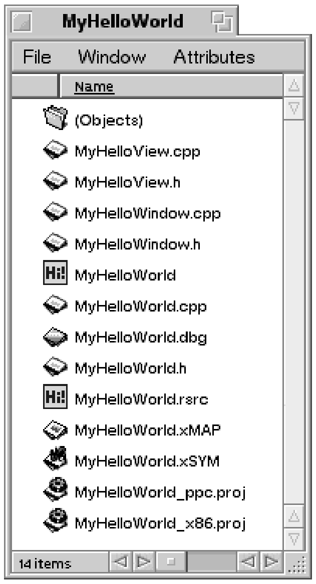
After changing the information in the File Name field, click the Save button. This is the only setting change you need to make, though you're of course free to explore the Settings window by clicking on any of the other setting topics in the list.

Testing the Changes

After making all your changes, you'll want to test things out—that's the last step in my list of how to create a new project. Choose Run from the Project menu in the project window's menubar. This will cause the BeIDE to:

1. Compile any files that have been touched (altered) since the last build.
2. Link the compiled code to build an application.
3. Merge resources from the resource file with the built application.
4. Run the application.

To verify that your changes to the resource file were noted by the BeIDE, return to the desktop and look in your project folder. There you should find your new application, complete with its own icon. Figure 2-13 shows the *MyHelloWorld* folder as it looks after my test build of a PowerPC version of the program.



(Objects)
MyHelloView.cpp
MyHelloView.h
MyHelloWindow.cpp
MyHelloWindow.h
MyHelloWorld
MyHelloWorld.cpp
MyHelloWorld.dbg
MyHelloWorld.h
MyHelloWorld.rsrc
MyHelloWorld.xMAP
MyHelloWorld.xSYM
MyHelloWorld_ppc.proj
MyHelloWorld_x86.proj
14 items

Figure 2-13. The MyHelloWorld project folder after building an application

The *MyHelloWorld* folder in Figure 2-13 shows one folder and two files that I had no hand in creating. The *(Objects)* folder holds, obviously enough, the object code that the BeIDE generated when compiling the project's source code. The *.xMAP* and *.xSYM* files were generated by the BeIDE during the building of the project's application. The files in the *(Objects)* folder and the *.xMAP* and *.xSYM* files aren't directly of use to you—they're used by the BeIDE during linking and debugging. If I were working on a PC and did a build of an Intel version of the *MyHelloWorld* program, my project folder would look a little different. There'd be no *.xMAP* or *.xSYM* files, as these are PowerPC-only. Instead, the information in these files (namely the symbol table and the debugging information for the application) would be contained in the ELF binary itself. There might also be a number of *.d* files, which are temporary dependency files the compilers create (and which may be better disposed of in a future release of the BeOS).

What's Next?

After reading this section you know how to create a project that's ready for your own use. Now you need to make the source code changes and additions that will cut the ties from the original project on which you're basing your new program. So you need to know a lot more about writing BeOS code. The next section starts you in that direction, and the remainder of this book takes you the rest of the way.

HelloWorld Source Code

In the previous section, you saw that a new BeIDE project is usually based on an existing project. Once you have a mastery of BeOS programming, you'll be able to look at existing projects and recognize which one or ones result in a program that bears some similarity to the program you intend to develop. Until that time, it makes sense to use a small project such as the HelloWorld project as your starting point. If you follow my advice and do that, your new project will hold the HelloWorld source code. You got a glimpse of some of that code earlier in this chapter. Because you'll be modifying the HelloWorld source code, you'll find it beneficial to have a good understanding of that code. This section provides you with that.

The HelloWorld/SimpleApp/MyHelloWorld Connection

The HelloWorld project defines three classes: `HelloView`, `HelloWindow`, and `HelloApplication`. Two of these classes, `HelloWindow` and `Hello-Application`, bear a strong resemblance to the `SimpleWindow` and `Simple-Application` classes from the SimpleApp example that was introduced in Chapter 1. Actually, the opposite is true—the `SimpleApp` classes are based on the `HelloWorld` classes.

To create the SimpleApp project, I started with the HelloWorld project. I then followed this chapter's steps for renaming the project files and renaming the application-defined classes. If you think that my following a Chapter 2 process while writing Chapter 1 was a good trick, just wait—there's more! I then wrote SimpleApp such that it became a simplified version of the HelloWorld program. Not many books can lay claim to simplifying what is traditionally the most basic program that can be written for an operating system.

In order to keep this book's first example small and simple, I deleted the *Hello-View.cpp* and *HelloView.h* files and removed any references to the class that was defined and implemented in those two files—the `HelloView` class. I also stripped out the `#ifndef` and `#define` preprocessor directives to further simplify things.

Now that the relationship between the Chapter 1 SimpleApp example project and the Be-supplied HelloWorld project has been established, where does the MyHelloWorld project fit in? As you saw in this chapter, the MyHelloWorld project also came about by duplicating the HelloWorld project. I renamed the files and the application-defined classes, but I left all the other code intact. Building an application from the MyHelloWorld project results in a program indistinguishable from the application that gets built from the HelloWorld project (except for the look of the application's icon).

In this section I describe the code that makes up the HelloWorld project. At the end of this section I make a few minor changes to this code. When I do that I'll use the MyHelloWorld project so that I can leave the original HelloWorld project untouched, and so that I can justify the time I invested in making the MyHelloWorld project!

HelloWorld View Class

A view is a rectangular area in which drawing takes place. Drawing is an important part of almost all Be programs, so views are important too. Views are discussed at length in Chapter 4, *Windows, Views, and Messages*, so I'll spare you the details here. I will, however, provide a summary of views so that you aren't in the dark until you reach the fourth chapter.

A view can encompass the entire content area of a window—but it doesn't have to. That is, a window's content area can consist of a single view or it can be divided into two or more views. Note that a view has no visible frame, so when I say that a window can be divided, I'm referring to a conceptual division—the user won't be aware of the areas occupied by views in a window.

A view serves as an independent graphics environment, or state. A view has its own coordinate grid so that the location at which something is drawn in the view can be kept independent of other views that may be present in a window. A view has a large set of drawing characteristics associated with it, and keeps track of the current state of these characteristics. The font used to draw text in the view and the width of lines that are to be drawn in the view are two examples of these characteristics.

The information about a single view is stored in a view object, which is of the `BView` class (or a class derived from the `BView` class). The `BView` class is a part of the Interface Kit.

When a window is created, it doesn't initially have any views attached to (associated with) it. Because drawing always takes place in a view and never directly in a window, any program that draws must define a class derived from the `BView` class. When the program creates a new window object, it will also create a new BView-derived object and attach this view object to the Window object.

A class derived from the BView class will typically define a minimum of three virtual member functions. That is, such a class will override at least three of the many BView member functions. These functions are the constructor function, the AttachedToWindow() function, and the Draw() function. The HelloView class defined in the HelloWorld project does just that:

```
class HelloView : public BView {

public:
                HelloView(BRect frame, char *name);
virtual  void   AttachedToWindow();
virtual  void   Draw(BRect updateRect);
};
```

These member functions, along with several other BView member functions, are discussed at length in Chapter 4. Here I'll provide only a brief overview of each.

The purpose of the constructor is to establish the size of the view and, optionally, provide a name for the view (NULL can be passed as the second parameter). If a window is to have only a single view, then the first parameter to the constructor is a rectangle of the same size as the content area of the window the view is to be attached to.

The HelloView constructor does nothing more than invoke the BView class constructor. When a HelloView view object is created by the HelloWorld program, the program passes to the HelloView constructor the size and name the view is to have. The HelloView constructor will in turn pass that information on to the BView constructor (as the rect and name parameters). The third BView constructor parameter describes how the view is to be resized as its window is resized. The Be constant B_FOLLOW_ALL sets the view to be resized in tandem with any resizing of the window. The final BView constructor parameter determines the types of notifications the view is to receive from the system. The Be constant B_WILL_DRAW means that the view should be notified when the visible portions of the view change (and an update is thus necessary).

```
HelloView::HelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

You might not be able to fully initialize a view object when creating it—some characteristics of the view may be dependent on the window the view becomes attached to. Thus the existence of the BView member function AttachedToView(). This function is automatically called by the operating system when the view is attached to a window (covered ahead in the discussion of the

HelloWorld's application class). The characteristics of the view should be included in the implementation of `AttachedToWindow()`.

The HelloView version of `AttachedToWindow()` sets the font and font size that are to be used for any text drawing that takes place within the view:

```
void HelloView::AttachedToWindow()
{
    SetFont(be_bold_font);
    SetFontSize(24);
}
```

The `SetFont()` and `SetFontSize()` functions are member functions of the `BView` class. The function names make the purpose of each obvious, so I won't offer any more information here. You will, however, find plenty of information on drawing strings in Chapter 5, *Drawing*.

In Chapter 1 you saw member functions invoked via an object—as in the `SimpleWindow` object invoking the `BWindow` member function `Show()`:

```
aWindow->Show();
```

In the `AttachedToWindow()` member function you see that the `BView` member functions `SetFont()` and `SetFontSize()` are invoked without the use of an object.

As a C++ programmer, it should be obvious how a member function can be invoked in this way. If it isn't, read on. The implementation of a member function results in a routine that can be invoked by any object of the type of class to which the function belongs. Therefore, the code that makes up a member function can operate on any number of objects. When a member function includes code that invokes a different member function (as the `HelloView` member function `AttachedToWindow()` invokes the `BView` member function `SetFont()`), it is implicit that the invocation is acting on the current object. So no object prefaces the invocation.

The `BView` class includes a `Draw()` member function that automatically gets called when the window a view is attached to needs updating. The system keeps track of the views that are attached to a window, and if more than one view object is attached to a window, the `Draw()` function for each view object is invoked. The `Draw()` function should implement the code that does the actual drawing in the view.

The HelloView version of `Draw()` simply establishes the starting position for drawing a string of text and then draws that string.

```
void HelloView::Draw(BRect updateRect)
{
    MovePenTo(BPoint(10, 30));
    DrawString("Hello, World!");
}
```

HelloWorld Window Class

In Chapter 1, you received an overview of the `BWindow` class and classes derived from it, so I won't go on at length about those topics here. In fact, I'll barely discuss the `HelloWindow` class at all. If you need a refresher on defining a class derived from `BWindow`, refer back to the SimpleApp example in Chapter 1—there you'll find that the `SimpleWindow` class consists of the same two member functions (a constructor and the `QuitRequested()` function) as the `HelloWindow` class.

```
class HelloWindow : public BWindow {

public:
               HelloWindow(BRect frame);
virtual  bool  QuitRequested();
};
```

In the SimpleApp project, the only thing the `SimpleWindow` class constructor does is invoke the `BWindow` class constructor. That's true for the `HelloWindow` class constructor as well. The first parameter to the `BWindow` constructor, which comes from the `HelloWindow` constructor, sets the size of the window. The second parameter is a string that is used as the window's title. The `SimpleWindow` class passes the string "A Simple Window," while the `HelloWindow` class passes the more appropriate string "Hello." The third parameter is a Be-defined constant that specifies the type of window to be displayed. The final parameter defines the behavior of the window. The `SimpleWindow` class passes the constant `B_NOT_RESIZABLE`, while the `HelloWindow` class passes the result of combining two constants, `B_NOT_RESIZABLE` and `B_NOT_ZOOMABLE`.

```
HelloWindow::HelloWindow(BRect frame)
    : BWindow(frame, "Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE |
                              B_NOT_ZOOMABLE)
{
}
```

You've already seen the code that makes up the `QuitRequested()` function. In the Chapter 1 SimpleApp project, I left this routine from the HelloWorld project unchanged. Recall from that chapter that the purpose of overriding this `BWindow` member function is to cause a mouse click on a window's close button to not only close the window, but to quit the application as well:

```
bool HelloWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}
```

HelloWorld Application Class

All programs must define a class derived from the BApplication class. Like the SimpleApp's SimpleApplication class, the HelloWorld's HelloApplication class consists of just a constructor:

```
class HelloApplication : public BApplication {

public:
           HelloApplication();
};
```

For the implementation of the SimpleApplication constructor, I started with the HelloApplication constructor, then stripped out the view-related code. As you look at the HelloApplication constructor you should recognize much of the code:

```
HelloWindow  *aWindow;
...
BRect         aRect;

aRect.Set(20, 30, 180, 70);
aWindow = new HelloWindow(aRect);
...
...
aWindow->Show();
```

This snippet begins by declaring a window variable and a rectangle variable. Next, the boundaries of the rectangle that sets the window's size are established. Then a new window object (an object of the HelloWindow class) is created. Finally, the BWindow member function Show() is invoked to display the new window. The code I omitted from the above snippet is the HelloApplication constructor's view-related code:

```
HelloView    *aView;

aRect.OffsetTo(B_ORIGIN);
aView = new HelloView(aRect, "HelloView");

aWindow->AddChild(aView);
```

In the above snippet, the size of the rectangle object isn't changed, but its existing boundaries (as set by the prior call to the BRect member function Set()) are offset. The one parameter in the BRect member function OffsetTo() is the location to move the upper-left corner of the rectangle to. The constant B_ORIGIN tells OffsetTo() to shift the rectangle aRect from its present location such that its upper-left corner aligns with the origin—the point (0, 0). This has the effect of changing the aRect coordinates from (20, 30, 180, 70) to (0, 0, 160, 40). The details of the coordinate system, by the way, are presented in Chapter 4. Next, a new HelloView object is created. The HelloView constructor sets the size of the view to aRect and the name of the view to “HelloView.” Finally, the window object aWindow invokes the BWindow member function AddChild() to attach the view to itself. Recall that in order for a view to be of use, it must be attached to a window; in order for drawing to take place in a window, it must have a view attached to it.

You’ve seen the implementation of the HelloApplication constructor in bits and pieces. Here it is in its entirety:

```
HelloApplication::HelloApplication()
    : BApplication("application/x-vnd.Be-HLWD")
{
    HelloWindow  *aWindow;
    HelloView    *aView;
    BRect         aRect;

    aRect.Set(20, 30, 180, 70);
    aWindow = new HelloWindow(aRect);

    aRect.OffsetTo(B_ORIGIN);
    aView = new HelloView(aRect, "HelloView");

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

You may have noticed that after the window object is created, no drawing appears to take place. Yet you know that the string “Hello, World!” gets drawn in the window. Recall that the updating of a window has the effect of calling the Draw() member function of any views attached to that window. When aWindow invokes the BWindow Show() member function, the window is displayed. That forces an update to occur. That update, in turn, causes the aView object’s Draw() method to be invoked. Look back at the implementation of the HelloView class Draw() member function to see that it is indeed this routine that draws the string.

HelloWorld main() Function

When I wrote the main() routine for SimpleApp, the only changes I made to the HelloWorld version of main() involved changing HelloApplication references to SimpleApplication references. For the details of what main() does, refer back to Chapter 1. I’ll summarize by saying that main() performs the mandatory creation of a new application object, then starts the program running by invoking the `BApplication` member function `Run()`:

```
int  main()
{
    HelloApplication  myApplication;

    myApplication.Run();

    return(0);
}
```

Altering the Source Code

To implement the functionality required of your own program, you'll no doubt make some changes and substantial additions to the code of whatever project you start with. After reading a few more chapters, you'll know the details of how to draw just about anything to a window and how to add controls (such as buttons) to a window. At that point you'll be ready to make large-scale revisions of existing code. While at this early point in the book you may not feel ready to make sweeping changes to Be code, you should be ready to make at least minor revisions. In this section, I'll do that to the HelloWorld project.

I'll leave the HelloWorld project intact and instead make the changes to a duplicate project—a project I've name MyHelloWorld. If you haven't followed the steps in this chapter's "Setting Up a New BeIDE Project" section, do so now. Besides getting experience at starting a new project, you'll bring yourself to the point where you can follow along with the code changes I'm about to make. Here they are:

- Change the text that is drawn in the window
- Change the window's title
- Add a zoom button to the window's tab
- Change the size of the window

At the top of Figure 2-14, you see the original window from the HelloWorld program, while at the bottom of the figure you see the new window from the MyHelloWorld program.

The drawing that takes place in the MyHelloWorld window is done by the `MyHelloView Draw()` member function. A call to the `BView` member function `DrawString()` writes the string "Hello, World!" to the window. Changing this one `DrawString()` parameter to "Hello, My World!" will cause this new string to be drawn to the window instead. The affected line is in the *MyHelloView.cpp* file, and is shown here in bold type:

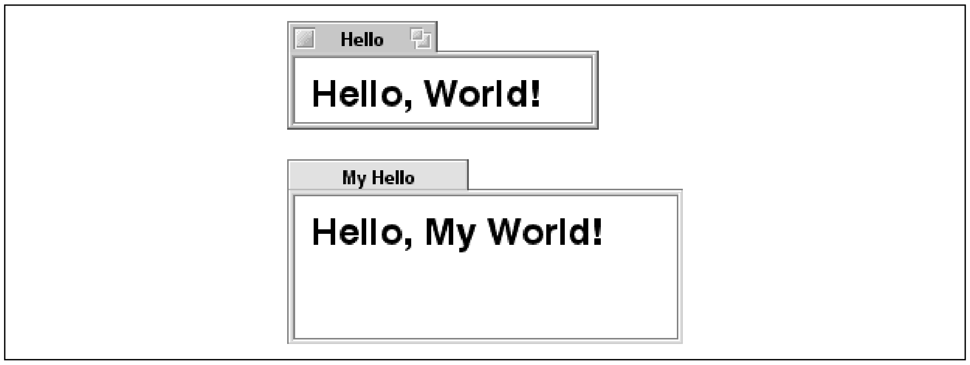


Figure 2-14. The window from HelloWindow (top) and MyHelloWindow (bottom)

```
void MyHelloView::Draw(BRect updateRect)
{
    MovePenTo(BPoint(10, 30));
    DrawString("Hello, My World!");
}
```

The change to the window's title and the addition of a zoom button to the window are both taken care of in the `MyHelloWindow` class constructor, so I'll open the *MyHelloWindow.cpp* file. `MyHelloWindow()` invokes the `BWindow` constructor. The second parameter in the `BWindow` constructor defines the window's title, so changing that parameter from "Hello" to "My Hello" handles the window title change. The last parameter in `BWindow()` is a Be-defined constant, or combination of Be-defined constants, that defines the behavior of the window. In the original `HelloWindow` class, this parameter was a combination of the constants that specified that the window be drawn without a resize knob and without a zoom button (`B_NOT_RESIZABLE | B_NOT_ZOOMABLE`). I'll change this parameter to the one constant `B_NOT_RESIZABLE` so that a window created from the `MyHelloWindow` class will be without a resize knob, but will now have a zoom button:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

The final change I'll make to the code that came from the HelloWorld project will change the size of the program's window. The `MyHelloApplication` constructor uses a `BRect` object to define the size of the window, so I'll go to this function in the *MyHelloWorld.cpp* file to make the change. In the following version of `MyHelloApplication()`, I've changed the coordinates of the rectangle object `aRect` from (20, 30, 180, 70) to (20, 30, 230, 100) in order to set up a window that's a little larger than the one used in the original HelloWorld program.

```
MyHelloApplication::MyHelloApplication()
    : BApplication("application/x-vnd.dps-myworld")
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;

    aRect.Set(20, 20, 240, 90);
    aWindow = new MyHelloWindow(aRect);

    aRect.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(aRect, "MyHelloView");

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

I can compile all the MyHelloWorld project files, build a new version of the MyHelloWorld program, and run that program by choosing Run from the Project menu in the menubar of the MyHelloWorld project window. If I've done everything correctly, the program will display a window that looks like the one at the bottom of Figure 2-14.

In this chapter:
- *Overview of the BeOS Software Kits*
- *Software Kit Class Descriptions*
- *Chapter Example: Adding an Alert to MyHelloWorld*

3
BeOS API Overview

Writing a Be application generally involves starting with an existing base of code and then using several of the Be software kit classes to add new functionality to the base code. In Chapter 2, *BeIDE Projects*, you saw how to ready an existing project to serve as the base for your new project. In this chapter, you'll see how to select and use a software kit class to modify your new project.

This chapter begins with an overview of the Be software kits. Knowing the purpose of each kit will help you quickly hone in on which kits will be of the most use in your project. After finding a kit of interest, you need to locate a useful class within that kit. To do that, you'll use the Be Book—the electronic document by Be, Inc. that serves as the BeOS class reference. Once you've found a class of possible interest, you'll read through the Be Book's class description to find out all about the class: an overview of how objects are created, what they're useful for, and so forth. In this chapter, you'll see how to get the most out of the class descriptions in the Be Book.

The Be Book is essential documentation for any Be programmer—but it isn't a tutorial. In this chapter, I close by looking at how the Be Book describes one class (the `BAlert` class), and then go on to integrate an object of this class type in a simple program. The remaining chapters in this book provide example snippets and programs that "bring to life" the class descriptions found in the Be Book.

Overview of the BeOS Software Kits

Chapter 1, *BeOS Programming Overview*, provided a very brief description of each kit—only a sentence or two. Because you hadn't been exposed to any of the details of BeOS programming at that point, out of necessity those descriptions didn't give examples of kit classes and member functions. Now that you've studied the fundamentals of Be programming and have looked at some example source code, it's time to rewrite the kit summaries, with an emphasis on the key classes and a few important member functions.

The focus of this book is on the first three kits described below: the Application Kit, the Interface Kit, and the Storage Kit. Don't feel as if you're being short-changed, though—these kits provide dozens of classes that allow you to create full-featured applications complete with windows, graphics, editable text, and all manner of controls.

While each of the software kits isn't represented by its own chapter in this book, all are at least briefly described below for the sake of completeness. A couple of the these kits can't be covered, as they aren't complete as of this writing. Be provides information on kit updates at the developer web page at *http://www.be.com/developers*, so you'll want to check that site occasionally. Other kits are complete, but their specialized functionality makes detailed descriptions out of scope for this book. Note that while some kits don't have a chapter devoted to them, some of their classes appear throughout the book. See the description of the Support Kit below for a specific example concerning the `BLocker` class.

Application Kit

The classes of the Application Kit communicate with the Application Server and directly with the kernel. Every program must create a single instance of a class derived from the Application Kit class `BApplication`—the HelloWorld program provides an example of how this is typically done. This `BApplication` object is necessary for a couple of reasons. The application object:

- Makes a connection to the Application Server. This connection is vital if the program is to display and maintain windows, which of course most Be programs do.
- Runs the program's main message loop. This loop provides a messaging system that keeps the program aware of events (such as a press of a keyboard key by the user).

An important member function of the `BApplication` class is `Run()`. The `main()` function of every Be program must create an instance of the `BApplication` class and then invoke `Run()` to start the program.

The `BApplication` class is derived from two other Application Kit classes—`BLooper` and `BHandler`. A `BLooper` object creates and then controls a message loop, a thread that exits to transfer messages to objects. A `BHandler` object is one that is capable of receiving a message from a `BLooper` object—it handles a message received from a message loop. Because a `BApplication` object is also a

BLooper and BHandler object, it acts as both a message loop and a message handler. Refer to Figure 1-2 in Chapter 1 for a look at the Application Kit class hierarchy that illustrates the relationship between the BApplication and BLooper and BHandler classes.

Interface Kit

With over two dozen classes, the Interface Kit is the largest of the Be software kits. It's also the one you'll make the most use of—as will this book. The chapters from Chapter 4, *Windows, Views, and Messages*, through Chapter 8, *Text*, deal almost exclusively with this kit. In Chapter 1 you saw that a window is an object derived from an Interface Kit class—the BWindow class. In Chapter 2 you were introduced to the concept that all drawing in a window is done via an object derived from another Interface Kit class—the BView class (much more on this important topic appears in Chapter 4 and Chapter 5, *Drawing*). In subsequent chapters you'll learn that controls (such as buttons and checkboxes), strings, and menus are also types of views (objects of classes that are derived from the BView class). Because all drawing takes place in a view, and because all of the aforementioned items are drawn, this should seem reasonable. It should also shed more light on the class hierarchy of the Interface Kit, as shown in Figure 1-4 back in Chapter 1.

Like a BApplication object (see the Application Kit above), a BWindow object is derived from both the BLooper and BHandler classes, so it is both an organizer of messages in a message loop and a handler of messages. When an event is directed at a window (such as a mouse button click while the cursor is over a window's close button), the system transfers a message to the window object's thread. Because the window is a message handler as well as a message loop, it may also be able to handle the message.

A window contains one or more views—objects of the BView class or one of its many derived classes. Often a window has one view that is the same size as the content area of the window (or larger than the content area of the window if it includes scrollbars). This view then serves as a holder of other views. These smaller, *nested*, views can consist of areas of the window that are to act independently of one another. Any one of these smaller views may also be used to display a single interface item, such as a button or a scrollbar. Because the contents of a view are automatically redrawn when a window is updated, it makes sense that each interface item exists in its own view. Some of the Interface Kit control classes that are derived from the BView class (and which you'll work with in Chapter 6, *Controls and Messages*) include BCheckBox, BRadioButton, and BPictureButton.

Storage Kit

All operating systems provide file system capabilities—without them, data couldn't be saved to disk. The Storage Kit defines classes that allow your program to store data to files, search through stored data, or both.

The `BNode` class is used to create an object that represents data on a disk. The `BFile` class is a subclass of `BNode`. A `BFile` object represents a file on disk. Creating a `BFile` object opens a file, while deleting the same object closes the file. A `BFile` object is the mechanism for reading and writing a file. The `BDirectory` class is another subclass of `BNode`. A `BDirectory` object represents a folder, and allows a program to walk through the folder's contents and create new files in the folder.

The concept of file attributes, associating extra information with a given file, allows for powerful file indexing and searching. The `BQuery` class is used to perform searches.

Support Kit

The Support Kit, as its name suggests, supports the other kits. This kit defines some datatypes, constants, and a few classes. While the nature of the classes of the Support Kit makes a chapter devoted to it impractical, you will nonetheless encounter a couple of this kit's classes throughout this book.

The `BArchivable` class defines a basic interface for storing an object in a message and instantiating a copy of that object from the message.

The `BLocker` class is used to limit program access to certain sections of code. Because the BeOS is multithreaded, there is the possibility that a program will attempt to access data from two different threads simultaneously. If both threads attempt to write to the same location, results will be unpredictable. To avoid this, programs use the `Lock()` and `Unlock()` member functions to protect code. Calls to these functions are necessary only under certain circumstances. Throughout this book mention of the use of `Lock()` and `Unlock()` will appear where required.

Media Kit

The Media Kit is designed to enable programs to work with audio and video data in real time—the kit classes provide a means for processing audio and video data. The Media Kit relies on nodes—specialized objects that perform media-related tasks. A node is always indirectly derived from the `BMediaNode` class, and there are several basic node types. Examples are producer and consumer nodes. A producer node sends output to media buffers, which are then received by consumer nodes.

Midi Kit

MIDI (Musical Instrument Digital Interface) is a communication standard for representing musical data that is generated by digital musical devices. MIDI was created to define a way for computer software and electronic music equipment to exchange information. The Midi Kit is a set of classes (such as `BMidiPort`) used to assemble and disassemble MIDI messages. A MIDI message describes a musical event, such as the playing of a note. To make use of the Midi Kit classes, you'll need to have prior knowledge of the MIDI software format.

Device Kit

The Device Kit classes (such as `BJoystick` and `BSerialPort`) are used for the control of input and output devices and for the development of device drivers. These classes serve as interfaces to the ports on the back of a computer running the BeOS.

Network Kit

The Network Kit consists of a number of C functions. The C functions are global (they can be used throughout your program), and exist to allow your program to communicate with other computers using either the TCP or UDP protocols. One such function is `gethostbyname()`, which is used to retrieve information about computers attached to the user's network.

OpenGL Kit

OpenGL is a cross-platform application programming interface developed to facilitate the inclusion of interactive 2D and 3D graphics in computer programs. Introduced in 1992, OpenGL has become the industry standard for high-performance graphics. The OpenGL Kit contains classes that simplify the implementation of animation and three-dimensional modeling in your programs. The OpenGL Kit is one of the newer BeOS kits, and is incomplete as of this writing. Working with the OpenGL classes requires some previous experience with OpenGL.

Game Kit

Like the OpenGL Kit, the Game Kit is incomplete. While it will eventually contain a number of classes that will aid in the development of games, at this time it includes just two classes. The `BWindowScreen` class is used by an application to gain direct access to the screen in order to speed up the display of graphics. The `BDirectWindow` class is an advanced class commonly used by game and media developers.

Kernel Kit

The primary purpose of the C functions that make up the Kernel Kit is to support the use of threads. While the BeOS automatically spawns and controls many threads (such as the one resulting from the creation of a new window), your program can manually spawn and control its own threads. This kit includes classes that support semaphores for protecting information in the BeOS multithreaded environment and shared memory areas for communicating between multiple threads and multiple applications.

Translation Kit

The Translation Kit provides services that ease the work in translating data from one format to another. For instance, this kit could be used to translate the data in an imported JPEG file into a `BBitmap` object (the `BBitmap` being a class defined in the Interface Kit) that your program could then manipulate.

Software Kit Class Descriptions

The classes (and in a few cases, the C functions and structures) that make up the BeOS software kits serve any imaginable programming need, yet they share many similarities. Becoming familiar with what makes up a software kit class definition and how Be documents such a class will help you make use of all of the software kits.

Contents of a Class

A Be software kit consists of classes. Each class can consist of member functions, data members, and overloaded operators. While a kit class will always have member functions, it isn't required to (and very often doesn't) have any data members or operators.

Data members

C++ programmers are used to creating classes that define a number of data members and a number of member functions. In the first few chapters of this book, though, you've read little about data members in Be classes. If a Be class does define data members, they are usually defined to be `private` rather than `public`. These `private` data members will be used within class member functions, but won't be used directly by your program's objects. That is, a data member generally exists for use in the implementation of the class rather than for direct use by your program—data members are thus of importance to a class, but they're almost never of importance to you.

You can see an example of the data members of a Be class by perusing the Be header files. In Chapter 1 you saw a snippet that consisted of a part of the `BWindow` class. In the following snippet I've again shown part of this class. Here, however, I've included the `private` keyword and some of the approximately three dozen data members that are a part of this class.

```
class BWindow : public BLooper {

public:
                    BWindow(BRect frame,
                            const char *title,
                            window_type type,
                            uint32 flags,
                            uint32 workspace = B_CURRENT_WORKSPACE);
virtual            ~BWindow();


...
         void  ResizeBy(float dx, float dy);
         void  ResizeTo(float width, float height);
virtual  void  Show();
virtual  void  Hide();
         bool  IsHidden() const;
...
private:
...
char      *fTitle;
uint32    server_token;
char      fInUpdate;
char      f_active;
short     fShowLevel;
uint32    fFlags;
port_id   send_port;
port_id   receive_port;
BView     *top_view;
BView     *fFocus;
...
}
```

Member functions

A class constructor and destructor are member functions, as are any class hook functions. While the constructor, destructor, and hook functions are often described and discussed separately from other member functions, all fall into the general category of member functions, as shown in Figure 3-1.

From programming in C++ on other platforms, you're familiar with constructors and destructors. But you may not know about hook functions. A *hook function* is a member function that can be called directly by a program, but can also be (and very often is) invoked automatically by the system.

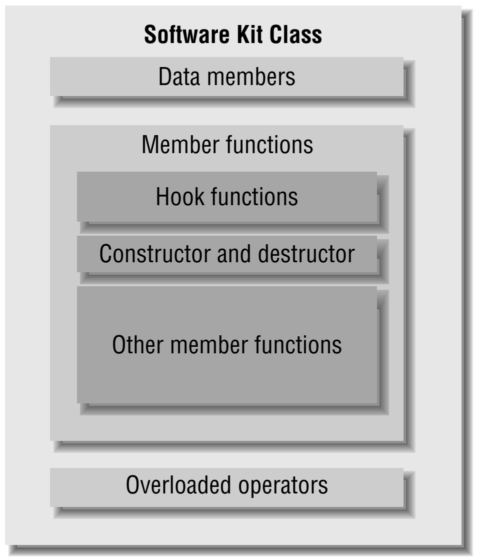


Figure 3-1. A kit class may consist of data members, member functions, and operators

Many software kit class member functions are declared using the C++ keyword `virtual`. The most common reason for declaring a member function virtual is so that a derived class can override the function. Additionally, hook functions are declared to be virtual for a second reason as well: your program may want to add functionality to that which is already provided by the hook function.

When an application-defined class defines a member function, that function is typically invoked by an object created by the application. A hook function is also a routine defined by an application-defined class, but it is one that is invoked automatically by the software kit, not by an object. In order to be called by the system, a hook function must have a specific name that the system is aware of.

You saw an example of a hook function in the SimpleApp example back in Chapter 1—the `QuitRequested()` function. When a window's close button is clicked on, the Be system automatically invokes a routine named `QuitRequested()`. If the application has defined such a function in the `BWindow`-derived class that the window object belongs to, it will be that member function that gets invoked. As a reminder, here's the `QuitRequested()` function as defined in the `SimpleWindow` class of the SimpleApp example:

```
bool SimpleWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}
```

A hook function is so named because the function serves as a place where you can "hook" your own code onto that of the Be-written software kit code. By implementing a hook function, your application in essence extends the functionality of the Be operating system. The system is responsible for calling a hook function, while your application is responsible for defining the functionality of that function.

Overloaded operators

Along with member functions and data members, you may find overloaded operators in a Be class. A few classes overload some of the C++ operators, but most classes don't overload any. You'll find that the need for a class to overload operators is usually intuitive. For instance, the `BRect` class overloads the comparison operator (==) so that it can be used to test for the equality of two rectangle objects. Because the comparison operator is defined in C++ such that it can be used to compare one number to another, the `BRect` class needs to rewrite its definition so that it can be used to test all four coordinates of one rectangle to the four coordinates of another rectangle.

As you just saw for the `BRect` class, if it makes sense for a class to redefine a C++ operator, it will. For most other classes, the use of operators with objects doesn't make sense, so there's no need to overload any. For instance, the `BWindow` and `BView` classes with which you're becoming familiar don't included any overloaded operators. After all, it wouldn't be easy to test if one window is "equal" to another window.

Class Descriptions and the Be Book

The definitive source of information for the many classes that make up the BeOS software kits is the Be class reference by the programmers of the BeOS. The electronic versions of this document (you'll find it in both HTML and Acrobat formats) go by the name of the Be Book, while the printed version is titled *The Be Developer's Guide* (available from O'Reilly). After programming the BeOS for awhile, you'll find the Be Book or its printed version indispensable. But now, as you take your first steps in programming the BeOS, you may find the voluminous size and the reference style of this book intimidating. While this one thousand or so page document is comprehensive and well-written, it is a class reference, not a BeOS programming tutorial. When you have a solid understanding of how classes are described in the Be Book you'll be able to use the Be Book in conjunction with this text if you wish.

The Be Book is organized into chapters. With the exception of the first chapter, which is a short introduction to the BeOS, each chapter describes the classes of one kit. Chapter 2 covers the classes of the Application Kit, Chapter 3 describes the classes that make up the Storage Kit, and so forth. Each class description in a chapter is itself divided into up to six sections: *Overview*, *Data Members*, *Hook Functions*, *Constructor and Destructor*, *Member Functions*, and *Operators*. If any one of these six sections doesn't apply to the class being described, it is omitted from the class description. For instance, the `BWindow` class doesn't overload any operators, so its class description doesn't include an *Operators* section.

The following list provides explanations of what appears in each of the six sections that may be present in a class description in the Be Book. For each software kit class, the sections will appear in the order listed below, though some of the sections may be omitted:

Overview
: A class description begins with an overview of the class. Such information as the purpose of the class, how objects of the class type are used, and related classes may be present in this section. The overview will generally be short, but for significant classes (such as `BWindow` and `BView`), it may be several pages in length.

Data Members
: This section lists and describes any public and protected data members declared by the class. If a class declares only private data members (which is usually the case), this section is omitted.

Hook Functions
: If any of the member functions of a class serve as hook functions, they will be listed and briefly described in this section. This section serves to summarize the purpose of the class hook functions—a more thorough description of each hook function appears in the *Member Functions* section of the class description. Many classes don't define any hook functions, so this section will be omitted from a number of class descriptions.

Constructor and Destructor
: A class constructor and destructor are described in this section. A few classes don't define a destructor (objects of such class types know how to clean up and delete themselves). In such cases, this section will be named *Constructor* rather than *Constructor and Destructor*.

Member Functions
: This section provides a detailed description of each of the member functions of a class, except the class constructor and destructor (which have their own section). While class hook functions have their own section, that section serves mostly as a list of hook functions—the full descriptions of such functions appear here in *Member Functions*. Every class consists of at least one member function, so this section is always present in a class description.

Operators

Here you'll find a description of any C++ operators a class overloads. Most classed don't overload any operators, so this section is frequently absent from a class description.

A BeOS Class Description: The BRect Class

Now that you've had a general look at how a class description appears in the Be Book, you'll want to see a specific example. Here I'll look at the Be Book description of the `BRect` class. Because this class doesn't have any hook functions, the *Hook Functions* section is omitted from the Be Book's class description. If you'd like to see a specific example of how a class implements hook functions, refer to the "A BeOS Class Description: The BWindow Class" section in this chapter.

As you read these pages, you may want to follow along in the electronic version of the Be Book. If you do, double-click on the Chapter 4 document and scroll to the start of the `BRect` class description.

Overview

The *Overview* section of the `BRect` class description informs you what a `BRect` object is (a rectangle) and how a `BRect` object is represented (by defining four coordinates that specify where the corners of the rectangle are located).

Next in this section is the object's general purpose (to serve as the simplest specification of a two-dimensional area) and a few specific examples of what such an object is used for (to specify the boundaries of windows, scrollbars, buttons, and so on). The `BRect` overview then provides the details of how your specification of a rectangle object's boundaries affects the rectangle's placement in a view.

As you read the overview, notice that no `BRect` data members or `BRect` member functions are mentioned by name. This is typical of a class *Overview* section; what a class object is used for is covered, but details of how to implement this usage aren't. Such details are found in the descriptions of the appropriate functions in the *Member Functions* section.

Data Members

The `BRect` class is one of the few software kit classes that declares public data members. So it is one of the few classes that includes a *Data Members* section. Here you'll find the names and datatypes of the four public data members (they're named `left`, `top`, `right`, and `bottom`, and each is of type `float`). A single-sentence description accompanies the listing of each data member. The specifics of how these data members are used by a `BRect` object appear in discussions of the `BRect` constructor and member functions in the *Constructor* and *Member Functions* sections.

Hook Functions

The `BRect` class defines several member functions, but none of them serves as a hook function, so no *Hook Functions* section appears in the `BRect` class description.

Constructor and Destructor

For the `BRect` class, this section is named *Constructor* rather than *Constructor and Destructor*. There is no `BRect` destructor, which implies that a `BRect` object knows how to delete itself. The `BRect` class is somewhat atypical of kit classes in that it is more like a primitive datatype (such as an `int` or `float`) than a class. The primitive datatypes serve as the foundation of C++, and in the BeOS the `BRect` class serves a somewhat similar purpose; is the basic datatype that serves as the foundation of Be graphics. `BRect` objects are declared in the same way primitive datatype variables are declared—there's no need to use the `new` operator. There's also no need to use the `delete` operator to destroy such objects.

The *Constructor* section reveals that there are four `BRect` constructors. One common method of creating a rectangle is to use the constructor that accepts the four rectangle coordinates as its parameters. The remainder of the *Constructor* section of the `BRect` class description provides example code that demonstrates how each of the four `BRect` constructors can be used.

Each of the four `BRect` constructor functions is declared using the `inline` keyword. As a C++ programmer, you may have encountered inline functions. If you haven't, here's a brief summary of this type of routine. Normally, when a function is invoked, a number of non-routine instructions are executed. These instructions ensure that control is properly moved from the calling function to the called function and then back to the calling function. The execution time for these extra instructions is slight, so their inclusion is seldom a concern. If it does become an issue, though, C++ provides a mechanism for eliminating them—the `inline` keyword. The upside to the use of the `inline` keyword is that execution time decreases slightly. The downside is that this reduced time is achieved by adding more code to the executable. Declaring a function `inline` tells the compiler to copy the body of the function to the location at which the function is called. If the function is called several times in a program, then its code appears several times in the executable.

Member Functions

This section lists each of the `BRect` member functions in alphabetical order. Along with the name and parameter list of any given function is a detailed description of the routine's purpose and use. You've already seen one `BRect` member function in action—the `Set()` function. In the Chapter 1 example, SimpleApp, a `BRect` object is created and assigned the values that define the rectangle used for the boundaries of the SimpleApp program's one window. The *Member Functions* section's description of `Set()` states that the four parameters are used to assign values to the four `BRect` data members: `left`, `top`, `right`, and `bottom`.

Operators

The `BRect` class overrides several C++ operators in order to redefine them so that they work with operations involving rectangles. In the *Operators* section, you see each such operator listed, along with a description of how the overloaded operator is used. The operator you'll use most often is probably the assignment operator (=). By C++ definition, this operator assigns a single number to a variable. Here the `BRect` class redefines the operator such that it assigns all four coordinates of one rectangle to the four coordinates of another rectangle. The *Rect.h* header file provides the implementation of the routine that redefines the assignment operator:

```
inline BRect &BRect::operator=(const BRect& from)
{
    left = from.left;
    top = from.top;
    right = from.right;
    bottom = from.bottom;
    return *this;
}
```

A BeOS Class Description: The BWindow Class

After reading the "A BeOS Class Description: The BRect Class" section of this chapter, you should have a good feel for how classes are described in the Be Book. The `BRect` class doesn't include hook functions, though, so if you want to get a little more information on this type of member function, read this section. The `BWindow` class doesn't define any public data members or overload any operators, so you won't find talk of a *Data Members* section or an *Operators* section here. The `BRect` class, however, does define public member functions and overload operators. If you skipped the previous part, you may want to read it as well.

Following along in the Be Book isn't a requirement, but it will be helpful. If you have access to the electronic version of the Be Book, double-click on the Chapter 4 document and scroll to the start of the `BWindow` class description.

Overview

The *Overview* section of the `BWindow` class description starts by telling you the overall purpose of the `BWindow` class (to provide an application interface to windows) and the more specific purpose of this class (to enable one object to correspond to one window).

A class overview may mention other classes, kits, or servers that play a role in the workings of the described class. For instance, the `BWindow` overview mentions that the Application Server is responsible for allocating window memory and the `BView` class makes drawing in a window possible.

Before working with a class for the first time, you'll want to read the *Overview* section to get, of course, an overview of what the class is all about. But you'll also want to read the *Overview* section to pick up bits of information that may be of vital interest to your use of the class. The `BWindow` overview, for instance, notes that there is a strong relationship between the `BApplication` class and the `BWindow` class, and that a `BApplication` object must be created before the first `BWindow` object is created. The overview also mentions that a newly created `BWindow` object is hidden and must be shown using the `Show()` member function. You already knew these two facts from Chapter 1 of this book, but that chapter didn't supply you with such useful information about each of the dozens of Be software kit classes!

Data Members

Like most classes, the `BWindow` class doesn't define any public data members, so no *Data Members* section appears in the `BWindow` class description.

Hook Functions

The `BWindow` class has close to a dozen member functions that serve as hook functions—routines that are invoked by the system rather than invoked by window objects. Here you find the names of the `BWindow` hook functions, along with a single-sentence description of each. A detailed description of each function, along with parameter types, appears in the *Member Functions* section.

Many of the short descriptions of the hook functions tell why the function can be implemented. For instance, the `FrameMoved()` hook function can be implemented to take note of the fact that a window has moved. Most programs won't have to perform any special action if one of the windows is moved. If, however, your application does need to respond to a moved window, it can—thanks to the `FrameMoved()` hook function. If your application implements a version of this function, the movement of a window causes the system to automatically execute your program's version of this routine.

Constructor and Destructor

Some classes have a constructor that includes a parameter whose value comes from a Be-defined constant. In such cases the Be-defined constants will be listed and described in the *Constructor and Destructor* section. For example, the third of the `BWindow` constructor's several parameters has a value that comes from one of several Be-defined constants. This `window_type` parameter specifies the type of window to be created. Your application can use either the `B_MODAL_WINDOW`, `B_BORDERED_WINDOW`, `B_TITLED_WINDOW`, or `B_DOCUMENT_WINDOW` constant in the creation of a `BWindow` object. In this *Constructor and Destructor* section you'll find a description of each of these constants.

While a class destructor may be defined, objects of a kit class type often don't need to explicitly call the object's destructor when being destroyed. That's because the BeOS does the work. The *Constructor and Destructor* section lets you know when this is the case for a class. The `BWindow` class provides an example. As stated in the *Constructor and Destructor* section of the `BWindow` class description, a `BWindow` object is destroyed by calling the `BWindow` member function `Quit()`. This routine is responsible for using the delete operator on the window object and then invoking the `BWindow` destructor.

Member Functions

This section describes each of the more than seventy member functions of the `BWindow` class. You've worked with a couple of these routines, including the `Show()` function, in Chapter 1 and Chapter 2. If you look up `Show()` in the *Member Functions*, you'll see that `Show()` is used to display a window—as you already know. Here you'll also learn that this routine places the window in front of any other windows and makes it the active window.

Operators

The `BWindow` class doesn't overload any C++ operators, so the *Operators* section is omitted from the `BWindow` class description.

Chapter Example: Adding an Alert to MyHelloWorld

Each remaining chapter in this book will include numerous code snippets that demonstrate the several topics presented in the chapter. The chapter will then close with the source code and a walk-through of a short but comprehensive example that exhibits many or all of the chapter topics. To keep new code to a minimum and focus on only the new material presented in the chapter, each example program will be a modification of the MyHelloWorld program introduced in Chapter 2.

This chapter provided an overview of the entire BeOS software kit layer, which makes including an example program relevant to the chapter a little difficult. Still, I'd feel uncomfortable ending a chapter without at least a short exercise in adding code to a Be application! In this section, I first rearrange some of the code in the MyHelloWorld listing to demonstrate that in Be programming—as in programming for any platform—there's more than one means to accomplish the same goal. After that, I have the new version of MyHelloWorld open both its original window and a new alert by adding just a few lines of code to the *MyHelloWorld.cpp* listing.

Revising MyHelloWorld

By this point in your studies of Be programming, you should have enough of an understanding of Be software kits, classes, and member functions that you feel comfortable making at least minimal changes to existing Be source code. While you'll often start a new programming endeavor from an existing base of code, you'll always need to adapt that code to make it fit your program's needs. You should also be gaining enough of an understanding of Be code that you feel comfortable with the idea that there is more than one way to solve any programming task. As you look at the source code of existing Be applications, be aware that different programmers will write different code to achieve the same results.

In this section I'll rearrange some of the code in the *MyHelloWindow.cpp* and *MyHelloWorld.cpp* source code files from the MyHelloWorld program introduced in Chapter 2. Building a MyHelloWorld application from the modified listings will result in a program that behaves identically to the version built in Chapter 2.

The version of MyHelloWorld presented here will be used as the basis for each of the example programs in the remainder of this book. While there's nothing tricky in the code presented here, you'll still want to take a close look at it so that you can focus on only the new code that gets added in subsequent example programs.

Two approaches to achieving the same task

While I could just go ahead and make a few alterations for the sake of change, I'll instead assume I have a valid reason for doing so! First, consider the Chapter 2 version of MyHelloWorld. The `MyHelloWindow` class has a constructor that defines an empty window. The `MyHelloView` class has a constructor that defines an empty view and a `Draw()` member function that draws the string "Hello, My

World!" in that view. Recall that there is no connection between a window object created from the MyHelloWindow class and a view object created from the MyHelloView class until after the window object is created and the view object is attached to it in the HelloApplication class constructor:

```
MyHelloApplication::MyHelloApplication()
    : BApplication("application/x-vnd.dps-myWorld")
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;

    aRect.Set(20, 20, 200, 60);
    aWindow = new MyHelloWindow(aRect);

    aRect.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(aRect, "HelloView");

    aWindow->AddChild(aView);

    aWindow->Show();
}
```

The above approach is a good one for a program that allows for the opening of windows that differ—two windows could use two different views. If MyHelloWorld defined a second view class, then a second window of the MyHelloWindow class type could be opened and an object of this different view could be attached to it.

Now consider a program that allows multiple windows to open, but these windows are initially to be identical. An example of such an application might be a graphics program that opens windows that each have the same tool palette along one edge. The palette could be a view object that consists of a number of icon buttons. Here it would make sense to use an approach that differs from the above. For such a program the view could be attached to the window in the window class constructor. That is, the code that is used to create a view and attach it to a window could be moved from the application constructor to the window constructor. If I were to use this approach for the MyHelloWorld program, the previously empty constructor for the MyHelloWindow class would now look like this:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    MyHelloView  *aView;

    frame.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(frame, "MyHelloView");

    AddChild(aView);
    Show();
}
```

For the MyHelloWorld program, the `MyHelloApplication` constructor would now hold less code, and would look like the version shown here:

```
MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    BRect           aRect;

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);
}
```

Now, when a new window is created in the application constructor, the `MyHelloWindow` constructor is responsible for creating a new view, adding the view to the new window, and then displaying the new window.

The new MyHelloWorld source code

Changing the MyHelloWorld program to use this new technique results in changes to two files: *MyHelloWindow.cpp* and *MyHelloWorld.cpp*. Here's how *MyHelloWindow.cpp* looks now:

```
// -----------------------------------------------------------
#ifndef _APPLICATION_H
#include <Application.h>
#endif
#ifndef MY_HELLO_VIEW_H
#include "MyHelloView.h"
#endif
#ifndef MY_HELLO_WINDOW_H
#include "MyHelloWindow.h"
#endif

MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    MyHelloView  *aView;

    frame.OffsetTo(B_ORIGIN);
    aView = new MyHelloView(frame, "MyHelloView");

    AddChild(aView);
    Show();
}


bool MyHelloWindow::QuitRequested()
{
    be_app->PostMessage(B_QUIT_REQUESTED);
    return(true);
}

// -----------------------------------------------------------
```

The *MyHelloWorld.cpp* file doesn't get any new code—it only gets code removed. Here's the new version of *MyHelloWorld.cpp*:

```
// --------------------------------------------------------
#ifndef MY_HELLO_WINDOW_H
#include "MyHelloWindow.h"
#endif
// removed inclusion of MyHelloView.h header file
#ifndef MY_HELLO_WORLD_H
#include "MyHelloWorld.h"
#endif

main()
{
    MyHelloApplication *myApplication;

    myApplication = new MyHelloApplication();
    myApplication->Run();

    delete(myApplication);
    return(0);
}


MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    BRect          aRect;
    // moved to MyHelloWindow constructor: MyHelloView variable declaration

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);
    // moved to MyHelloWindow constructor: the code to create view,
    // attach it to window, and show window
}

// --------------------------------------------------------
```

As is the case for all of this book's examples, you'll find a folder that holds the files for this new version of MyHelloWorld on the included CD-ROM. Make sure to compile and build an application from the project file to convince yourself that this latest version of the MyHelloWorld executable is the same as the Chapter 2 version.

Adding an Alert to HelloWorld

The Interface Kit defines classes for the common interface elements. There's the `BWindow` class for a window, the `BMenuBar` class for a menubar, the `BMenu` class for a menu, the `BMenuItem` class for a menu item, and so forth. When you want to add an interface element to a program, you rely on an Interface Kit class to create an object that represents that element. Consider a program that is to include an alert. Armed with the above knowledge, it's a pretty safe guess that the Interface Kit defines a `BAlert` class to ease the task of creating alerts.

The BAlert class description

The `BAlert` class is a simple one: it consists of a constructor and a handful of member functions. The following is the `BAlert` class definition (less its private data members, which you won't use) from the *Alert.h* header file:

```
class  BAlert : public BWindow
{
public:
                      BAlert(const char *title,
                             const char *text,
                             const char *button1,
                             const char *button2 = NULL,
                             const char *button3 = NULL,
                             button_width width = B_WIDTH_AS_USUAL,
                             alert_type type = B_INFO_ALERT);

virtual               ~BAlert();

                      BAlert(BMessage *data);
static   BArchivable  *Instantiate(BMessage *data);
virtual  status_t     Archive(BMessage *data, bool deep = true) const;

         void         SetShortcut(int32 button_index, char key);
         char         Shortcut(int32 button_index) const;

         int32        Go();
         status_t     Go(BInvoker *invoker);

virtual  void         MessageReceived(BMessage *an_event);
virtual  void         FrameResized(float new_width, float new_height);
         BButton      *ButtonAt(int32 index) const;
         BTextView    *TextView() const;

virtual  BHandler     *ResolveSpecifier(BMessage *msg,
                                        int32 index,
                                        BMessage *specifier,
                                        int32 form,
                                        const char *property);
virtual  status_t     GetSupportedSuites(BMessage *data);

static   BPoint       AlertPosition(float width, float height);
...
}
```

The *Overview* section of the `BAlert` class description in the Be Book places you on familiar ground by letting you know that an alert is nothing more than a window with some text and one or more buttons in it. In fact, the overview states that you could forego the BAlert class altogether and use BWindow objects in their place. You *could* do that, but you won't want to. The BAlert class takes care of creating the alert window, establishing views within it, creating and displaying text and button objects in the alert, and displaying the alert upon creation.

The *Overview* section tells you the two steps you need to take to display an alert: construct a BAlert object using the new operator and the BAlert constructor, then invoke the Go() member function to display the new alert window.

The *Constructor* section of the BAlert class description provides the details of the variable parameter list of the BAlert constructor. While the constructor can have up to seven parameters, it can be invoked with fewer—only the first three parameters are required. Here's the BAlert constructor:

```
BAlert(const char *title,
       const char *text,
       const char *button1,
       const char *button2 = NULL,
       const char *button3 = NULL,
       button_width width = B_WIDTH_AS_USUAL,
       alert_type type = B_INFO_ALERT);
```

The required parameters specify the alert's title, its text, and a title for the alert's mandatory button. The first parameter is a string that represents the alert's title. While an alert doesn't have a tab as an ordinary window does, a title is nonetheless required. The second parameter is a string that represents the text that is to appear in the alert. The third parameter is a string (such as "OK," "Done," or "Accept") that is to be used as the title of a button that appears in the alert. This button is required so that the user has a means of dismissing the alert.

The fourth through seventh parameters have default values assigned to them so that a call to the constructor can omit any or all of them. That is, an alert can be constructed by passing just three strings to the BAlert constructor—illustrated in this snippet:

```
BAlert  *alert;

alert = new BAlert("Alert", "Close the My Hello window to quit", "OK");
```

The fourth and fifth parameters are optionally used for the text of up to two more buttons that will appear in the alert. The number of buttons in an alert varies depending on the number of strings passed to the BAlert constructor. By default, the second and third button titles are NULL, which tells BAlert to place only a single button in the alert. A Be-defined constant can be used as the sixth parameter to BAlert(). This constant specifies the width of the alert's buttons. By default, the width of each button will be a standard size (B_WIDTH_AS_USUAL) that is capable of holding most button titles. Providing a different constant here changes the width of each button. The final parameter specifies the icon to be displayed in the upper-left of the alert. By default a lowercase "i" (for "information") is used (as denoted by the Be-defined `B_INFO_ALERT` constant).

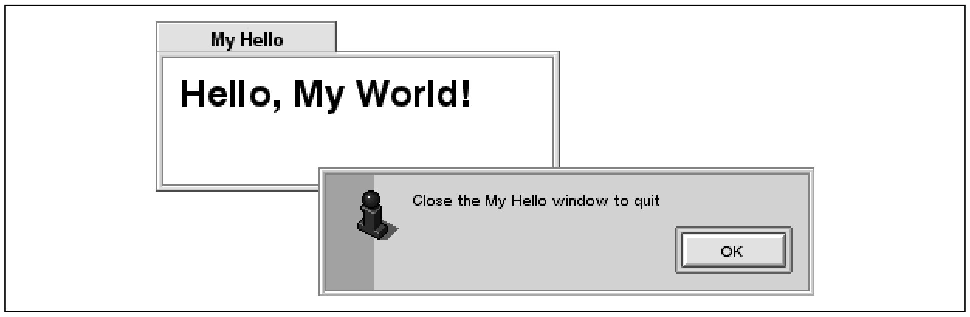
The Alert program

In the *C03* folder on this book's CD-ROM, you'll find two folders: *MyHelloWorld* and *Alert*. The *MyHelloWorld* folder holds the new version of the MyHelloWorld project—the version just described. The *Alert* folder also holds a version of the MyHelloWorld project. The only difference between the two projects appears in the *MyHelloApplication* constructor in the *MyHelloWorld.cpp* file of the alert example. In the alert example, I've added four lines of code, shown in bold type here:

```
MyHelloApplication::MyHelloApplication()
    : BApplication('myWD')
{
    MyHelloWindow  *aWindow;
    MyHelloView    *aView;
    BRect           aRect;
    BAlert         *alert;
    long            result;

    aRect.Set(20, 20, 250, 100);
    aWindow = new MyHelloWindow(aRect);

    alert = new BAlert("", "Close the My Hello window to quit.", "OK");
    result = alert->Go();
}
```

This latest incarnation of MyHelloWorld will serve as the base from which the remainder of this book's examples are built. To make it easy to recognize the code that doesn't change from example to example (which is most of the code), I won't rename the files and classes for each example. That means you'll always find the familiar `MyHelloView`, `MyHelloWindow`, and `HelloApplication` classes. As exhibited in the above snippet, the differences between one version of MyHelloWorld and another will consist of minimal code changes or additions. My goal in this book is to present short examples that consist mostly of existing, thoroughly discussed code, with only a minimum amount of new code. The new code will be, of course, code that is pertinent to the topic at hand. To distinguish one MyHelloWorld project from the next, I'll simply rename the project folder to something descriptive of the example. For instance, the alert example discussed here appears in a folder titled *Alert*.

In the above call to `BAlert()`, the first parameter is an empty string that represents the alert's title. An alert doesn't have a tab as a window does, but a title is required. Passing an empty string suffices here. The second parameter is a string that represents the text that is to appear in the alert. While this trivial example exists only to show how the alert code integrates into a program, I did want the alert to have at least some bearing on the program—so I've made the alert serve as a means of letting the user know how to quit the program. The final parameter is the name that appears on the alert's button. For a one-button alert, OK is typically used. Figure 3-2 shows what the alert example program looks like when running.



Close the My Hello window to quit

Figure 3-2. An alert overlapping a window

4

In this chapter:
- *Windows*
- *Views*
- *Messaging*

Windows, Views, and Messages

A window serves as a program's means of communicating with the user. In order to provide information to a user, a window needs to be able to draw either text or graphics. And in order to receive information from a user, a window needs to be aware of user actions such as mouse button clicks or key presses. Views make both these modes of communication possible. All drawing takes place in views. And views are recipients of messages that are transmitted from the Application Server to the program in response to user actions. All three of these topics—windows, views, and messages—can be discussed individually, and this chapter does just that. To be of real use, though, the interaction of these topics must be described; this chapter of course does that as well.

Windows

Your program's windows will be objects of a class, or classes, that your project derives from the `BWindow` class. The `BWindow` class is one of many classes in the Interface Kit—the largest of the Be kits. Most other Interface Kit class objects draw to a window, so they expect a `BWindow` object to exist—they work in conjunction with the window object.

Because it is a type of `BLooper`, a `BWindow` object runs in its own thread and runs its own message loop. This loop is used to receive and respond to messages from the Application Server. In this chapter's "Messaging" section, you'll see how a window often delegates the handling of a message to one of the views present in the window. The ever-present interaction of windows, views, and messages accounts for the combining of these three topics in this chapter.

Window Characteristics

A window's characteristics—its size, screen location, and peripheral elements (close button, zoom button, and so forth)—are all established in the constructor of the BWindow-derived class of the window.

BWindow constructor

A typical BWindow-derived class constructor is often empty:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
}
```

The purpose of the constructor is to pass window size and window screen location on to the BWindow constructor. In this next snippet, this is done by invoking the MyHelloWindow constructor, using the BRect parameter frame as the first argument in the BWindow constructor:

```
MyHelloWindow   *aWindow;
BRect            aRect(20, 30, 250, 100);

aWindow = new MyHelloWindow(aRect);
```

It is the BWindow constructor that does the work of creating a new window. The four BWindow constructor parameters allow you to specify the window's:

- Size and screen placement
- Title
- Type or look
- Behavioral and peripheral elements

The BWindow constructor prototype, shown here, has four required parameters and an optional fifth. Each of the five parameters is discussed following this prototype:

```
BWindow(BRect       frame,
        const char  *title,
        window_type type,
        ulong       flags,
        ulong       workspaces = B_CURRENT_WORKSPACE)
```

Window size and location (frame argument)

The first BWindow constructor parameter, frame, is a rectangle that defines both the size and screen location of the window. The rectangle's coordinates are relative to the screen's coordinates. The top left corner of the screen is point (0, 0), and coordinate values increase when referring to a location downward or rightward. For instance, the lower right corner of a 640×480 screen has a screen coordinate point of (639, 479). Because the initialization of a `BRect` variable is specified in the order left, top, right, bottom; the following declaration results in a variable that can be used to create a window that has a top left corner fifty pixels from the top of the user's screen and seventy pixels in from the left of that screen:

```
BRect  frame(50, 70, 350, 270);
```

The width of the window based on `frame` is determined simply from the delta of the first and third `BRect` initialization parameters, while the height is the difference between the second and fourth. The above declaration results in a rectangle that could be used to generate a window 301 pixels wide by 201 pixels high. (The "extra" pixel in each direction is the result of zero-based coordinate systems.)

The frame coordinates specify the content area of a window—the window's title tab is not considered. For titled windows, you'll want to use a top coordinate of at least 20 so that none of the window's title tab ends up off the top of the user's screen.

If your program creates a window whose size depends on the dimensions of the user's screen, make use of the `BScreen` class. A `BScreen` object holds information about one screen, and the `BScreen` member functions provide a means for your program to obtain information about this monitor. Invoking `Frame()`, for instance, returns a `BRect` that holds the coordinates of the user's screen. This next snippet shows how this rectangle is used to determine the width of a monitor:

```
BScreen  mainScreen(B_MAIN_SCREEN_ID);
BRect    screenRect;
int32    screenWidth;

screenRect = mainScreen->Frame();
screenWidth = screenRect.right - screenRect.left;
```

As of this writing, the BeOS supports only a single monitor, but the above snippet anticipates that this will change. The Be-defined constant `B_MAIN_SCREEN_ID` is used to create an object that represents the user's main monitor (the monitor that displays the Deskbar). Additionally, the width of the screen can be determined by subtracting the left coordinate from the right, and the height by subtracting the top from the bottom. On the main monitor, the `left` and `top` fields of the `BRect` returned by `Frame()` are 0, so the `right` and `bottom` fields provide the width and height of this screen. When an additional monitor is added, though, the `left` and `top` fields will be non-zero; they'll pick up where the main screen "ends."

Window title

The second `BWindow` constructor argument, `title`, establishes the title that is to appear in the window's tab. If the window won't display a tab, this parameter value is unimportant—you can pass `NULL` or an empty string (`""`) here (though you may want to include a name in case your program may eventually access the window through scripting.

Window type

The third `BWindow` constructor parameter, `type`, defines the style of window to be created. Here you use one of five Be-defined constants:

`B_DOCUMENT_WINDOW`
: Is the most common type, and creates a nonmodal window that has a title tab. Additionally, the window has right and bottom borders that are thinner than the border on its other two sides. This narrower border is designed to integrate well with the scrollbars that may be present in such a window.

`B_TITLED_WINDOW`
: Results in a nonmodal window that has a title tab.

`B_MODAL_WINDOW`
: Creates a modal window, a window that prevents other application activity until it is dismissed. Such a window is also referred to as a dialog box. A window of this type has no title tab.

`B_BORDERED_WINDOW`
: Creates a nonmodal window that has no title tab.

`B_FLOATING_WINDOW`
: Creates a window that floats above (won't be obscured by) other application windows.

There's another version of the `BWindow` constructor that has two parameters (`look` and `feel`) in place of the one `type` parameter discussed above. The separate look and feel parameters provide a means of more concisely stating just how a window is to look and behave. The single type parameter can be thought of as a shorthand notation that encapsulates both these descriptions. Refer to the `BWindow` class section of the Interface Kit chapter of the Be Book for more details (and a list of Be-defined `look` and `feel` constants).

Window behavior and elements

The fourth `BWindow` constructor argument, `flags`, determines a window's behavior (such as whether the window is movable) and the window's peripheral elements (such as the presence of a title tab or zoom button). There are a number of Be-defined constants that can be used singly or in any combination to achieve the desired window properties. To use more than a single constant, list each and separate them with the OR (|) operator. The following example demonstrates how to create a window that has no zoom button or close button:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, windowName, B_TITLED_WINDOW, B_NOT_ZOOMABLE | B_NOT_
CLOSABLE)
{
}
```

If you use 0 (zero) as the fourth parameter, it serves as a shortcut for specifying that a window include all the characteristics expected of a titled window. Default windows are movable, resizable, and have close and zoom buttons:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    :BWindow(frame, windowName, B_TITLED_WINDOW, 0)
{
}
```

The following briefly describes many of the several Be-defined constants available for use as the fourth parameter in the `BWindow` constructor:

`B_NOT_MOVABLE`
: Creates a window that cannot be moved—even if the window has a title tab. By default, a window with a title tab is movable.

`B_NOT_H_RESIZABLE`
: Generates a window that can't be resized horizontally. By default, a window can be resized both horizontally and vertically.

`B_NOT_V_RESIZABLE`
: Generates a window that can't be resized vertically. By default, a window can be resized both horizontally and vertically.

`B_NOT_RESIZABLE`
: Creates a window that cannot be resized horizontally or vertically.

`B_NOT_CLOSABLE`
: Results in a window that has no close button. By default, a window with a title tab has a close button.

`B_NOT_ZOOMABLE`
: Results in a window that has no zoom box. By default, a window with a title tab has a zoom box.

`B_NOT_MINIMIZABLE`
: Defines a window that cannot be minimized (collapsed). By default, a window can be minimized by double-clicking on the window's title bar.

`B_WILL_ACCEPT_FIRST_CLICK`
: Results in a window that is aware of mouse button clicks in it—even when the window isn't frontmost. By default, a window is aware only of mouse button clicks that occur when the window is the frontmost, or active, window.

Workspace

The `BWindow` constructor has an optional fifth parameter, `workspaces`, that specifies which workspace or workspaces should contain the new window. Desktop information such as screen resolution and color depth (number of bits of color data per pixel) can be adjusted by the user. Different configurations can be saved to different workspaces. Workspaces can be thought of as virtual monitors to which the user can switch. Under different circumstances, a user may wish to display different types of desktops. By omitting this parameter, you tell the `BWindow` constructor to use the default Be-defined constant `B_CURRENT_WORKSPACE`. Doing so means the window will show up in whatever workspace is currently selected by the user. To create a window that appears in all of the user's workspaces, use the Be-defined constant `B_ALL_WORKSPACES` as the fifth parameter to the `BWindow` constructor.

You can find out more about workspaces from the user's perspective in the BeOS User's Guide, and from the programmer's perspective in the `BWindow` constructor section of the Interface Kit chapter of the Be Book.

Accessing Windows

Fortunately for you, the programmer, the Be operating system takes care of much of the work in keeping track of your application's windows and the user's actions that affect those windows. There will be times, however, when you'll need to directly manipulate one or all of your program's windows. For instance, you may want to access the frontmost window to draw to it, or access all open windows to implement a Close All menu item.

The Application Server keeps a list that holds references to an application's open windows. The list indices begin at 0, and continue integrally. The windows aren't entered in this list in any predefined order, so you can't rely on a particular index referencing a particular window. You can, however, use the `BApplication` member function `WindowAt()` to find any given window.

Accessing a window using WindowAt()

`WindowAt()` accepts a single argument, an integer that serves as a window list index. Calling `WindowAt()` returns the `BWindow` object this index references. A call to `WindowAt()` returns the first window in the list:

```
BWindow  *aWindow;

aWindow = be_app->WindowAt(0);
```

From Chapter 1, *BeOS Programming Overview*, you know that the Be-defined global variable `be_app` always points to the active application, so you can use it anywhere in your code to invoke a `BApplication` member function such as `WindowAt()`.

When `WindowAt()` is passed a value that is an out-of-bounds index, the routine returns `NULL`. You can use this fact to create a simple loop that accesses each open window:

```
BWindow  *theWindow;
int32    i = 0;

while (theWindow = be_app->WindowAt(i++)) {
   // do something, such as close theWindow
}
```

The preceding loop starts at window 0 in the window list and continues until the last window in the list is reached.

A good use for the `WindowAt()` loop is to determine the frontmost window. The `BWindow` member function `IsFront()` returns a `bool` (Boolean) value that indicates whether a window is frontmost. If you set up a loop to cycle through each open window and invoke `IsFront()` for each returned window, the frontmost window will eventually be encountered:

```
BWindow  *theWindow;
BWindow  *frontWindow = NULL;
int32    i = 0;

while (theWindow = be_app->WindowAt(i++)) {
   if (theWindow->IsFront())
      frontWindow = theWindow;
}
```

In the preceding snippet, note that `frontWindow` is initialized to `NULL`. If no windows are open when the loop runs, `frontWindow` will retain the value of `NULL`, properly indicating that no window is frontmost.

Frontmost window routine

With the exception of `main()`, all the functions you've encountered to this point have been part of the BeOS API—they've all been Be-defined member functions of Be-defined classes. Your nontrivial projects will also include application-defined member functions, either in classes you define from scratch or in classes you derive from a Be-defined class. Here I provide an example of this second category of application-defined routine. The `MyHelloApplication` class is derived from the Be-defined `BApplication` class. This version of `MyHelloApplication` adds a new application-defined routine to the class declaration:

```
class MyHelloApplication : public BApplication {

   public:
                  MyHelloApplication();
      BWindow *   GetFrontWindow();
};
```

The function implementation is familiar to you—it's based on the previous snippet that included a loop that repeatedly calls `AtWindow()`:

```
BWindow * MyHelloApplication::GetFrontWindow()
{
   BWindow  *theWindow;
   BWindow  *frontWindow = NULL;
   int32    i = 0;

   while (theWindow = be_app->WindowAt(i++)) {
      if (theWindow->IsFront())
         frontWindow = theWindow;
   }
   return frontWindow;
}
```

When execution of `GetFrontWindow()` ends, the routine returns the `BWindow` object that is the frontmost window. Before using the returned window, typecast it to the `BWindow`-derived class that matches its actual type, as in:

```
MyHelloWindow  *frontWindow;

frontWindow = (MyHelloWindow *)GetFrontWindow();
```

With access to the frontmost window attained, any `BWindow` member function can be invoked to perform some action on the window. Here I call the `BWindow` member function `MoveBy()` to make the frontmost window jump down and to the right 100 pixels in each direction:

```
frontWindow->MoveBy(100, 100);
```

Frontmost window example project

I've taken the preceding `GetFrontWindow()` routine and included it in a new version of MyHelloWorld. To test out the function, I open three MyHelloWorld windows, one directly on top of another. Then I call `GetFrontWindow()` and use the returned `BWindow` reference to move the frontmost window off the other two. The result appears in Figure 4-1.

```
MyHelloApplication::MyHelloApplication()
   : BApplication("application/x-vnd.dps-mywd")
{
   MyHelloWindow  *aWindow;
   BRect          aRect;
   MyHelloWindow  *frontWindow;

   aRect.Set(20, 30, 250, 100);
   aWindow = new MyHelloWindow(aRect);
   aWindow = new MyHelloWindow(aRect);
   aWindow = new MyHelloWindow(aRect);

   frontWindow = (MyHelloWindow *)GetFrontWindow();
   if (frontWindow)
      frontWindow->MoveBy(100, 100);
}
```

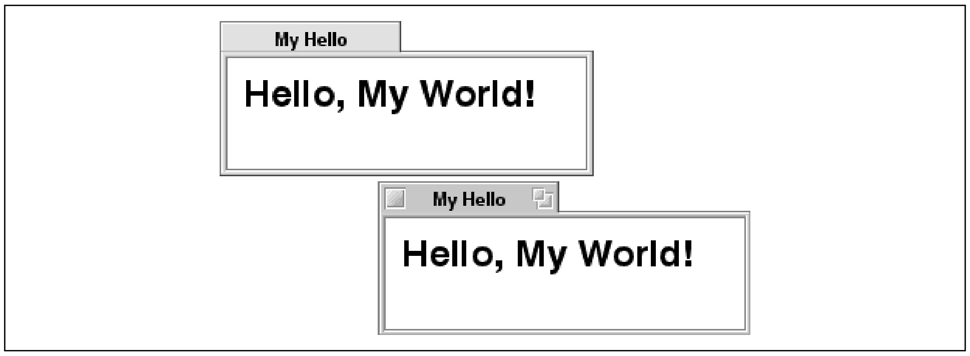


Figure 4-1. The result of running the FrontWindow program

Notice that before working with the returned window reference, I verify that it has a non-`NULL` value. If no windows are open when `GetFrontWindow()` is invoked, that routine returns `NULL`. In such a case, a call to a `BWindow` member function such as `MoveBy()` will fail.

The `MyHelloWindow` class doesn't define any of its own member functions—it relies on `BWindow`-inherited functions. So in this example, I could have declared `frontWindow` to be of type `BWindow` and omitted the typecasting of the returned `BWindow` reference. This code would still work:

```
...
BWindow  *frontWindow;
```

```
    ...
    frontWindow = GetFrontWindow();
       if (frontWindow)
          frontWindow->MoveBy(100, 100);
    }
```

But instead of working with the returned reference as a `BWindow` object, I opted to typecast it to a `MyHelloWindow` object. That's a good habit to get into—the type of window being accessed is then evident to anyone looking at the source code listing. It also sets up the returned object so that it can invoke any `BWindow`-derived class member function. A `BWindow` object knows about only `BWindow` functions, so if I define a `SpinWindow()` member function in the `MyHelloWindow` class and then attempt to call it without typecasting the `GetFrontWindow()`-returned `BWindow` reference, the compiler will complain:

```
BWindow  *frontWindow;

frontWindow = GetFrontWindow();
if (frontWindow)
   frontWindow->SpinWindow();    // compilation error at this line
```

The corrected version of the above snippet looks like this:

```
MyHelloWindow  *frontWindow;

frontWindow = (MyHelloWindow *)GetFrontWindow();
if (frontWindow)
   frontWindow->SpinWindow();    // compiles just fine!
```

Windows and Data Members

Defining a `GetFrontWindow()` or some similar member function to locate a window is one way to access a window. If you have only one instance of any given window class in your program, though, you should consider using a technique that stores window references in data members in the application object.

Defining a window object data member in the application class

For each type of window in your application, you can add to the class definition a private data member of the window class type. Consider a program that displays two windows: an input window for entering a mathematical equation, and an output window that displays a graph of the entered equation. If such a program defines `BWindow`-derived classes named `EquationWindow` and `GraphWindow`, the `BApplication`-derived class could include two data members. As shown below, Be convention uses a lowercase *f* as the first character of a data member name:

```
class MathApp : public BApplication {

   public:
                      MathApp();
```

```
                     ...
   private:
      EquationWindow  *fEquationWindow;
      GraphWindow     *fGraphWindow;
};
```

For the MyHelloWorld project, the `MyHelloApplication` class is defined as:

```
class MyHelloApplication : public BApplication {

   public:
                     MyHelloApplication();

   private:
      MyHelloWindow  *fMyWindow;
};
```

Storing a window object in the data member

In past examples, I created an instance of a window by declaring a local window variable in the application constructor, then using that variable in a call to the window's class constructor:

```
MyHelloWindow  *aWindow;
...
aWindow = new MyHelloWindow(aRect);
```

With the new technique, there's no need to use a local variable. Instead, assign the object returned by the window constructor to the window data member. The new version of the `MyHelloApplication` class defines an `fMyWindow` data member, so the result would be:

```
fMyWindow = new MyHelloWindow(aRect);
```

Here's how the new version of the `MyHelloApplication` constructor looks:

```
MyHelloApplication::MyHelloApplication()
   : BApplication("application/x-vnd.dps-mywd")
{
   BRect  aRect;

   aRect.Set(20, 30, 250, 100);
   fMyWindow = new MyHelloWindow(aRect);
}
```

Once created, the new window can be accessed from any application member function. For instance, to jump the window across part of the screen requires only one statement:

```
fMyWindow->MoveBy(100, 100);
```

Window object data member example projects

This chapter's `MyHelloWorld` project consists of the new version of the `MyHelloApplication` class—the version that includes an `fMyWindow` data member. The executable built from this project is indistinguishable from that built from prior versions of the project; running the program results in the display of a single window that holds the string "Hello, My World!"

The WindowTester project picks up where MyHelloWorld leaves off. Like MyHelloWorld, it includes an `fMyWindow` data member in the `MyHelloApplication` class. The WindowTester version of the `MyHelloApplication` class also includes a new application-defined member function:

```
class MyHelloApplication : public BApplication {

   public:
                     MyHelloApplication();
      void           DoWindowStuff();

   private:
      MyHelloWindow  *fMyWindow;
};
```

After creating a window and assigning it to the `fMyWindow` data member, the `MyHelloApplication` constructor invokes `DoWindowStuff()`:

```
MyHelloApplication::MyHelloApplication()
   : BApplication("application/x-vnd.dps-mywd")
{
   BRect  aRect;

   aRect.Set(20, 30, 250, 100);
   fMyWindow = new MyHelloWindow(aRect);

   DoWindowStuff();
}
```

I've implemented `DoWindowStuff()` such that it glides the program's one window diagonally across the screen:

```
void  MyHelloApplication::DoWindowStuff()
{
   int16  i;

   for (i=0; i<200; i++) {
      fMyWindow->MoveBy(1, 1);
   }
}
```

Feel free to experiment by commenting out the code in `DoWindowStuff()` and replacing it with code that has `fMyWindow` invoke `BWindow` member functions other than `MoveBy()`. Refer to the `BWindow` section of the Interface Kit chapter of the Be Book for the details on such `BWindow` member functions as `Close()`, `Hide()`, `Show()`, `Minimize()`, `ResizeTo()`, and `SetTitle()`.

Views

A window always holds one or more views. While examples up to this point have all displayed windows that include only a single view, real-world Be applications make use of windows that often consist of a number of views. Because all drawing must take place in a view, everything you see within a window appears in a view. A scrollbar, button, picture, or text lies within a view. The topic of drawing in views is significant enough that it warrants its own chapter—Chapter 5, *Drawing*. In this chapter, the focus will be on how views are created and accessed. Additionally, you'll get an introduction to how a view responds to a message.

A view is capable of responding to a message sent from the Application Server to a `BWindow` object and then on to the view. This messaging system is the principle on which controls such as buttons work. The details of working with controls are saved for Chapter 6, *Controls and Messages*, but this chapter ends with a discussion of views and messages that will hold you over until you reach that chapter.

Accessing Views

You've seen that a window can be accessed by storing a reference to the window in the `BApplication`-derived class (as demonstrated with the `fMyWindow` data member) or via the BeOS API (through use of the `BApplication` member function `WindowAt()`). A similar situation exists for accessing a view.

Views and data members

Just as a reference to a window can be stored in an application class data member, a reference to a view can be stored in a window class data member. The MyHelloWorld project defines a single view class named `MyHelloView` that is used with the project's single window class, the `MyHelloWindow` class. Here I'll add a `MyHelloView` reference data member to the `MyHelloWindow` class:

```
class MyHelloWindow : public BWindow {

   public:
```

```
                      MyHelloWindow(BRect frame);
         virtual bool    QuitRequested();

      private:
         MyHelloView  *fMyView;
   };
```

Using this new technique, a view can be added to a new window in the window's constructor, much as you've seen in past examples. The `MyHelloWindow` constructor creates a new view, and a call to the `BWindow` member function `AddChild()` makes the view a child of the window:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   Show();
}
```

The window's view can now be easily accessed and manipulated from any `MyHelloWindow` member function.

View data member example projects

This chapter's NewMyHelloWorld project includes the new versions of the `MyHelloWindow` class and the `MyHelloWindow` constructor—the versions developed above. Once again, performing a build on the project results in an executable that displays a single "Hello, My World!" window. This is as expected. Using a data member to keep track of the window's one view simply sets up the window for easy access to the view—it doesn't change how the window or view behaves.

The ViewDataMember project serves as an example of view access via a data member—the `fMyView` data member that was just added to the NewMyHelloWorld project. Here's how the ViewDataMember project defines the `MyHelloWindow` class:

```
class MyHelloWindow : public BWindow {

   public:
                      MyHelloWindow(BRect frame);
      virtual bool    QuitRequested();
      void            SetHelloViewFont(BFont newFont, int32 newSize);

   private:
      MyHelloView  *fMyView;
};
```

The difference between this project and the previous version is that this project uses the newly added `SetHelloViewFont()` member function to set the type and size of the font used in a view. In particular, the project calls this routine to set the characteristics of the font used in the `MyHelloView` view that the `fMyView` data member references. Here's what the `SetHelloViewFont()` implementation looks like:

```
void MyHelloWindow::SetHelloViewFont(BFont newFont, int32 newSize)
{
   fMyView->SetFont(&newFont);
   fMyView->SetFontSize(newSize);
}
```

`SetFont()` and `SetFontSize()` are `BView` member functions with which you are familiar—they're both invoked from the `MyHelloView AttachedToWindow()` function, and were introduced in Chapter 2, *BeIDE Projects*.

To change a view's font, `SetHelloViewFont()` is invoked by a `MyHelloWindow` object. To demonstrate its use, I chose to include the call in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   BFont  theFont = be_plain_font;
   int32  theSize = 12;
   SetHelloViewFont(theFont, theSize);

   Show();
}
```

The call to `SetHelloViewFont()` results in the about-to-be shown window having text characteristics that include a font type of plain and a font size of 12. Figure 4-2 shows the results of creating a new window. While `SetHelloViewFont()` is a trivial routine, it does the job of demonstrating view access and the fact that characteristics of a view can be changed at any time during a program's execution.

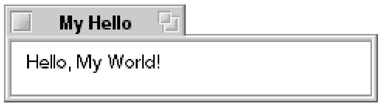


Figure 4-2. The ViewDataMember window displays text in a 12-point plain font

A More Practical Use For SetHelloViewFont()

Attaching a view to a window by calling `AddChild()` automatically invokes the view's `AttachedToWindow()` routine to take care of any final view setup. Recall that the `MyHelloView` class overrides this `BView` member function and invokes `SetFont()` and `SetFontSize()` in the `AttachedToWindow()` implementation:

```
void MyHelloView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

So it turns out that in the above version of the `MyHelloWindow` constructor, the view's font information is set twice, almost in succession. The result is that when the view is displayed, the last calls to `SetFont()` and `SetFontSize()` are used when drawing in the view, as shown in Figure 4-2.

Because this example project has very few member functions (intentionally, to keep it easily readable), I'm limited in where I can place a call to `SetHelloViewFont()`. In a larger project, a call to `SetHelloViewFont()` might be invoked from the code that responds to, say, a button click or a menu item selection. After reading Chapter 6 and Chapter 7, *Menus*, you'll be able to easily try out one of these more practical uses for a routine such as `SetHelloViewFont()`.

Accessing a view using FindView()

When a view is created, one of the arguments passed to the view constructor is a string that represents the view's name:

```
fMyView = new MyHelloView(frame, "MyHelloView");
```

The `MyHelloView` class constructor invokes the `BView` constructor to take care of the creation of the view. When it does that, it in turn passes on the string as the second argument, as done here:

```
MyHelloView::MyHelloView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

If your code provides each view with a unique name, access to any particular view can be easily gained by using the `BWindow` member function `FindView()`. For instance, in this next snippet a pointer to the previously created view with the name "MyHelloView" is being obtained. Assume that the following code is called from within a `MyHelloApplication` member function, and that a window has already been created and a reference to it stored in the `MyHelloApplication` data member `fMainWindow`:

```
MyHelloView  *theView;

theView = (MyHelloView *)fMainWindow->FindView("MyHelloView");
```

`FindView()` returns a `BView` object. The above snippet typecasts this `BView` object to one that matches the exact type of view being referenced—a `MyHelloView` view.

FindView() example project

The FindByName project does just that—it finds a view using a view name. This project is another version of this chapter's MyHelloWorld. Here I keep track of the program's one window using a data member in the `MyHelloApplication` class. A reference to the program's one view isn't, however, stored in a data member in the `MyHelloWindow` class. Instead, the view is accessed from the window using a call to `FindView()`. Here's the `MyHelloWindow` constructor that creates a view named "MyHelloView" and adds it to a new window:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   MyHelloView  *aView;

   frame.OffsetTo(B_ORIGIN);
   aView = new MyHelloView(frame, "MyHelloView");
   AddChild(aView);

   Show();
}
```

The `MyHelloWindow` member function `QuitRequested()` has remained unchanged since its introduction in Chapter 1. All it did was post a `B_QUIT_REQUESTED` and return `true`. I'll change that by adding a chunk of code. Figure 4-3 shows how the program's window looks just before closing.

```
bool MyHelloWindow::QuitRequested()
{
   MyHelloView  *aView;
   bigtime_t    microseconds = 1000000;

   aView = (MyHelloView *)FindView("MyHelloView");
   if (aView) {
      aView->MovePenTo(BPoint(20, 60));
      aView->DrawString("Quitting...");
      aView->Invalidate();
   }
```

```
        snooze(microseconds);

        be_app->PostMessage(B_QUIT_REQUESTED);
        return(true);
    }
```

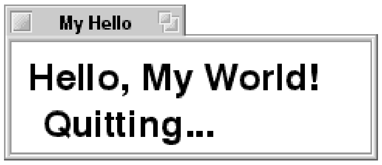


Figure 4-3. The FindByName program adds text to a window before closing it

The new version of `QuitRequested()` now does the following:

- Accesses the view named "MyHelloView."
- Calls a few `BView` member functions to draw a string and update the view.
- Pauses for one second.
- Closes the window and quits.

Several lines of code are worthy of further discussion.

The "Accessing a view using FindView()" section in this chapter demonstrates the use of `FindView()` from an existing window object:

```
MyHelloView  *theView;

theView = (MyHelloView *)fMainWindow->FindView("MyHelloView");
```

This latest example demonstrates the use of `FindView()` from within a window member function. The specific object `FindView()` acts on is the one invoking `QuitRequested()`, so unlike the above example, here no `MyHelloWindow` object variable precedes the call to `FindView()`:

```
MyHelloView  *aView;

aView = (MyHelloView *)FindView("MyHelloView");
```

With a reference to the `MyHelloView` object, `QuitRequested()` can invoke any `BView` member function. `MovePenTo()` and `DrawString()` are functions you've seen before—they also appear in the `MyHelloView` member function `Draw()`. `Invalidate()` is new to you. When a view's contents are altered—as they are here with the writing of the string "Quitting..."—the view needs to be updated before the changes become visible onscreen. If the changes are made while the view's window is hidden, then the subsequent act of showing that window brings on the update. Here, with the window showing and frontmost, no update automatically occurs after the call to `DrawString()`. The `BView` member function `Invalidate()` tells the system that the current contents of the view are no longer valid and require updating. When the system receives this update message, it immediately obliges the view by redrawing it.

Finally, the `snooze()` function is new to you. The BeOS API includes a number of global, or nonmember, functions—`snooze()` is one of them. A global function isn't associated with any class or object, so once the `BApplication`-defined object is created in `main()`, it can be called from any point in a program. The `snooze()` function requires one argument, the number of microseconds for which execution should pause. The parameter is of type `bigtime_t`, which is a `typedef` equivalent to the `int64` datatype. Here, the first call to `snooze()` pauses execution for one million microseconds, or one second, while the second call pauses execution for fifty thousand microseconds, or one-twentieth of one second:

```
bigtime_t  microseconds = 1000000;

snooze(microseconds);
snooze(50000);
```

In this book I'll make occasional use of a few global functions. In particular, you'll see calls to `snooze()` and `beep()` in several examples. You'll quickly recognize a function as being global because it starts with a lowercase character. A global function is associated with one of the Be kits, so you'll find it documented in the *Global Functions* section of the appropriate kit chapter in the Be Book. For instance, `snooze()` puts a thread to sleep, so it's documented in the thread-related chapter of the Be Book, the Kernel Kit chapter. The `beep()` global function plays the system beep. Sound (and thus the `beep()` function) is a topic covered in the Media Kit chapter of the Be Book.

View Hierarchy

A window can hold any number of views. When a window holds more than one, the views fall into a hierarchy.

Top view

Every window contains at least one view, even if none is explicitly created and added with calls to `AddChild()`. That's because upon creation, a window is always automatically given a top view—a view that occupies the entire content area of the window. Even if the window is resized, the top view occupies the entire window content. A top view exists only to serve as an organizer, or container, of other views. The other views are added by the application. Such an application-added view maps out a window area that has its own drawing characteristics (such as font type and line width), is capable of being drawn to, and is able to respond to messages.

Application-added views and the view hierarchy

Each view you add to the window falls into a window view hierarchy. Any view that is added directly to the window (via a call to the `BWindow` member function `AddChild()`) falls into the hierarchy just below the top view. Adding a few views to a window in this way could result in a window and view hierarchy like those shown in Figure 4-4.

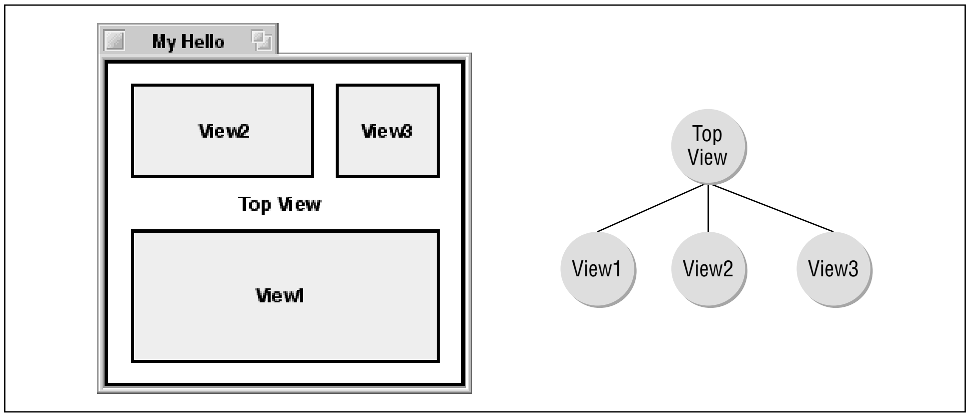


Figure 4-4. A window with three views added to it and that window's view hierarchy

When a view is added to a window, there is no visible sign that the view exists. So in Figure 4-4, the window's views—including the top view—are outlined and are named. The added views have also been given a light gray background. While the view framing, shading, and text have been added for clarity, you could in fact easily create a window that highlighted its views in this way. You already know how to add text to a view using `DrawString()`. Later in this chapter you'll see how to draw a rectangle in a view. And in Chapter 5 you'll see how to change the background color of a view.

The views you add to a window don't have to exist on the same hierarchical level; they can be nested one inside another. Figure 4-5 shows another window with three views added to the top view. Here, one view has been placed inside another.

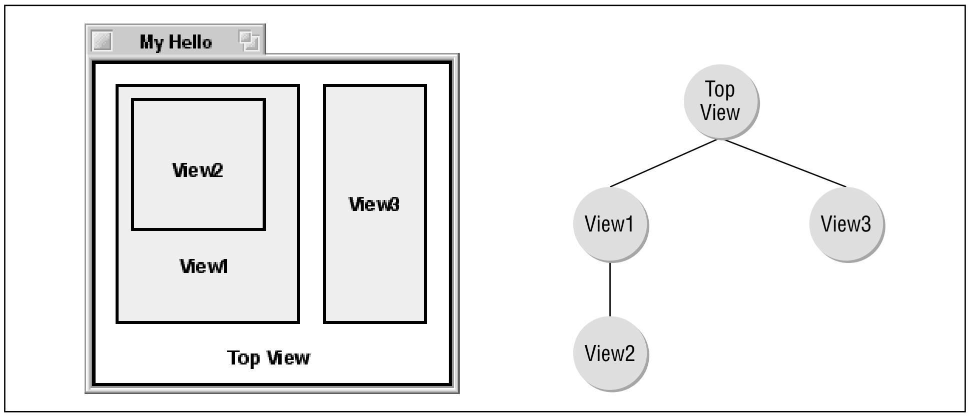


Figure 4-5. A window with nested views added to it and that window's view hierarchy

To place a view within another, you add the view to the container view rather than to the window. Just as the `BWindow` class has an `AddChild()` member function, so does the `BView` class. This next snippet shows a window constructor that creates two views. The first is 200 pixels by 300 pixels in size, and is added to the window. The second 150 pixels by 150 pixels, and is added to the first view.

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "Nested Views", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   BRect        viewFrame;
   MyHelloView  *view1;
   MyHelloView  *view2;

   viewFrame.Set(30, 30, 230, 330);
   view1 = new MyHelloView(viewFrame, "MyFirstView");
   AddChild(view1);

   viewFrame.Set(10, 10, 160, 160);
   view2 = new MyHelloView(viewFrame, "MySecondView");
   view1->AddChild(view2);

   Show();
}
```

Multiple views example project

Later in this chapter you'll see a few example projects that place two views of type `MyHelloView` in a window. Having the views be the same type isn't required, of course—they can be different class types. The TwoViewClasses project defines a view named `MyDrawView` and adds one such view to a window, along with an instance of the `MyHelloView` class with which you're already familiar. Figure 4-6 shows the window that results from running the TwoViewClasses program.

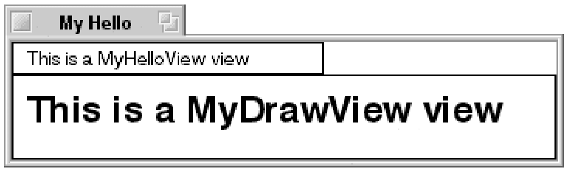
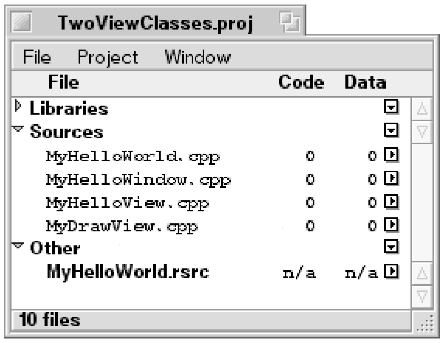


Figure 4-6. A window that holds two different types of views

In keeping with the informal convention of placing the code for a class declaration in its own header file and the code for the implementation of the member functions of that class in its own source code file, the TwoViewClasses project now has a new source code file added to it. Figure 4-7 shows the project window for this project.



```
File                   Code  Data
▷ Libraries
▽ Sources
   MyHelloWorld.cpp      0     0
   MyHelloWindow.cpp     0     0
   MyHelloView.cpp       0     0
   MyDrawView.cpp        0     0
▽ Other
   MyHelloWorld.rsrc    n/a   n/a
10 files
```

Figure 4-7. The TwoViewClasses project window shows the addition of a new source code file

I haven't shown a project window since Chapter 2, and won't show one again. I did it here to lend emphasis to the way in which projects are set up throughout this book (and by many other Be programmers as well).

I created the new class by first copying the *MyHelloView.h* and *MyHelloView.cpp* files and renaming them to *MyDrawView.h* and *MyDrawView.cpp*, respectively. My intent here is to demonstrate that a project can derive any number of classes from the `BView` class and readily mix them in any one window. So I'll only make a couple of trivial changes to the copied `MyHelloView` class to make it evident that this is a new class. In your own project, the `BView`-derived classes you define may be very different from one another.

With the exception of the class name and the name of the constructor, the `MyDrawView` class declaration is identical to the `MyHelloView` class declaration. From the *MyDrawView.h* file, here's that declaration:

```
class MyDrawView : public BView {

public:
                  MyDrawView(BRect frame, char *name);
virtual  void     AttachedToWindow();
virtual  void     Draw(BRect updateRect);
};
```

Like the `MyHelloView` constructor, the `MyDrawView` constructor is empty:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

The `MyDrawView` member function `AttachedToWindow()` sets up the view's font and font size. Whereas the `MyHelloView` specified a 12-point font, the `MyDrawView` specifies a 24-point font:

```
void MyHelloView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

Except for the text drawn to the view, the `MyDrawView` member function `Draw()` looks like the `MyHelloView` version of this function:

```
void MyDrawView::Draw(BRect)
{
   BRect frame = Bounds();
   StrokeRect(frame);

   MovePenTo(BPoint(10, 30));
   DrawString("This is a MyDrawView view");
}
```

To create a further contrast in the way the two views display text, I turned to the `MyHelloView` and made one minor modification. In the `AttachedToWindow()` member function of that class, I changed the font set by `SetFont()` from `be_bold_font` to `be_plain_font`. Refer to Figure 4-6 to see the difference in text appearances in the two views.

In order for a window to be able to reference both of the views it will hold, a new data member has been added to the `MyHelloWindow` class. In the *MyHelloWindow.h* header file, you'll find the addition of a `MyDrawView` data member named `fMyDrawView`:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual  bool     QuitRequested();

   private:
      MyHelloView       *fMyView;
      MyDrawView        *fMyDrawView;
};
```

In the past the `MyHelloWindow` constructor created and added a single view to itself. Now the constructor adds a second view:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.Set(0, 0, 200, 20);
   fMyView = new MyHelloView(frame, "MyHelloView");
   AddChild(fMyView);

   frame.Set(0, 21, 350, 300);
   fMyDrawView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyDrawView);

   Show();
}
```

Both views have been added directly to the window (to the top view), rather than to another view, so both views are on the same level in the window's view hierarchy. The `Draw()` function of each view type includes code to frame the view, so you can easily see the results of any view size changes you might make to the views here in the `MyHelloWindow` constructor.

Coordinate System

In order to specify where a window is to be placed on the screen and where a view is to be placed within a window, a coordinate system is required.

Global coordinate system

To allow a programmer to reference any point on the computer screen, Be defines a coordinate system that gives every pixel a pair of values: one for the pixel's distance from the left edge of the screen and one for the pixel's distance from the top of the screen. Figure 4-8 points out a few pixels and their corresponding coordinate pairs.

For display devices, the concept of fractional pixels doesn't apply. Consider a window that is to have its top left corner appear 100 pixels from the left edge of the screen and 50 pixels from the top of the screen. This point is specified as (100.0,

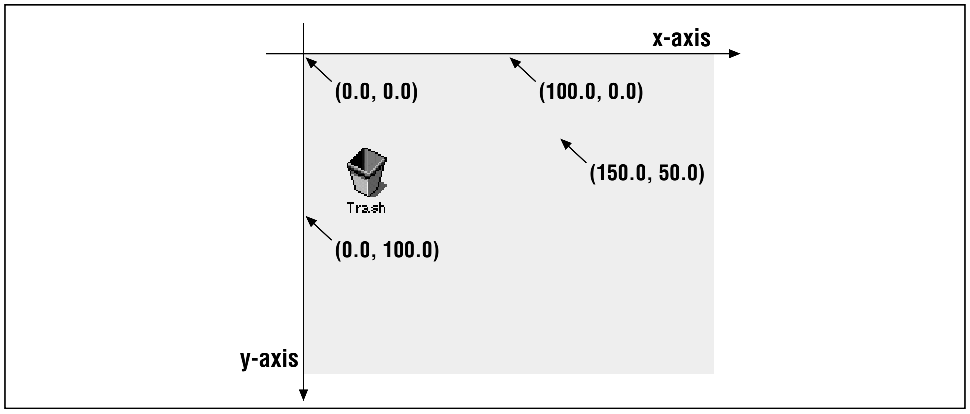


Figure 4-8. The global coordinate system maps the screen to a two-dimensional graph

50.0). If the point (100.1, 49.9) is used instead, the result is the same—the window's corner ends up 100 pixels from the left and 50 pixels from the top of the screen.

The above scenario begs the question: if the coordinates of pixel locations are simply rounded to integral values, why use floating points at all? The answer lies in the current state of output devices: most printers have high resolutions. On such a device, one coordinate unit doesn't map to one printed dot. A coordinate unit is always 1/72 of an inch. If a printer has a resolution of 72 dots per inch by 72 dots per inch (72 dpi × 72 dpi), then one coordinate unit would in fact translate to one printed dot. Typically printers have much higher resolutions, such as 300 dpi or 600 dpi. If a program specifies that a horizontal line be given a height of 1.3 units, then that line will occupy one row of pixels on the screen (the fractional part of the line height is rounded off). When that same line is sent to a printer with a resolution of 600 dpi, however, that printer will print the line with a height of 11 rows. This value comes from the fact that one coordinate unit translates to 8.33 dots (that's 1/72 of 600). Here there is no rounding of the fractional coordinate unit, so 1.3 coordinate units is left at 1.3 units (rather than 1 unit) and translates to 11 dots (1.3 times 8.33 is 10.83). Whether the line is viewed on the monitor or on hardcopy, it will have roughly the same look—it will be about 1/72 inch high. It's just that the rows of dots on a printer are denser than the rows of pixels on the monitor.

Window coordinate system

When a program places a view in a window, it does so relative to the window, not to the screen. That is, regardless of where a window is positioned on the screen when the view is added, the view ends up in the same location within the content area of the window. This is possible because a window has its own coordinate system—one that's independent of the global screen coordinate system. The type of system is the same as the global system (floating point values that get larger as you move right and down)—but the origin is different. The origin of a window's coordinate system is the top left corner of the window's content area.

When a program adds a view to a window, the view's boundary rectangle values are stated in terms of the window's coordinate system. Consider the following window constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   MyHelloView  *aView;
   BRect        viewFrame(20.0, 30.0, 120.0, 130.0);

   aView = new MyHelloView(viewFrame, "MyHelloView");
   AddChild(aView);

   Show();
}
```

The coordinate systems for the window and the view are different. The window's size and screen placement, which are established by the `BRect` variable `frame` that is passed to the constructor, are expressed in the global coordinate system. The view's size and placement, established by the local `BRect` variable `viewFrame`, are expressed in the window coordinate system. Regardless of where the window is placed, the view `aView` will have its top left corner at point (20.0, 30.0) within the window.

In all previous examples, the arguments to a `BRect` constructor, or to the `BRect` member function `Set()`, were integer values, such as (20, 30, 120, 130). Since none of the examples were concerned with high precision printouts, that technique worked fine. It also may have been comforting to you if you come from a Mac or Windows programming background, where rectangle boundaries use integral values. Now that we've seen the true nature of the BeOS coordinate system, however, we'll start—and continue—to use floating point values.

View coordinate system

When a program draws in a view, it draws relative to the view, not to the window or the screen. It doesn't matter where a window is onscreen, or where a view is within a window—the drawing will take place using the view's own coordinate system. Like the global and window coordinate systems, the view coordinate system is one of floating point coordinate pairs that increase in value from left to right and from top to bottom. The origin is located at the top left corner of the view. Consider this version of the `MyHelloView` member function `Draw()`:

```
void MyHelloView::Draw(BRect)
{
   MovePenTo(BPoint(10.0, 30.0));
   DrawString("Hello, My World!");
}
```

The arguments in the call to the `BView` member function `MovePenTo()` are local to the view's coordinate system. Regardless of where the view is located within its window, text drawing will start 10 units in from the left edge of the view and 30 units down from the top edge of the view.

Figure 4-9 highlights the fact that there are three separate coordinate systems at work in a program that displays a window that holds a view.

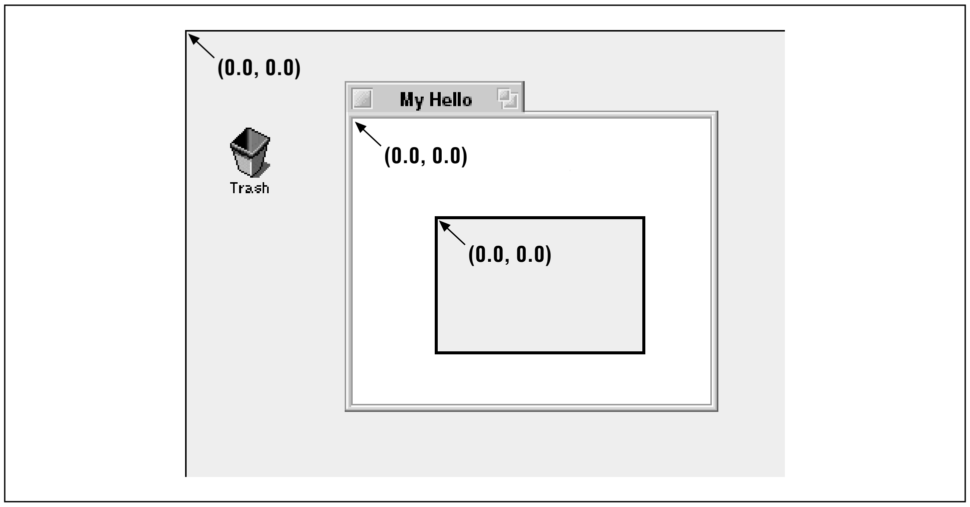


Figure 4-9. The screen, windows, and views have their own coordinate systems

Coordinate system example projects

To determine the size of a view in its own coordinate system (whether the view resides in a window or within another view), begin by invoking the `BView` member function `Bounds()`. In this chapter's OneView project, a call to this function has been added to the MyHelloView member function `Draw()`. One other `BView` member function call has been added too—a call to `StrokeRect()`. This routine draws a rectangle at the coordinates specified by the `BRect` argument passed to it:

```
void MyHelloView::Draw(BRect)
{
   BRect frame = Bounds();
   StrokeRect(frame);
```

```
        MovePenTo(BPoint(10.0, 30.0));
        DrawString("Hello, My World!");
    }
```

Since the rectangle returned by the `Bounds()` function call is relative to the view's own coordinate system, the `left` and `top` fields are always 0.0. The `right` and `bottom` fields reveal the view's width and height, respectively.

To find a view's boundaries relative to the window or view it resides in, call the `BView` member function `Frame()`. The rectangle returned by a call to `Frame()` has `left` and `top` fields that indicate the view's distance in and down from the window or view it resides in.

The OneView project creates a single `MyHelloView` view and adds it to a window. These steps take place in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    frame.OffsetTo(B_ORIGIN);
    fMyView = new MyHelloView(frame, "MyHelloView");

    AddChild(fMyView);
    Show();
}
```

Now that you know about the different coordinate systems, setting up the view rectangle might make more sense to you. In the above snippet, the `BRect` parameter `frame` holds the coordinates of the window. These coordinates directly define the screen placement of the window and indirectly define the size of the window (subtract `frame.left` from `frame.right` to get the window's width, and subtract `frame.top` from `frame.bottom` to get the window's height). Calling the `BRect` member function `OffsetTo()` with `B_ORIGIN` as the parameter shifts these coordinates so that each of the `frame.left` and `frame.top` fields has a value of 0.0. The overall size of the `frame` rectangle itself, however, doesn't change—it is still the size of the window. It just no longer reflects the screen positioning of the window. Next, the view that is to be added to the window is created. The view is to be positioned in the window using the window's coordinate system, so if the view is to fit snugly in the window, the view must have its top left corner at the window's origin. The `frame` rectangle that was initially used to define the placement and size of the window can now be used to define the placement and size of the view that is to fill the window.

When the `BWindow` member function `Show()` is invoked from the window constructor, the window is drawn to the screen and the view's `Draw()` function is automatically called to update the view. When that happens, the view is outlined—the `Draw()` function draws a line around the perimeter of the view. Figure 4-10 shows the result of creating a new window in the OneView project.

Because the window's one view is exactly the size of the content area of the window, the entire content area gets a line drawn around it.

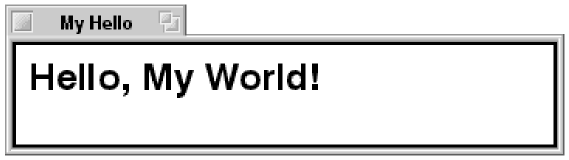


Figure 4-10. Drawing a rectangle around the OneView window's view

The OneSmallView project is very similar to the OneView project—both draw a frame around the one view that resides in the program's window. To demonstrate that a view doesn't have to occupy the entire content area of a window, the OneSmallView project sets up the view's boundary rectangle to be smaller than the window. This is done in the `MyHelloWindow` constructor:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
    frame.Set(100.0, 80.0, 250.0, 120.0);
    fMyView = new MyHelloView(frame, "MyHelloView");
    AddChild(fMyView);

    Show();
}
```

Here the line that offsets the window boundary rectangle (the `BRect` parameter `frame`) has been replaced by one that calls the `BRect` member function `Set()` to reset all the values of the `frame` rectangle. Figure 4-11 shows the resulting window. Note that a view is aware of its own boundaries, so that when you try to draw (or write) beyond a view edge, the result is truncated.

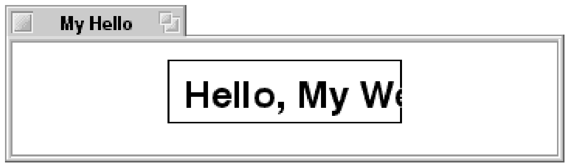


Figure 4-11. When a view is too small for the window content

Messaging

As discussed in Chapter 1, the Application Server communicates with (serves) an application by making the program aware of user actions. This communication is done in the form of *system messages* sent from the server to the application. Messages are received by a window and, often, passed on to a view in that window. The BeOS shoulders most of the responsibility of this communication between the Application Server, windows, and views. Your application (typically a view in a window in your application) is responsible for performing some specific action in response to a message.

System Messages

A system message is sent from the Application Server to a `BLooper` object. Both the `BApplication` and `BWindow` classes are derived from `BLooper`, so objects of these two classes (or objects of classes derived from these two classes) can receive messages. The Application server is responsible for directing a system message to the appropriate type of object.

The message loop of a program's `BWindow`-derived object receives messages that hold information about user actions. If the user typed a character, that character may need to be entered into a window. If the user clicked a mouse button, that click may have been made while the cursor was over a button in the window. The system message types of these two user actions are `B_KEY_DOWN` and `B_MOUSE_DOWN`. Such `BWindow`-directed system messages are referred to as *interface messages.*

The message loop of a program's `BApplication`-derived object receives messages that pertain to the application itself (as opposed to messages that pertain to a window or view, which are sent to a `BWindow`-derived object). If the user chooses the About menu item present in most programs, the program dispatches to the application object a message of type `B_ABOUT_REQUESTED`. Such `BApplication`-directed system messages are referred to as *application messages.*

See the Application Kit chapter of the Be Book for a description of all of the application messages, and the Interface Kit chapter for a description of all the interface messages.

System message dispatching

When a system message reaches a looper object (such as the application object or a window object), that object handles, or dispatches, the message by automatically invoking a virtual hook function. Such a function is declared `virtual` so that your own derived classes can override it in order to reimplement it to match your program's specific needs. In that sense, you're "hooking" your own code onto the system code.

Each system message has a corresponding hook function. For the three system messages mentioned above (B_ABOUT_REQUESTED, B_KEY_DOWN, and B_MOUSE_DOWN), those functions are AboutRequested(), KeyDown(), and MouseDown(). The application object itself handles a B_ABOUT_REQUESTED message by calling the BApplication member function AboutRequested(). A window object, on the other hand, passes a B_KEY_DOWN or B_MOUSE_DOWN message on to the particular view object to which the message pertains. This view object then invokes the BView member function KeyDown() or MouseDown() to handle the message.

Types of hook functions

For some system messages, the hook function defined by the Be class takes care of all the work suggested by the message. For instance, a click on a window's zoom button results in a B_ZOOM message being sent to the affected window. The receiving of this message automatically brings about the execution of the BWindow member function Zoom(). This hook function is fully implemented, meaning that you need to add no code to your project in order to support a click in a window's zoom button.

All hook functions are declared virtual, so your code can override even fully implemented ones. Unless your application needs to perform some nonstandard action in response to the message, though, there's no need to do so.

For other system messages, the hook function is implemented in such a way that the most common response to the message is handled. A program may override this type of hook function and reimplement it in such a way that the new version handles application-specific needs. This new application-defined version of the hook function may also call the original Be-defined BView version of the routine in order to incorporate the default actions of that BView version. An example of this type of hook function is ScreenChanged(), which is invoked in response to a B_SCREEN_CHANGED message. When the user changes the screen (perhaps by altering the monitor resolution), the application may need to make special adjustments to an open window. After doing that, the application-defined version of ScreenChanged() should invoke the BView-defined version of this routine so that the standard screen-changing code that's been supplied by Be can execute too.

Finally, for some system messages, the hook function implementation is left to the application. If an application is to respond to user actions that generate messages of types such as B_KEY_DOWN and B_MOUSE_DOWN, that application needs to override BView hook functions such as KeyDown() and MouseDown().

Interface messages

A system message directed at the application object is an application message, while a system message directed at a window object is an interface message. Responding to user actions is of great importance to a user-friendly application, so the remainder of this chapter is dedicated to illustrating how a project goes about doing this. In particular, I'll discuss the handling of two of the interface messages (`B_KEY_DOWN` and `B_MOUSE_DOWN`). Summarized below are several of the interface messages; refer to the Interface Kit chapter of the Be Book for a description of each of the 18 message types.

`B_KEY_DOWN`
: Goes to the active window in response to the user pressing a character key. The recipient window invokes the `BView` hook function `KeyDown()` of the affected view. The affected view is typically one that accepts text entry, such as a view of the yet-to-be-discussed `BTextControl` or `BTextView` classes. An example of handling a `B_KEY_DOWN` message is presented later in this chapter.

`B_KEY_UP`
: Is sent to the active window when the user releases a pressed character key. The recipient window invokes the `BView` hook function `KeyUp()` of the affected view. Typically, a program responds to a `B_KEY_DOWN` message and ignores the `B_KEY_UP` message that follows. In other words, the program doesn't override the `BView` hook function `KeyUp()`.

`B_MOUSE_DOWN`
: Is sent to the window over which the cursor was located at the time of the mouse button click. The window that receives the message calls the `BView` hook function `MouseDown()` of the view the cursor was over at the time of the mouse button click.

`B_MOUSE_UP`
: Reaches the window that was affected by a `B_MOUSE_DOWN` message when the user releases a pressed mouse button. The `MouseDown()` hook function that executes in response to a `B_MOUSE_DOWN` message often sufficiently handles a mouse button click, so a `B_MOUSE_UP` message is often ignored by a program. That is, the program doesn't override the `BView` hook function `MouseUp()`.

`B_MOUSE_MOVED`
: Is sent to a window when the user moves the cursor over the window. As the user drags the mouse, repeated `B_MOUSE_MOVED` messages are issued by the Application Server. As the cursor moves over one window to another, the window to which the messages are sent changes. When the mouse is moved over the desktop rather than a window, a `B_MOUSE_MOVED` message is sent to the Desktop window of the Tracker.

Mouse Clicks and Views

When a window receives a B_MOUSE_DOWN message from the Application Server, the window object (without help from you) determines which of its views should respond. It is that view's MouseDown() hook function that is then invoked.

The ViewsMouseMessages project includes a MouseDown() routine with the MyHelloView class in order to make the program "mouse-click aware." The ViewsMouseMessages program displays a single window that holds two framed MyHelloView views. Clicking the mouse while the cursor is over either view results in the playing of the system beep.

The mechanism for responding to a mouse click has already been present in every example project in this book, so there's very little new code in the ViewsMouse-Messages project. The ViewsMouseMessages program, and every other program you've seen in this book, works as follows: when the user clicks the mouse button while the cursor is over a window, the Application Server sends a B_MOUSE_DOWN message to the affected window, causing it to invoke the affected view's MouseDown() hook function. The MyHelloView class is derived from the BView class, and the BView class defines its version of MouseDown() as an empty function. So unless the MyHelloView class overrides MouseDown(), it inherits this "do-nothing" routine. In all previous examples, a mouse button click while the cursor was over a view resulted in the execution of this empty routine—so effectively the mouse button click was ignored. The ViewsMouseMessages project overrides MouseDown() so that a mouse button click with the cursor over a view now results in something happening. Here's the latest version of the MyHelloView class definition, with the addition of the MouseDown() declaration:

```
class MyHelloView : public BView {

   public:
                    MyHelloView(BRect frame, char *name);
      virtual void  AttachedToWindow();
      virtual void  Draw(BRect updateRect);
      virtual void  MouseDown(BPoint point);
};
```

The one MouseDown() parameter is a BPoint that is passed to the routine by the Application Server. This point parameter holds the location of the cursor at the time the mouse button was clicked. The values of the point are in the view's coordinate system. For example, if the cursor was over the very top left corner of the view at the time of the mouse click, the point's coordinates would be close to (0.0, 0.0). In other words, both point.x and point.y would have a value close to 0.0.

To verify that the B_MOUSE_DOWN message has worked its way to the new version of MouseDown(), the implementation of MouseDown() sounds the system beep:

```
void MyHelloView::MouseDown(BPoint point)
{
   beep();
}
```

Recall that `beep()` is a global function that, like the `snooze()` routine covered earlier in this chapter, can be called from any point in your project's source code.

Key Presses and Views

In response to a `B_MOUSE_DOWN` message, a window object invokes the `MouseDown()` function of the affected view. For the window object, determining which view is involved is simple—it chooses whichever view object is under the cursor at the time of the mouse button click. This same test isn't made by the window in response to a `B_KEY_DOWN` message. That's because the location of the cursor when a key is pressed is generally insignificant. The scheme used to determine which view's `KeyDown()` hook function to invoke involves a focus view.

Focus view

A program can make any view the focus view by invoking that view's `MakeFocus()` function. For a view that accepts typed input (such as `BTextControl` or a `BTextView` view), the call is made implicitly when the user clicks in the view to activate the insertion bar. Any view, however, can be made the focus view by explicitly calling `MakeFocus()`. Here a click of the mouse button while the cursor is over a view of type `MyHelloView` makes that view the focus view:

```
void MyHelloView::MouseDown(BPoint point)
{
   MakeFocus();
}
```

Now, when a key is pressed, the `KeyDown()` hook function of the last clicked-on view of type `MyHelloView` will automatically execute.

Because a `MyHelloView` view doesn't accept keyboard input, there is no obvious reason to make a view of this type the focus view. We haven't worked with many view types, so the above example must suffice here. If you're more comfortable having a reason for making a `MyHelloView` accept keyboard input, consider this rather contrived scenario. You want the user to click on a view of type `MyHelloView` to make it active. Then you want the user to type any character and have the view echo that character back—perhaps in a large, bold font. Including the above `MouseDown()` routine in a project suffices to make the view the focus view when clicked on. Now a `MyHelloView` `KeyDown()` routine can be written to examine the typed character, clear the view, and draw the typed character.

KeyDown() example project

The ViewsKeyMessages project adds to the ViewsMouseMessages project to create a program that responds to both mouse button clicks and key presses. Once again, a mouse button click while the cursor is over a view results in the sounding of the system beep. Additionally, ViewsKeyMessages beeps twice if the Return key is pressed and three times if the 0 (zero) key is pressed.

To allow a `MyHelloView` view to respond to a press of a key, the `BView` hook function `KeyDown()` needs to be overridden:

```
class MyHelloView : public BView {

   public:
                     MyHelloView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
      virtual void   MouseDown(BPoint point);
      virtual void   KeyDown(const char *bytes, int32 numBytes);
};
```

The first `KeyDown()` parameter is an array that encodes the typed character along with any modifier keys (such as the Shift key) that were down at the time of the key press. The second parameter tells how many bytes are in the array that is the first parameter. As with all hook functions, the values of these parameters are filled in by the system and are available in your implementation of the hook function should they be of use.

The `KeyDown()` routine responds to two key presses: the Return key and the 0 (zero) key. Pressing the Return key plays the system beep sound twice, while pressing the 0 key plays the sound three times:

```
void MyHelloView::KeyDown(const char *bytes, int32 numBytes)
{
   bigtime_t  microseconds = 1000000;

   switch ( *bytes ) {

      case B_RETURN:
         beep();
         snooze(microseconds);
         beep();
         break;

      case '0':
         beep();
         snooze(microseconds);
         beep();
         snooze(microseconds);
         beep();
         break;
      default:
         break;
   }
}
```

There are a number of Be-defined constants you can test bytes against; B_RETURN is one of them. The others are: B_BACKSPACE, B_LEFT_ARROW, B_INSERT, B_ENTER, B_RIGHT_ARROW, B_DELETE, B_UP_ARROW, B_HOME, B_SPACE, B_DOWN_ARROW, B_END, B_TAB, B_PAGE_UP, B_ESCAPE, B_FUNCTION_KEY, and B_PAGE_DOWN. For a key representing an alphanumeric character, just place the character between single quotes, as shown above for the 0 (zero) character.

Notice that calls to the global function snooze() appear between calls to the global function beep(). The beep() routine executes in its own thread, which means as soon as the function starts, control returns to the caller. If successive, uninterrupted calls are made to beep(), the multiple playing of the system beep will seem like a single sound.

Only the focus view responds to a key press, so the ViewsKeyMessages program needs to make one of its two views the focus view. I've elected to do this in the MyHelloView MouseDown() routine. When the user clicks on a view, that view becomes the focus view:

```
void MyHelloView::MouseDown(BPoint point)
{
   beep();

   MakeFocus();
}
```

When the user then presses a key, the resulting B_KEY_DOWN message is directed at that view. Since the views are derived from the BView class, rather than a class that accepts keyboard input, a typed character won't appear in the view. But the view's KeyDown() routine will still be called.

5

In this chapter:
- *Colors*
- *Patterns*
- *The Drawing Pen*
- *Shapes*

Drawing

When a Be application draws, it always draws in a view. That's why the chapter that deals with views precedes this chapter. In Chapter 4, *Windows, Views, and Messages*, you saw that a `BView`-derived class overrides its inherited hook member function `Draw()` so that it can define exactly what view objects should draw in when they get updated. The example projects in this chapter contain classes and member functions that remain unchanged, or changed very little, from previous example projects. What will be different is the content of the `Draw()` function. The code that demonstrates the concepts of each drawing topic can usually be added to the `Draw()` routine.

In Be programming, the colors and patterns that fill a shape aren't defined explicitly for that shape. Instead, traits of the graphics environment of the view that receives the drawing are first altered. In other words, many drawing characteristics, such as color and font, are defined at the view level, so all subsequent drawing can use the view settings. In this chapter, you'll see how to define a color, then set a view to draw in that color. You'll see how the same is done for patterns—whether using Be-defined patterns or your own application-defined ones.

After you learn how to manipulate the graphic characteristics of a view, it's on to the drawing of specific shapes. The point (represented by `BPoint` objects) is used on its own, to define the end points of a line, and to define the vertices of more sophisticated shapes (such as triangles or polygons). The rectangle (represented by `BRect` objects) is used on its own and as the basis of more sophisticated shapes. These shapes include round rectangles, ellipses, and regions. Round rectangles and ellipses are closely related to `BRect` objects, and aren't defined by their own classes. Polygons and regions are more sophisticated shapes that make use of points and rectangles, but are represented by their own class types (`BPolygon` and `BRegion`). In this chapter, you'll see how to outline and fill each of these different shapes. Finally, I show how to combine any type and number of these various shapes into a picture represented by a `BPicture` object.

Colors

The BeOS is capable of defining colors using any of a number of *color spaces*. A color space is a scheme, or system, for representing colors as numbers. There are several color space Be-defined constants, each containing a number that reflects the number of bits used to represent a single color in a single pixel. For instance, the `B_COLOR_8_BIT` color space devotes 8 bits to defining the color of a single pixel. The more memory devoted to defining the color of a single pixel, the more possible colors a pixel can display.

`B_GRAY1`
: Each pixel in a drawing is either black (bit is on, or 1) or white (bit is off, or 0).

`B_GRAY8`
: Each pixel in a drawing can be one of 256 shades of gray—from black (bit is set to a value of 255) to white (bit is set to a value of 0).

`B_CMAP8`
: Each pixel in a drawing can be one of 256 colors. A pixel value in the range of 0 to 255 is used as an index into a color map. This system color map is identical for all applications. That means that when two programs use the same value to color a pixel, the same color will be displayed.

`B_RGB15`
: Each pixel in a drawing is created from three separate color components: red, green, and blue. Five out of a total of sixteen bits are devoted to defining each color component. The sixteenth bit is ignored.

`B_RGB32`
: Like the `B_RGB15` color space, each pixel in a drawing is created from three separate color components: red, green, and blue. In `B_RGB32` space, however, eight bits are devoted to defining each color component. The remaining eight bits are ignored.

`B_RGBA32`
: Like the `B_RGB32` color space, each pixel in a drawing is created from three separate color components: red, green, and blue. Like `B_RGB`, eight bits are used to define each of the three color components. In `B_RGBA32` space, however, the remaining eight bits aren't ignored—they're devoted to defining an alpha byte, which is used to specify a transparency level for a color.

RGB Color System

As listed above, the BeOS supports a number of color spaces. The RGB color space is popular because it provides over sixteen million unique colors (the number of combinations using values in the range of 0 to 255 for each of the three color components), and because it is a color system with which many programmers and end users are familiar with (it's common to several operating systems).

The BeOS defines `rgb_color` as a `struct` with four fields:

```
typedef struct {
   uint8  red;
   uint8  green;
   uint8  blue;
   uint8  alpha;
} rgb_color
```

A variable of type `rgb_color` can be initialized at the time of declaration. The order of the supplied values corresponds to the ordering of the `struct` definition. The following declares an `rgb_color` variable named `redColor` and assigns the `red` and `alpha` fields a value of 255 and the other two fields a value of 0:

```
rgb_color  redColor = {255, 0, 0, 255};
```

To add a hint of blue to the color defined by `redColor`, the third value could be changed from 0 to, say, 50. Because the `alpha` component of a color isn't supported at the time of this writing, the last value should be 255. Once supported, an `alpha` value of 255 will represent a color that is completely opaque; an object of that color will completely cover anything underneath it. An `alpha` field value of 0 will result in a color that is completely transparent—an effect you probably don't want. An `rgb_color` variable can be set to represent a new color at any time by specifying new values for some or all of the three color components. Here an `rgb_color` variable named `blueColor` is first declared, then assigned values:

```
rgb_color  blueColor;

blueColor.red   = 0;
blueColor.green = 0;
blueColor.blue  = 255;
blueColor.alpha = 255;
```

While choosing values for the red, green, and blue components of a color is easy if you want a primary color, the process isn't completely intuitive for other colors. Quickly now, what values should you use to generate chartreuse? To experiment with colors and their RGB components, run the ColorControl program that's discussed a little later in this chapter. By the way, to create the pale, yellowish green color that's chartreuse, try values of about 200, 230, and 100 for the red, green, and blue components, respectively.

High and Low Colors

Like all graphics objects, an `rgb_color` variable doesn't display any color in a window on its own—it only sets up a color for later use. A view always keeps track of two colors, dubbed the *high* and *low* colors. When you draw in the view, you specify whether the current high color, the current low color, or a mix of the two colors should be used.

Views and default colors

When a new view comes into existence, it sets a number of drawing characteristics to default values. Included among these are:

- A high color of black
- A low color of white
- A background color of white

Additionally, when a `BView` drawing function is invoked, by default it uses the view's high color for the drawing. Together, these facts tell you that unless you explicitly specify otherwise, drawing will be in black on a white background.

Setting the high and low colors

The `BView` member functions `SetHighColor()` and `SetLowColor()` alter the current high and low colors of a view. Pass `SetHighColor()` an `rgb_color` and that color becomes the new high color—and remains as such until the next call to `SetHighColor()`. The `SetLowColor()` routine works the same way. This next snippet sets a view's high color to red and its low color to blue:

```
rgb_color  redColor  = {255, 0, 0, 255};
rgb_color  blueColor = {0, 0, 255, 255};

SetHighColor(redColor);
SetLowColor(blueColor);
```

Drawing with the high and low colors

Passing an `rgb_color` structure to `SetHighColor()` or `SetLowColor()` establishes that color as the one to be used by a view when drawing. Now let's see how the high color is used to draw in color:

```
rgb_color  redColor  = {255, 0, 0, 255};
BRect      aRect(10, 10, 110, 110);

SetHighColor(redColor);

FillRect(aRect, B_SOLID_HIGH);
```

The previous snippet declares redColor to be a variable of type rgb_color and defines that variable to represent red. The snippet also declares a BRect variable named aRect, and sets that variable to represent a rectangle with a width and height of 100 pixels. The call to SetHighColor() sets the high color to red. Finally, a call to the BView member function FillRect() fills the rectangle aRect with the current high color (as specified by the Be-defined constant B_SOLID_HIGH)—the color red.

Shape-drawing routines such as FillRect() are described in detail later in this chapter. For now, a brief introduction will suffice. A shape is typically drawn by first creating a shape object to define the shape, then invoking a BView member function to draw it. That's what the previous snippet does: it creates a rectangle shape based on a BRect object, then calls the BView member function FillRect() to draw the rectangle.

One of the parameters to a BView shape-drawing routine is a pattern. As you'll see ahead in the "Patterns" section of this chapter, a pattern is an 8-pixel-by-8-pixel template that defines some combination of the current high color and low color. This small template can be repeatedly "stamped" into an area of any size to fill that area with the pattern. Patterns are everywhere these days: desktop backgrounds, web page backgrounds, and so on. You can create your own patterns, or use one of the three Be-defined patterns. Each of the Be-defined patterns is represented by a constant:

- B_SOLID_HIGH is a solid fill of the current high color.
- B_SOLID_LOW is a solid fill of the current low color.
- B_MIXED_COLORS is a checkerboard pattern of alternating current high color and low color pixels (providing a dithered effect—what looks like a single color blended from the two colors).

A view's default high color is black. So before a view calls SetHighColor(), the use of B_SOLID_HIGH results in a solid black pattern being used. The above snippet invokes SetHighColor() to set the current high color to red, so subsequent uses of B_SOLID_HIGH for this one view result in a solid red pattern being used.

Determining the current high and low colors

You can find out the current high or low color for a view at any time by invoking the BView member functions HighColor() or LowColor(). Each routine returns a value of type rgb_color. This snippet demonstrates the calls:

```
rgb_color  currentHighColor;
rgb_color  currentLowColor;

currentHighColor = HighColor();
currentLowColor  = LowColor();
```

The default high color is black, so if you invoke `HighColor()` before using `SetHighColor()`, an `rgb_color` with red, green, and blue field values of 0 will be returned to the program. The default low color is white, so a call to `LowColor()` before a call to `SetLowColor()` will result in the return of an `rgb_color` with red, green, and blue field values of 255. Because the `alpha` field of the high and low colors is ignored at the time of this writing, the `alpha` field will be 255 in both cases.

RGB, low, and high color example project

The RGBColor project is used to build a program that displays a window like the one shown in Figure 5-1. Given the nature of this topic, you can well imagine that the window isn't just as it appears in this figure. Instead a shade of each being a shade of gray, the three rectangles in the window are, from left to right, red, blue, and a red-blue checkerboard. Because of the high resolution typical of today's monitors, the contents of the rightmost rectangle dither to a solid purple rather than appearing to the eye as alternating red and blue pixels.

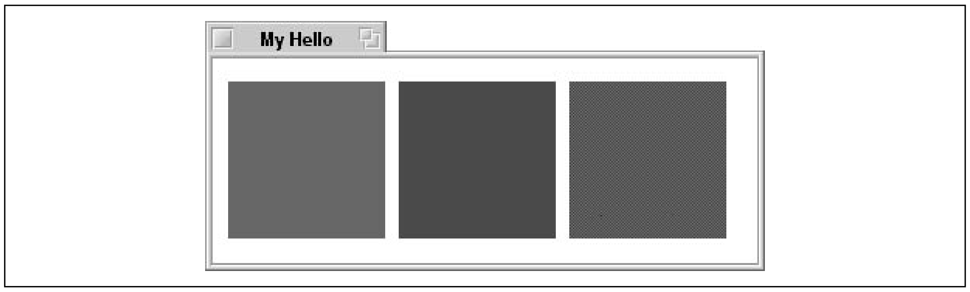


Figure 5-1. The window that results from running the RGBColor program

Chapter 4 included the TwoViewClasses project—a project that introduced a new view class named `MyDrawView`. That class definition was almost identical to the original `MyHelloView`. This chapter's RGBColor project and all remaining projects in this chapter display a single window that holds a single `MyDrawView` view, and no `MyHelloView`. So the *MyHelloView.cpp* file is omitted from these projects, and the data member meant to keep track of a `MyHelloView` in the `MyHelloWindow` class (reproduced below) is also omitted:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual  bool     QuitRequested();

   private:
      MyDrawView        *fMyDrawView;
};
```

Creating a new `MyHelloWindow` object now entails creating just a single `MyDrawView` view that fills the window, then attaching the view to the window:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyDrawView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyDrawView);

   Show();
}
```

Drawing in a view takes place automatically when the system calls the view's `Draw()` routine. That function is the code I play with in order to try out drawing ideas. Here's how the RGBColor project implements the `MyDrawView` version of `Draw()`:

```
void MyDrawView::Draw(BRect)
{
   BRect      aRect;
   rgb_color  redColor = {255, 0, 0, 255};
   rgb_color  blueColor;

   blueColor.red = 0;
   blueColor.green = 0;
   blueColor.blue = 255;
   blueColor.alpha = 255;

   SetHighColor(redColor);
   SetLowColor(blueColor);

   aRect.Set(10, 10, 110, 110);
   FillRect(aRect, B_SOLID_HIGH);

   aRect.Set(120, 10, 220, 110);
   FillRect(aRect, B_SOLID_LOW);

   aRect.Set(230, 10, 330, 110);
   FillRect(aRect, B_MIXED_COLORS);
}
```

The previous routine demonstrates two methods of assigning an `rgb_color` variable a color value. After that, the `SetHighColor()` and `SetLowColor()` functions set the `MyDrawView` high color and low color to red and blue, respectively. Then in turn each of the three rectangles is set up, then filled.

The View Color (Background)

To color a shape, the program often refers to the `B_SOLID_HIGH` constant. As you just saw in the previous example project, the `B_SOLID_LOW` and `B_MIXED_COLORS` constants can also be used to include the view's current low color in the drawing. By now it should be apparent that neither the high nor low color implicitly has anything to do with a view's background color.

Setting a view's background color

By default, a new view has a background color of white. This background color can be set to any RGB color by invoking the `BView` member function `SetViewColor()`. Here a view's background color is being set to purple:

```
rgb_color  purpleColor = {255, 0, 255, 255};
SetViewColor(purpleColor);
```

Calling `SetViewColor()` changes the background color of a view without affecting either the high color or the low color. Consider a view with a current high color of blue, a current low color of yellow, and a background color set to purple. Calling a `BView` fill routine with a pattern argument of `B_SOLID_HIGH` draws a blue shape. An argument of `B_SOLID_LOW` draws a yellow shape. Finally, an argument of `B_MIXED_COLORS` draws a green shape. All shapes are drawn against the view's purple background.

View color example project

The ViewColor program displays a window that looks identical to that displayed by the RGBColor example, except for one feature. Both programs display a window with a red, blue, and purple rectangle in it, but the ViewColor window background is pink rather than white. This trick is performed by adding just a few lines of code to the `AttachedToWindow()` routine defined in the *MyDrawView.cpp* file in the RGBColor project. Here an `rgb_color` variable is set up to define the color pink, and that variable is used as the argument to a call to `SetViewColor()`. Here's the new version of the `MyDrawView` member function `AttachedToWindow()`:

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);

   rgb_color  pinkColor = {255, 160, 220, 255};
   SetViewColor(pinkColor);
}
```

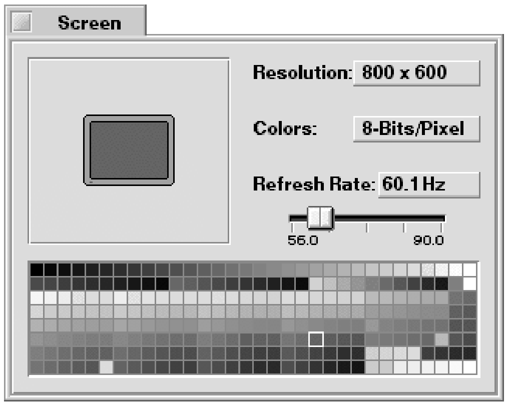
Color Control View

The RGB components of any given color won't be known by a program's user. There are exceptions, of course—graphics artists involved in electronic media or electronic publications may have a working knowledge of how RGB values correspond to colors. Those exceptions aside, if your program allows users to select their own colors, your program should provide a very user-friendly means for them to accomplish this task. The `BColorControl` class does just that.

Color levels and the BColorControl object

The `BColorControl` class is derived from the `BControl` class, which itself is derived from the `BView` class. So a `BColorControl` object is a type of view. Your program creates a `BColorControl` object in order to allow a user to select an RGB color without the user knowing anything about the RGB color system or RGB values.

What the `BColorControl` object displays to the user depends on the number of colors the user's monitor is currently displaying. The user can set that parameter by choosing Screen from the preferences menu in the Deskbar. Coincidentally, the Screen preferences window (which has been revamped and turned into the Background preferences application) itself holds a `BColorControl` object. So you can see how the monitor's color depth is set and take a look at a `BColorControl` object by selecting the Screen preferences or by simply looking at Figure 5-2.



Resolution: 800 x 600

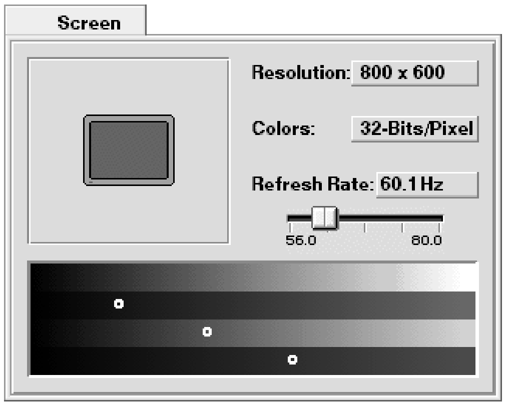
Colors: 8-Bits/Pixel

Refresh Rate: 60.1 Hz

Figure 5-2. The Screen preferences program set to display 8-bit color

The Screen preferences window holds a number of objects representing `BView`-derived classes. Among them is a pop-up menu titled Colors. In Figure 5-2, you see that I have my monitor set to devote 8 bits of graphics memory to each pixel, so my monitor can display up to 256 colors. The Screen preferences window lets me choose one of the 256 system colors to be used as my desktop color. This is done by clicking on one of the 256 small colored squares. This matrix, or block of squares, is a `BColorControl` object.

Choosing 32-Bits/Pixel from the Colors pop-up menu in the Screen preferences window sets a monitor to display any of millions of colors. As shown in Figure 5-3, doing so also changes the look of the `BColorControl` object. Now a color is selected by clicking on the red, green, and blue ramps, or bands, of color along the bottom of the Screen preferences window. Unbeknownst to the user, doing this sets up an RGB color. The Screen preferences program combines the user's three color choices and uses the resulting RGB value as the desktop color.



Resolution: 800 x 600
Colors: 32-Bits/Pixel
Refresh Rate: 60.1 Hz

Figure 5-3. The Screen preferences program set to display 32-bit color

Creating a BColorControl object

The Screen preferences window serves as a good example of how a `BColorControl` object can help the user. To use the class in your own program, declare a `BColorControl` object and the variables that will be used as parameters to the `BColorControl` constructor. Then create the new object using `new` and the `BColorControl` constructor:

```
BColorControl         *aColorControl;
BPoint                leftTop(20.0, 50.0);
color_control_layout  matrix = B_CELLS_16x16;
long                  cellSide = 16;

aColorControl = new BColorControl(leftTop, matrix, cellSide, "ColorControl");

AddChild(aColorControl);
```

The first `BColorControl` constructor parameter, `leftTop`, indicates where the top left corner of the color control should appear. The color control will be placed in a view (by calling the `AddChild()` function of the host view, as shown above), so you should set `BPoint`'s coordinates relative to the view's borders.

The second parameter, `matrix`, is of the Be-defined datatype `color_control_layout`. When the user has the monitor set to 8 bits per pixel, the 256 system colors are displayed in a matrix. This parameter specifies how these squares should be arranged. Use one of five Be-defined constants here: `B_CELLS_4x64`, `B_CELLS_8x32`, `B_CELLS_16x16`, `B_CELLS_32x8`, or `B_CELLS_64x4`. The two numbers in each constant name represent the number of columns and rows, respectively, that the colored squares are placed in. For example, `B_CELLS_32x8` displays the 256 colors in eight rows with 32 colored squares in each row.

The third `BColorControl` constructor parameter, `cellSide`, determines the pixel size of each colored square in the matrix. A value of 10, for instance, results in 256 squares that are each 10 pixels by 10 pixels in size.

The fourth parameter provides a name for the `BColorControl` object. Like any view, a `BColorControl` object has a name that can be used to find the view. The name can be supplied using a string (as shown in the previous snippet) or by passing in a constant variable that was defined as a `const char *` (as in `const char *name = "ColorControl";`).

Note that the overall size of the `BColorControl` object isn't directly specified in the constructor. The size is calculated by the constructor, and depends on the values supplied in the second and third parameters. The matrix parameter specifies the shape of the block of colors, while the `cellSide` value indirectly determines the overall size. A matrix with 8 rows of cells that are each 10 pixels high will have a height of 80 pixels, for instance.

I've discussed the `BColorControl` constructor parameters as if they will be used with 8-bit pixels. For the values used in the previous example, the resulting color control looks like the one displayed in the window in Figure 5-4. If the user instead has the monitor set to 32 bits for each pixel, the same arguments are used in the display of four bands, three of which the user clicks on in order to create a single color. The top gray band represents the `alpha`, or color transparency level, component. As of this writing, the `alpha` component is unimplemented, but it should be implemented by the time you read this. Instead of creating a matrix of color squares, the arguments are now used to determine the shape and overall size the four bands occupy. Figure 5-5 shows the color control that results from executing the previous snippet when the monitor is set to 32 bits per pixel.

The user can set the monitor to the desired bits-per-pixel level, so your program can't count on being used for a matrix or bands. In Figure 5-5, you see that the color bands are very broad—that's the result of specifying a 16-by-16 matrix (`B_CELLS_16x16`). To display longer, narrower color bands, choose a different Be-defined constant for the `BColorControl` constructor `matrix` argument (such as `B_CELLS_64x4`). Regardless of the values you choose for the `matrix` and `cellSize` parameters, test the resulting color control under both monitor settings to verify that the displayed control fits well in the window that displays it.

Using a color control

When a window displays a color control, the user selects a color by clicking on its cell (if the user's monitor is set to 8 bits per pixel) or by clicking on a color intensity in each of the three color component bands (if the user's monitor is set to 32 bits per pixel). In either case, the `BColorControl` object always keeps track of the currently selected color. Your program can obtain this color at any time via a call to the `BColorControl` member function `ValueAsColor()`. Obviously enough, a call to this routine returns the value of the color control object in the form of an RGB color. In this next snippet, the user's current color choice is returned and stored in an `rgb_color` variable named `userColorChoice`:

```
rgb_color  userColorChoice;

userColorChoice = aColorControl->ValueAsColor();
```

What your program does with the returned color is application-specific. Just bear in mind that this value can be used the same way any `rgb_color` is used. You know about the `SetHighColor()` routine that sets a view's high color, and you've seen how to fill a rectangle with the current high color by calling the `BView` member function `FillRect()`, so an example that carries on with the previous snippet's `userColorChoice` RGB color will be easily understandable:

```
BRect  aRect(40.0, 50.0, 400.0, 55.0);

SetHighColor(userColorChoice);

FillRect(aRect, B_SOLID_HIGH);
```

The previous snippet creates a rectangle in the shape of a long horizontal bar, sets the high color to whatever color the user has the color control currently set at, then fills the rectangle with that color.

ColorControl example project

If you set your monitor to use 8 bits per pixel (using the Screen preferences utility), running this chapter's ColorControl example program results in a window like the one shown in Figure 5-4. If you instead have your monitor set to use 32 bits per pixel, running the same program displays a window like that shown in Figure 5-5.

Regardless of your monitor's pixel setting, the ColorControl program displays three text boxes to the right of the color matrix or color bands. These text boxes are displayed automatically by the `BColorControl` object, and the area they occupy constitutes a part of the total area occupied by the control. If you click on a color cell or a color band, the numbers in these boxes will change to reflect the appropriate RGB values for the color you've selected. If you click in a text box (or use the Tab key to move to a text box) and type in a value between 0 and 255, the

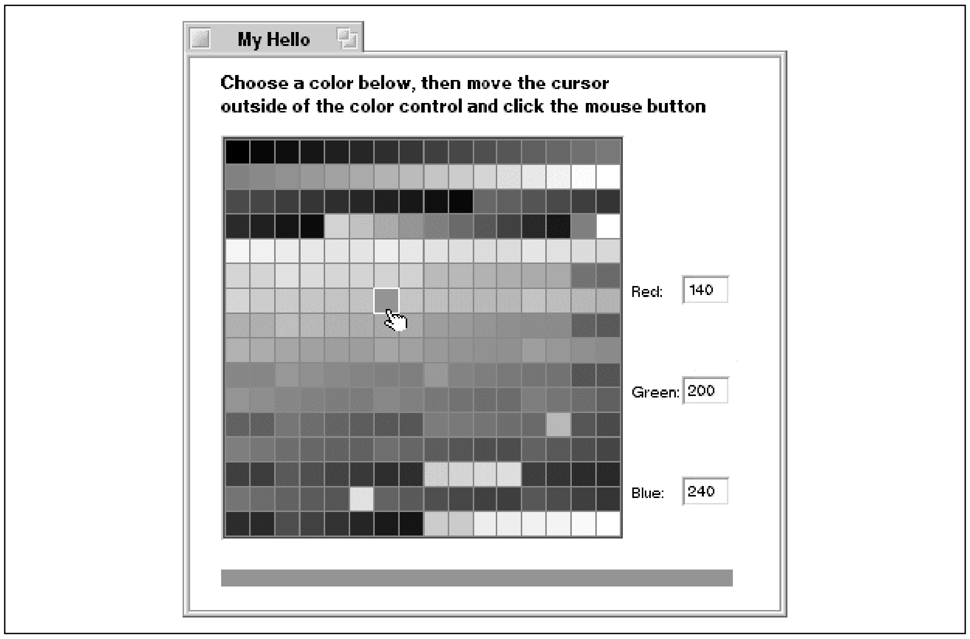

My Hello
Choose a color below, then move the cursor
outside of the color control and click the mouse button
Red: 140
Green: 200
Blue: 240

Figure 5-4. The ColorControl program's window that results from running at 8-bit color

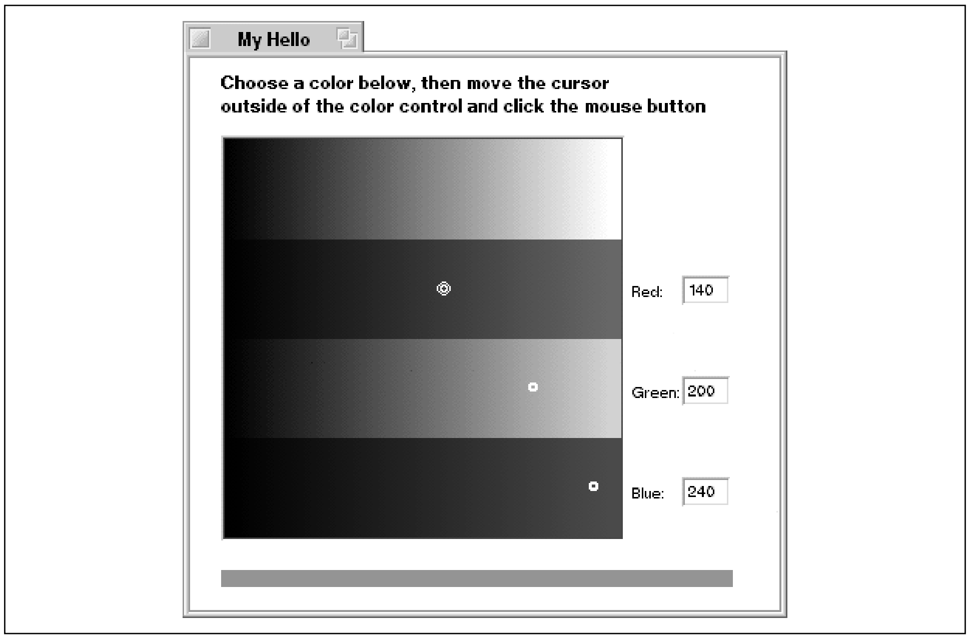

My Hello
Choose a color below, then move the cursor
outside of the color control and click the mouse button
Red: 140
Green: 200
Blue: 240

Figure 5-5. The ColorControl program's window that results from running at 32-bit color

color display will update itself to display the color that best matches the value you've entered. These actions are all automatic, and require no coding effort on your part. The reason this handy feature works is that the `BControl` class overrides the `BView` class member function `KeyDown()`, and in turn the `BColorControl` class overrides the `BControl` version of `KeyDown()`. The `BColorControl` version of the routine sees to it that the text box values reflect the displayed color.

If you move the cursor out of the color control area (keep in mind that this area includes the text boxes), then click the mouse button, a long, narrow bar is drawn along the bottom of the window—as shown in Figures 5-4 and 5-5. The color of this bar will match whatever color you have currently selected in the color control. The selection of this color and the drawing of the bar are handled by the `MyDrawView` version of the `MouseDown()` routine. Besides overriding the `Bview` hook function `MouseDown()`, this project's version of the `MyDrawView` class adds a `BColorControl` data member. The color control data member will be used to keep track of the control. Here's how the ColorControl project declares the `MyDrawView` class:

```
class MyDrawView : public BView {

   public:
                        MyDrawView(BRect frame, char *name);
      virtual  void     AttachedToWindow();
      virtual  void     Draw(BRect updateRect);
      virtual  void     MouseDown(BPoint point);

   private:
      BColorControl     *fColorControl;
};
```

The `MyHelloApplication` class and `MyHelloWindow` class are almost identical to versions found in previous examples. The `MyHelloApplication` constructor defines the size of a window and creates a single `MyHelloWindow`, and the `MyHelloWindow` defines a single `MyDrawView` whose size is the same as the window it resides in.

The `MyDrawView` constructor, which in other projects has been empty, sets up and creates a color control. The control object is added to the newly created `MyDrawView` object as a child:

```
MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BPoint               leftTop(20.0, 50.0);
   color_control_layout  matrix = B_CELLS_16x16;
   long                 cellSide = 16;

   fColorControl = new BColorControl(leftTop, matrix, cellSide,
                                 "ColorControl");
   AddChild(fColorControl);
}
```

In the `MyDrawView` constructor, you see that the control will have its top left corner start 20 pixels from the left and 50 pixels from the top of the `MyDrawView` view that the control appears in. Starting down 50 pixels from the top of the view leaves room for the two lines of instructional text that are displayed in the window (refer back to Figure 5-4 or 5-5). Those lines are drawn each time the system has to update the view they appear in:

```
void MyDrawView::Draw(BRect)
{
   MovePenTo(BPoint(20.0, 20.0));
   DrawString("Choose a color below, then move the cursor");
   MovePenTo(BPoint(20.0, 35.0));
   DrawString("outside of the color control and click the mouse button");
}
```

When the user clicks in the `MyDrawView` view, the `MouseDown()` routine that the `MyDrawView` class overrides is automatically invoked:

```
void MyDrawView::MouseDown(BPoint point)
{
   BRect      aRect(20.0, 330.0, 350.0, 340.0);
   rgb_color  userColorChoice;

   userColorChoice = fColorControl->ValueAsColor();

   SetHighColor(userColorChoice);

   FillRect(aRect, B_SOLID_HIGH);
}
```

`MouseDown()` creates the long, thin rectangle that appears along the bottom of the view when the user clicks the mouse button. Before this function draws the rectangle with a call to `FillRect()`, a `ValueAsColor()` call obtains the color currently selected in the view's color control. A call to `SetHighColor()` makes the user-selected color the one used in function calls that include `B_SOLID_HIGH` as a parameter.

Improving the ColorControl example project

For brevity, the ColorControl example sets the high color and fills in the colored rectangle in the `MouseDown()` routine. Typically, drawing takes place only in a view's `Draw()` function. One way to accomplish that would be to move the code currently in `MouseDown()` to `Draw()`:

```
void MyDrawView::Draw(BRect)
{
   BRect      aRect(20.0, 330.0, 350.0, 340.0);
```

```
    rgb_color  userColorChoice;

    MovePenTo(BPoint(20.0, 20.0));
    DrawString("Choose a color below, then move the cursor");
    MovePenTo(BPoint(20.0, 35.0));
    DrawString("outside of the color control and click the mouse button");

    userColorChoice = fColorControl->ValueAsColor();
    SetHighColor(userColorChoice);
    FillRect(aRect, B_SOLID_HIGH);
}
```

The body of `MouseDown()` could then consist of a single line of code: a call to the `BView` function `Invalidate()`. Then, when the user clicks the mouse in the `MyDrawView` view, `MouseDown()` makes the system aware of the fact that the view needs updating, and the system invokes `Draw()`:

```
void MyDrawView::MouseDown(BPoint point)
{
    Invalidate();
}
```

One further improvement to the ColorControl example program would be to preserve the current state of the view before changing its high color. As implemented (here and in the previous section), the text the program draws is drawn in black the first time the `Draw()` function executes. Subsequent calls will update any previously obscured text (as in the case when an overlapping window is moved off the ColorControl program's window) in whatever color was selected by the user. That is, the program's call to `SetHighColor()` affects not only the long, narrow color rectangle at the bottom of the program's window, but also text drawn with calls to `DrawString()`. To remedy this, preserve the state of the high color by invoking the `BView` function `HighColor()` to get the current high color before changing it. After calling `SetHighColor()` and `FillRect()`, use the `rgb_color` value returned by `HighColor()` to reset the high color to its state prior to the use of the user-selected color. Here's how `Draw()` now looks:

```
void MyDrawView::Draw(BRect)
{
    BRect      aRect(20.0, 330.0, 350.0, 340.0);
    rgb_color  userColorChoice;
    rgb_color  origHighColor;

    MovePenTo(BPoint(20.0, 20.0));
    DrawString("Choose a color below, then move the cursor");
    MovePenTo(BPoint(20.0, 35.0));
    DrawString("outside of the color control and click the mouse button");

    origHighColor = HighColor();

    userColorChoice = fColorControl->ValueAsColor();
    SetHighColor(userColorChoice);
    FillRect(aRect, B_SOLID_HIGH);

    SetHighColor(origHighColor);
}
```

Patterns

A pattern is an 8-pixel-by-8-pixel area. Each of the 64 pixels in this area has the color of either the current high or current low color. A pattern can be one solid color (by designating that all pixels in the pattern be only the current high color or only the current low color), or it can be any arrangement of the two colors, as in a checkerboard, stripes, and so forth. Regardless of the arrangement of the pixels that make up the pattern, it can be used to fill an area of any size. And regardless of the size or shape of an area, once a pattern is defined it can be easily "poured" into this area to give the entire area the look of the pattern.

Be-Defined Patterns

You've already encountered three patterns—the Be-defined constants B_SOLID_HIGH, B_SOLID_LOW, B_MIXED_COLORS each specify a specific arrangement of colors in an 8-pixel-by-8-pixel area. Here the B_MIXED_COLORS pattern is used to fill a rectangle with a checkerboard pattern made up of alternating current high and current low colors:

```
BRect  aRect(20.0, 20.0, 300.0, 300.0);

FillRect(aRect, B_MIXED_COLORS);
```

The BView class defines a number of stroke and fill member functions. Each stroke function (such as StrokeRect() and StrokePolygon()) outlines a shape using a specified pattern. Patterns have the greatest effect on the look of a shape outline when the outline has a thickness greater than one pixel (setting the thickness at which lines are drawn is covered ahead in the "The Drawing Pen" section). Each fill function (such as FillRect() and FillPolygon()) fills a shape using a specified pattern. This may not be entirely obvious when looking at some source code snippets because these drawing routines make the pattern parameter optional. When the pattern parameter is skipped, the function uses the B_SOLID_HIGH pattern by default. So both of the following calls to FillRect() produce a rectangle filled with a solid pattern in the current high color:

```
BRect  rect1(100.0, 100.0, 150.0, 150.0);
BRect  rect2(150.0, 150.0, 200.0, 200.0);

FillRect(rect1, B_SOLID_HIGH);
FillRect(rect2);
```

Earlier in this chapter, the example project RGBColor demonstrated the use of the `B_SOLID_HIGH`, `B_SOLID_LOW`, and `B_MIXED_COLORS` constants by using these constants in the filling of three rectangles (see Figure 5-1). After setting the high color to red and the low color to blue, the rectangle that was filled using the `B_MIXED_COLORS` constant appeared to be purple. I say "appeared to be purple" because, in fact, none of the pixels in the rectangle are purple. Instead, each is either red or blue. Because the pixels alternate between these two colors, and because pixel density is high on a typical monitor, the resulting rectangle appears to the eye to be solid purple. Figure 5-6 illustrates this by showing the RGBColor program's window and the window of the pixel-viewing utility program Magnify. The Magnify program (which is a Be-supplied application that was placed on your machine during installation of the BeOS) shows an enlarged view of the pixels surrounding the cursor. In Figure 5-6, the cursor is over a part of the purple rectangle in the RGBColor window, and the pixels are displayed in the Magnify window.

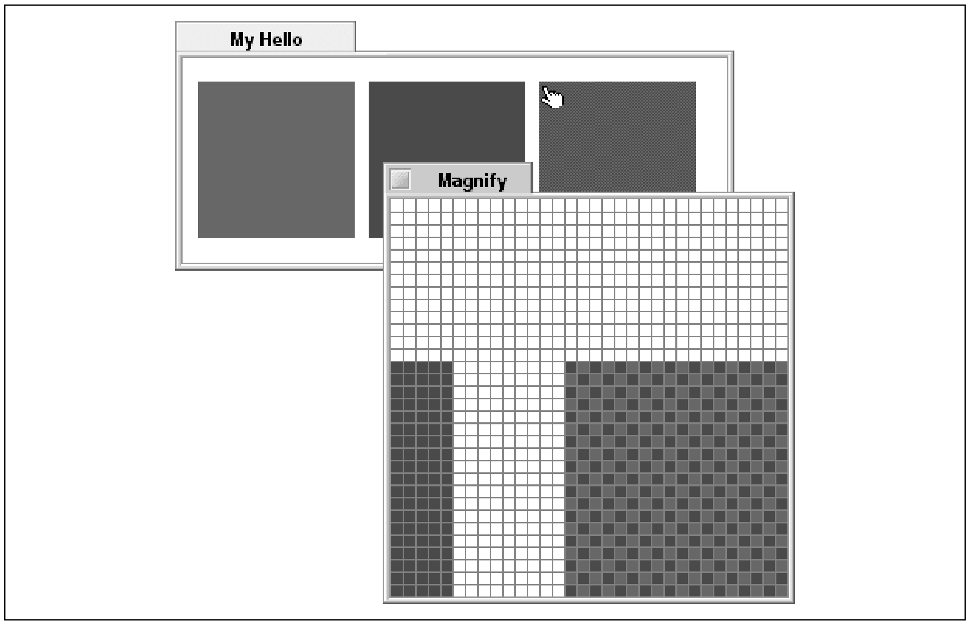


Figure 5-6. Using the Magnify program to view the B_MIXED_COLORS pattern

Application-Defined Patterns

The three Be-defined patterns come in handy, but they don't exploit the real power of patterns. Your project can define a pattern that carries out precisely your idea of what a shape should be filled with.

Bit definition of a pattern

A pattern designates which of the 64 bits (8 rows of 8 bits) in an 8-pixel-by-8-pixel area display the current high color and which display the current low color. Thus the specific colors displayed by the pattern aren't designated by the pattern. Instead, a pattern definition marks each of its 64 bits as either a 1 to display the high color or a 0 to display the low color. The colors themselves come from the high and low colors at the time the pattern is used in drawing.

A pattern is specified by listing the hexadecimal values of the eight bits that make up each row of the pattern. Consider the pattern shown in Figure 5-7. Here I show the 8-by-8 grid for a pattern that produces a diagonal stripe. You can do the same using a pencil and graph paper. Each cell represents one pixel, with a filled-in cell considered on, or 1, and an untouched cell considered off, or 0. Since a pattern defines only on and off, not color, this technique works fine regardless of the colors to be used when drawing with the pattern.

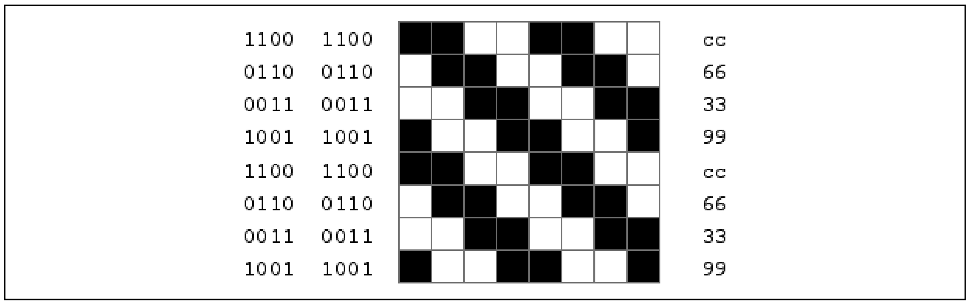


Figure 5-7. The binary and hexadecimal representations of a pattern

The left side of Figure 5-7 shows the binary representation of each row in the pattern, with a row chunked into groups of four bits. The right side of the figure shows the corresponding hexadecimal values for each row. Looking at the top row, from left to right, the pixels are on/on/off/off, or binary 1100. The second set of four pixels in the top row has the same value. A binary value of 1100 is hexadecimal c, so the binary pair translates to the hexadecimal pair cc. The hexadecimal values for each remaining row are determined in the same manner. If you're proficient at working with hexadecimal values, you can skip the intermediate binary step and write the hexadecimal value for each row by simply looking at the pixels in groups of four.

Row by row, the hexadecimal values for the pattern in Figure 5-7 are: cc, 66, 33, 99, cc, 66, 33, 99. Using the convention of preceding a hexadecimal value with 0x, the pattern specification becomes: 0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33, 0x99.

The pattern datatype

Using the previous method to define a pattern isn't just an exercise in your knowledge of hexadecimal numbers, of course! Instead, you'll use a pattern's eight hexadecimal pairs in assigning a `pattern` variable. Here's how Be defines the `pattern` datatype:

```
typedef  struct {
   uchar  data[8];
} pattern;
```

Each of the eight elements in the pattern array is one byte in size, so each can hold a single unsigned value in the range of 0 to 255. Each of the hexadecimal pairs in each of the eight rows in a pattern falls into this range (0x00 = 0, 0xff = 255). To create a `pattern` variable, determine the hexadecimal pairs for the pattern (as shown above) and assign the variable those values. Here I'm doing that for the pattern I designed back in Figure 5-7:

```
pattern  stripePattern = {0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33, 0x99};
```

This is also how Be defines its three built-in patterns. The `B_SOLID_HIGH` pattern is one that has all its bits set to 1, or on; the `B_SOLID_LOW` pattern has all its bits set to 0, or off; and the `B_MIXED_COLORS` pattern has its bits set to alternate between 1 and 0:

```
const pattern B_SOLID_HIGH   = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff}
const pattern B_SOLID_LOW    = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00}
const pattern B_MIXED_COLORS = {0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55, 0xaa, 0x55}
```

Using a pattern variable

Once initialized, a variable of type `pattern` is used just as one of the Be-defined pattern constants—pass the variable as an argument to any routine that requires a pattern as a parameter. The following snippet defines a pattern variable and two rectangle variables. The code then fills one rectangle with a solid color and the other with diagonal stripes:

```
pattern  stripePattern = {0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33, 0x99};
BRect    solidRect(10.0, 10.0, 110.0, 110.0);
BRect    stripedRect(200.0, 10.0, 210.0, 110.0);

FillRect(solidRect, B_SOLID_HIGH);
FillRect(stripedRect, stripePattern);
```

Because the previous snippet doesn't include any code that hints at the current state of the high and low colors, you can't infer what colors will be in the resulting rectangles—you know only that one will have a solid fill while the other will have diagonal stripes running through it.

Pattern example project

The Pattern project builds a program that displays a window with a single rectangle drawn in it. The rectangle is filled with the diagonal stripe pattern that was introduced on the preceding pages. Figure 5-8 shows the Pattern program's window. Also shown is the Magnify program's window as it displays an enlarged view of some of the pixels in the Pattern program's filled rectangle.

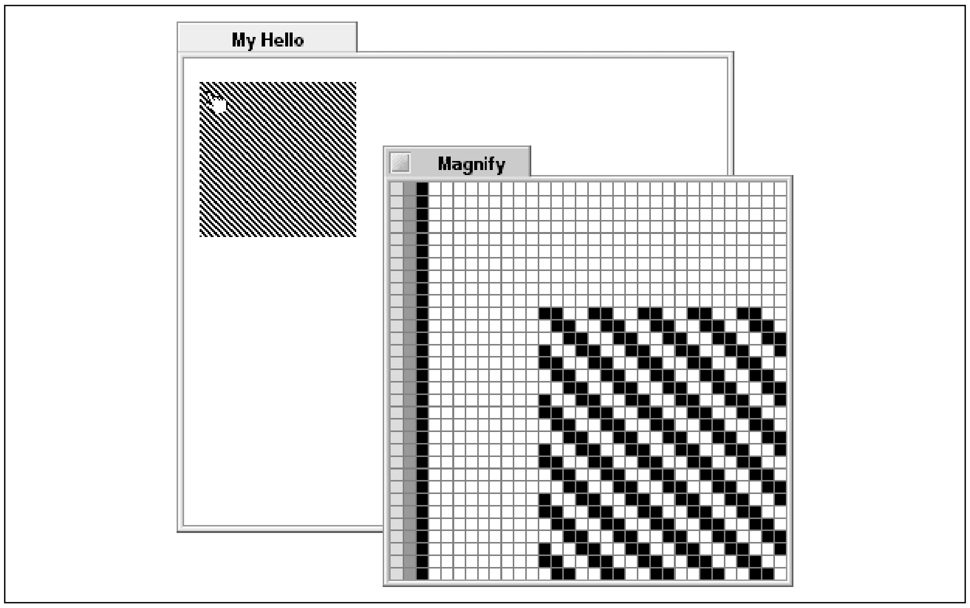


Figure 5-8. Using the Magnify program to view the application-defined pattern

As is the case for many of this chapter's remaining examples, the Pattern project was created by starting with a recent project (such as the RGBColor project) and altering the code in just one of the routines—the `Draw()` member function of the `MyDrawView` class. Here's how the new version of that routine looks:

```
void MyDrawView::Draw(BRect)
{
   BRect    aRect;
   pattern  stripePattern = {0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33, 0x99};

   aRect.Set(10.0, 10.0, 110.0, 110.0);
   FillRect(aRect, stripePattern);
}
```

You can experiment with the Pattern project by adding color to the rectangle. Precede the call to `FillRect()` with a call to `SetHighColor()`, `SetLowColor()`, or both. Changing the current high color will change the color of what are presently the black stripes, while changing the current low color will change the color of what are presently the white stripes.

The Drawing Pen

Whenever drawing takes place, the pen is at work. The pen is a way to summarize and express two properties of a view's drawing environment. When drawing starts, it starts at some specific location in a view. When line drawing takes place, the line has some specific thickness to it. These traits are stored in a view and can be altered by invoking `BView` member functions. In keeping with the analogy of drawing with a pen, the names of these routines include the word "pen."

Pen Location

The location of the rectangle drawn by `Fillrect()` has been previously established during the setting up of the rectangle:

```
BRect  aRect(10.0, 10.0, 110.0, 110.0);

FillRect(aRect, B_SOLID_HIGH);
```

For the drawing of text and lines, this isn't the case. Typically, you'll move the pen to the location where drawing is to start, then draw.

Moving the pen

When it's said that the pen is moved, what's actually taking place is that a call sets the starting location for subsequent drawing. Moving the pen doesn't have any visible effect—nothing gets drawn. To move the pen, invoke the view's `MovePenTo()` or `MovePenBy()` function.

The `MovePenTo()` function accepts either a single `BPoint` argument or a pair of floating-point arguments. In either case, the result is that the arguments specify the coordinate at which the next act of drawing starts. The first argument to `MovePenTo()` is the horizontal coordinate to move to, while the second argument is the vertical coordinate. The movement is relative to the view's origin. Each of the following four calls to `MovePenTo()` has the same result—drawing will be set to start at pixel (30.0, 40.0) in the view's system coordinate:

```
BPoint  aPoint(30.0, 40.0);
float   x = 30.0;
float   y = 40.0;

MovePenTo(aPoint);
MovePenTo(BPoint(30.0, 40.0));
MovePenTo(30.0, 40.0);
MovePenTo(x, y);
```

Like `MovePenTo()`, the `MovePenBy()` function moves the starting location for drawing. `MovePenTo()` moves the pen relative to the view's origin. `MovePenBy()` moves the pen relative to its current location in the view. Consider this snippet:

```
MovePenTo(30.0, 40.0);
MovePenBy(70.0, 10.0);
```

The call to `MovePenTo()` moves the pen to the location 30 pixels from the left of the view and 40 pixels from the top of the view. That places the pen at the point (30.0, 40.0). The call to `MovePenBy()` uses this current location as the reference and moves the pen 70 pixels to the left and 10 pixels down. The result is that, relative to the view's origin, the pen is at the point (100.0, 50.0).

Negative values for `MovePenBy()` move the pen "backwards" in the view. A negative horizontal argument moves the pen to the left, while a negative vertical argument moves the pen up. Because `MovePenTo()` specifies a particular coordinate to end up at rather than a direction to move in, negative values shouldn't be passed as the arguments of this routine.

MovePen example project

As shown in Figure 5-9, the MovePen project results in a program that displays a window with three characters written to it. I've added coordinate information to the figure—the arrows and values aren't actually displayed in the window.

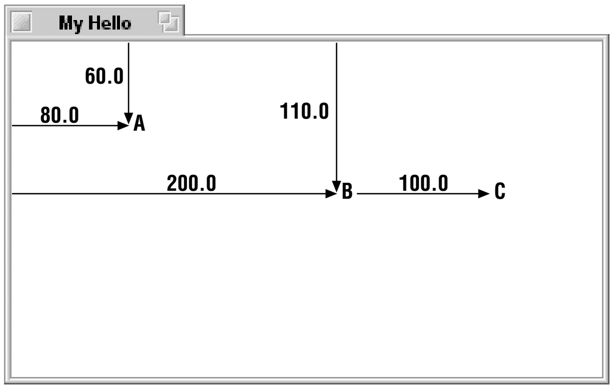


Figure 5-9. The window that results from running the MovePen program

Where the characters "A," "B," and "C" get drawn in the window depends on calls to the `MovePenTo()` and `MovePenBy()` functions. To get a good feel for working with a view's coordinate system, edit the arguments to the existing function calls in the `Draw()` routine, or add additional calls to `MovePenTo()` and `MovePenBy()`:

```
void MyDrawView::Draw(BRect)
{
   MovePenTo(BPoint(80.0, 60.0));
   DrawString("A");

   MovePenTo(200.0, 110.0);
   DrawString("B");

   MovePenBy(100.0, 0.0);
   DrawString("C");
}
```

Pen Size

The pen is used to draw shape outlines. Chapter 4 introduced the `BView` member function `StrokeRect()`, which draws the outline of a rectangle. Here a 100-pixel-by-100-pixel rectangle is framed in a view:

```
BRect  aRect(10.0, 10.0, 110.0, 110.0);

StrokeRect(aRect);
```

As you'll see later in this chapter, there are a number of `BView StrokeXxx()` member functions, each of which outlines a particular shape. All of these functions are affected by the current size of the pen.

Setting the pen size

By default, the pen can be thought of as having a tip that is the size of a single pixel. Thus, drawing with the pen produces a line with a thickness of one pixel. You can change the size or thickness of the pen at any time by invoking the `BView` member function `SetPenSize()`. This routine accepts a single argument, the new pixel thickness for the pen. In this next snippet, the same rectangle that was outlined in the previous snippet is given a frame, or border, three pixels thick. The lines you specify to be drawn are in the center of the pixels in a thick pen.

```
BRect  aRect(10.0, 10.0, 110.0, 110.0);

SetPenSize(3.0);
StrokeRect(aRect);
```

Setting the pen size for a view affects all subsequent calls to `StrokeXxx()` functions. After changing the pen size and invoking a `StrokeXxx()` function, you may want to reset the pen size to a thickness of a single pixel:

```
SetPenSize(3.0);
StrokeRect(aRect);
SetPenSize(1.0);
```

Getting the pen size

The best process to use when altering the pen size is to obtain and save the current pen size, change it, perform the desired drawing using the new pen size, then restore the pen to the saved size. The `BView` member function `PenSize()` allows you to do that. When invoked, `PenSize()` returns a float that holds the current thickness of the pen. This next snippet provides an example of the use of `PenSize()`:

```
float  savedPenSize;

savedPenSize = PenSize();
SetPenSize(3.0);
StrokeRect(aRect);
SetPenSize(savedPenSize);
```

PenSize example project

The PenSize project displays a window that holds six rectangles. As shown in Figure 5-10, the thickness of the outline of each rectangle differs.

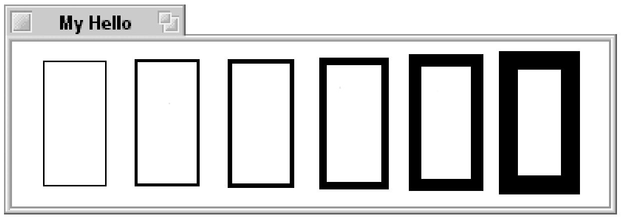


Figure 5-10. The window that results from running the PenSize program

As is becoming a habit, the `Draw()` function of the `MyDrawView` class does the example's work. The function begins by defining and setting six rectangles. Each rectangle has a width of 40 pixels and a height of 80 pixels. The outline thickness of each rectangle differs due to the call to `SetPenSize()` that precedes each call to `StrokeRect()`:

```
void MyDrawView::Draw(BRect)
{
   BRect  rect1(20.0, 20.0, 60.0, 100.0);
   BRect  rect2(80.0, 20.0, 120.0, 100.0);
   BRect  rect3(140.0, 20.0, 180.0, 100.0);
   BRect  rect5(200.0, 20.0, 240.0, 100.0);
   BRect  rect8(260.0, 20.0, 300.0, 100.0);
   BRect  rect12(320.0, 20.0, 360.0, 100.0);

   SetPenSize(1.0);
   StrokeRect(rect1);

   SetPenSize(2.0);
   StrokeRect(rect2);

   SetPenSize(3.0);
   StrokeRect(rect3);
```

```
    SetPenSize(5.0);
    StrokeRect(rect5);

    SetPenSize(8.0);
    StrokeRect(rect8);

    SetPenSize(12.0);
    StrokeRect(rect12);
```

Shapes

Be claims that the BeOS is an operating system for the graphics community. So it's hoped that the BeOS makes it easy for programmers to support the drawing of both simple and complex shapes. And, of course, it does.

Points and Lines

Drawing a line or lines is drawing at its most basic level. Points are used in specifying where a line should be drawn within a view.

The BPoint class

A point is represented by a `BPoint` object. The `BPoint` object consists of two floating point values, one denoting an `x` or horizontal coordinate and the other defining a `y` or vertical coordinate. On a monitor, this coordinate pair matches one particular pixel.

A `BPoint` object can have values assigned to its coordinate pair members either at the time of declaration or anytime thereafter. At declaration, one of the `BPoint` constructors can be used to take care of the task of assigning values to the `BPoint` members:

```
BPoint  aPoint(40.0, 70.0);
```

To assign coordinates after the object's declaration, either use the `BPoint` member function `Set()` or assign values directly to the `x` and `y` data members:

```
BPoint  point1;
BPoint  point2;

point1.Set(100.0, 200.0);
point2.x = 100.0;
point2.y = 200.0;
```

Note the lack of the use of `new` in the previous line of code, and the direct assignment of values to an object's data members. The point is basic to drawing operations, so Be has defined the `BPoint` to act like a basic datatype rather than a class. The declaration of a `BPoint` is all that's needed to actually create a `BPoint` object.

Because the BPoint data members are declared public, direct access is allowed. You've seen a similar situation with the BRect class as well.

Line drawing

Unlike BPoint, the BeOS defines no BLine class to represent lines. Instead, line drawing takes place using the BView member function StrokeLine(). When invoking this function, you can use a pair of BPoint objects to specify the start and end points of a line, or a single BPoint object to specify just the line's end point. If only the end point is specified, the current pen location in the view is used as the start point. Both types of call to StrokeLine() are demonstrated here:

```
BPoint  start1(50.0, 50.0);
BPoint  end1(150.0, 50.0);
BPoint  start2(20.0, 200.0);
BPoint  end2(20.0, 250.0);

StrokeLine(start1, end1);

MovePenTo(start2);
StrokeLine(end2);
```

In the previous snippet, two lines are drawn. The first is a horizontal line 100 pixels in length that is drawn from the point (50.0, 50.0) to the point (150.0, 50). The second is a vertical line 50 pixels in length that is drawn from the point (20.0, 200. 0) to the point (20.0, 250.0).

Both versions of StrokeLine() offer a final optional parameter that specifies a pattern in which the line is drawn. The following snippet draws lines of the same length and location as the previous snippet (assume the same BPoint variables are declared). Here, however, the horizontal line is red and the vertical line is green:

```
rgb_color  redColor   = {255, 0, 0, 255};
rgb_color  greenColor = {0, 255, 0, 255};

SetHighColor(redColor);
SetLowColor(greenColor);

StrokeLine(start1, end1, B_SOLID_HIGH);

MovePenTo(start2);
StrokELine(end2, B_SOLID_LOW);
```

You aren't limited to the Be-defined patterns—any pattern can be used as an argument to StrokeLine(). In this next snippet, the diagonal stripe pattern discussed earlier in this chapter is used. The high color is set to red, and the low color is left in its default state of white. Additionally, the pen size is set to 10.0. The result is a line 10 pixels in thickness with diagonal red stripes running through it. Figure 5-11 shows the line that is drawn from this snippet:

```
pattern    stripePattern = {0xcc, 0x66, 0x33, 0x99, 0xcc, 0x66, 0x33, 0x99};
rgb_color  redColor      = {255, 0, 0, 255};
BPoint     start1(50.0, 50.0);
BPoint     end1(150.0, 50.0);

SetHighColor(redColor);

SetPenSize(10.0);

StrokeLine(start1, end1, stripePattern);
```

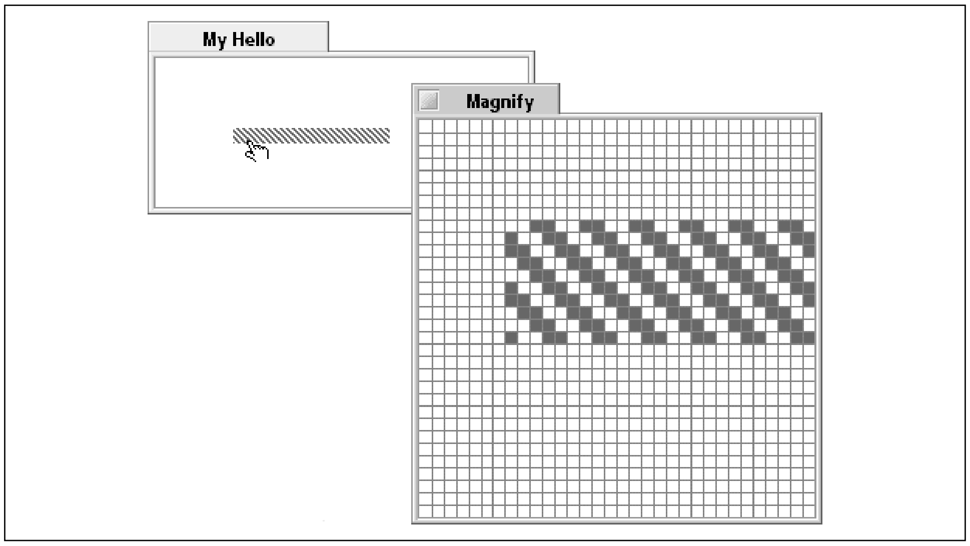


Figure 5-11. Using the Magnify program to view a thick, patterned line

PointAndLine example project

The PointAndLine project is a simple exercise in moving the pen and drawing lines. The resulting program draws the two lines pictured in Figure 5-12.

The `MyDrawView` member function `Draw()` has been written to define two `BPoint` objects that specify the start and end of a horizontal line. After drawing the line, the pen is first moved to the line's start point, and then moved down a number of pixels. `StrokeLine()` is called again, this time to draw a diagonal line that ends at the same point where the first line ended.

```
void MyDrawView::Draw(BRect)
{
   BPoint   point1;
   BPoint   point2;
```

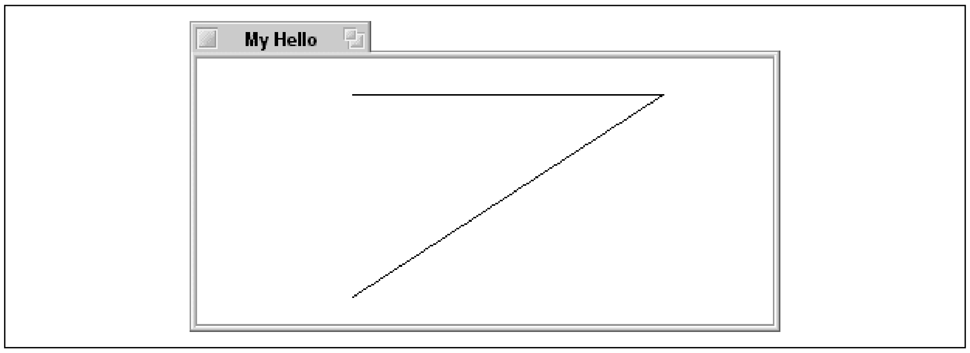


Figure 5-12. The window that results from running the PointAndLine program

```
        point1.Set(100.0, 80.0);
        point2.Set(300.0, 80.0);

        StrokeLine(point1, point2);

        MovePenTo(point1);
        MovePenBy(0.0, 130.0);
        StrokeLine(point2);
    }
```

Rectangles

The `BRect` class is used to create objects that represent rectangles. Rectangles are important in their own right, as they're the basis for defining the boundaries of interface elements such as windows and controls. But they are also instrumental in the definition of other shapes, including round rectangles, ellipses, and regions—as you'll see ahead.

Setting a rectangle

A `BRect` object has four data members of type `float`: `left`, `top`, `right`, and `bottom`. Each data member specifies a pixel location that defines one of the four edges of a rectangle. The values of the `left` and `right` members are relative to the left edge of the view that is to hold the rectangle, while the values of the `top` and `bottom` members are relative to the top edge of the view.

Upon declaration, the boundaries of a `BRect` object can be set by specifying values for each of the four data members or by listing two points—one of which specifies the rectangle's top left corner while the other specifies the rectangle's bottom right corner. In this next snippet, both of the `BRect` objects would have the same boundaries:

```
    BRect   rect1(10.0, 30.0, 110.0, 130.0);

    BPoint  leftTopPt(10.0, 30.0);
```

```
BPoint  rightBottomPt(110.0, 130.0);
BRect   rect2(leftTopPt, rightBottomPt);
```

A rectangle's data members can also be set after the `BRect` object's declaration. Again, the following rectangle shares the same coordinates as the previous two:

```
BRect  aRect;

aRect.left   =  10.0;
aRect.top    =  30.0;
aRect.right  = 110.0;
aRect.bottom = 130.0;
```

A more efficient means of achieving the above result is to use the `BRect` member function `Set()`. The order of the values passed to `Set()` is `left`, `top`, `right`, `bottom`:

```
BRect  aRect;

aRect.Set(10.0, 30.0, 110.0, 130.0);
```

Like a point, which is represented by a `BPoint` object, a rectangle is considered basic to drawing. The `BRect` class is set up so that a `BRect` object is created upon declaration of a `BRect`. Additionally, the `BRect` data members are public, so they can be accessed either directly or via member functions such as `Set()`. In short, `BRect` objects and `BPoint` objects can and should be created on the stack (declared locally).

Drawing a rectangle

In Chapter 4, you saw that shape drawing is achieved by invoking a `BView` member function. There, `StrokeRect()` was called to draw a framing rectangle around a view. In this chapter, you've already seen that a rectangle is filled by calling `FillRect()`. Both routines accept as an argument a previously set up rectangle. This snippet calls `StrokeRect()` to frame one rectangle, then calls `FillRect()` to fill a smaller rectangle centered within the first:

```
BRect  outerRect(10.0, 10.0, 160.0, 160.0);
BRect  innerRect(30.0, 30.0, 140.0, 140.0);

StrokeRect(outerRect);
FillRect(innerRect);
```

Both `StrokeRect()` and `FillRect()` accept an optional `pattern` parameter. Here the rectangles just defined are drawn with a black and white checkerboard pattern:

```
StrokeRect(outerRect, B_MIXED_COLORS);
FillRect(innerRect, B_MIXED_COLORS);
```

Rectangles example project

The Rectangles project displays a window consisting of the rectangles shown in Figure 5-13.

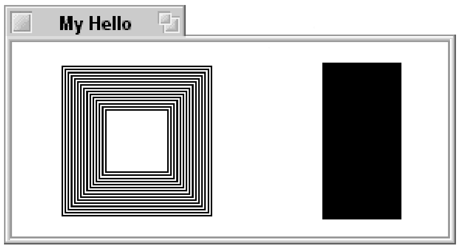


Figure 5-13. The window that results from running the Rectangles program

The left rectangle consists of many nested, or inset, rectangles. The `MyDrawView` member function `Draw()` sets up what will be the outermost rectangle shown. After the `BRect` object's boundaries are set, a `for` loop insets and draws each rectangle. The `BRect` member function `InsetBy()` creates the desired inset effect by adjusting the values of the data members of the calling `BRect` object. After fifteen rectangles are framed one inside the other, a solid black rectangle is drawn to the right of the window:

```
void MyDrawView::Draw(BRect)
{
   int32  i;
   BRect  aRect;

   aRect.Set(30.0, 30.0, 130.0, 130.0);

   for (i = 0; i < 15; i++) {
      aRect.InsetBy(2.0, 2.0);
      StrokeRect(aRect);
   }

   aRect.Set(200.0, 30.0, 250.0, 130.0);
   FillRect(aRect);
}
```

Round rectangles

A rectangle with rounded corners is a shape that appears in both user-interface elements (most buttons are round rectangles) and everyday objects (picture some of the boxes in a flow chart). You can use the `BView` member function `StrokeRoundRect()` to take an "ordinary" rectangle—a `BRect` object—and provide a degree of rounding to each of its corners.

In `StrokeRoundRect()`, the first argument is a previously set up rectangle. The second and third `StrokeRoundRect()` parameters specify the amount of rounding to be applied to each corner. Together these two parameters specify the shape of an ellipse; the second parameter establishes the ellipse radius along the x-axis, while the third parameter establishes the ellipse radius along the y-axis. Think of setting an ellipse in each corner of the rectangle and drawing only one quarter of the ellipse in order to form a rounded corner. The following snippet demonstrates a call to `StrokeRoundRect()`:

```
BRect   aRect;

aRect.Set(30.0, 30.0, 130.0, 130.0);

StrokeRoundRect(aRect, 20.0, 20.0);

FillRoundRect(aRect, 30.0, 30.0);
```

To fill a round rectangle, call the `BView` member function `FillRoundRect()`. The parameters to this function match those used in `StrokeRoundRect()`:

```
BRect   aRect;

aRect.Set(30.0, 30.0, 130.0, 130.0);

FillRoundRect(aRect, 20.0, 20.0);
```

Like other `BView` drawing routines, both `StrokeRoundRect()` and `FillRoundRect()` accept an optional `pattern` argument. Omitting this argument tells the view to use `B_SOLID_HIGH`.

Ellipses

You've seen that a line isn't represented by a class, but is instead drawn using a `BView` member function. A similar situation exists for an ellipse. And like a round rectangle, an ellipse is specified using a rectangle.

Drawing an ellipse

After defining a rectangle, an ellipse is drawn by inscribing an oval within the boundaries of the rectangle. The rectangle itself isn't drawn—it simply serves to specify the size of the ellipse. A circle, of course, is an ellipse whose height and width are equal.

The `BView` member function `StrokeEllipse()` does the drawing:

```
BRect  aRect;

aRect.Set(30.0, 30.0, 130.0, 130.0);
StrokeEllipse(aRect);
```

An ellipse can be filled by the `BView` member function `FillEllipse()`:

```
BRect  aRect;

aRect.Set(200.0, 30.0, 250.0, 130.0);
FillEllipse(aRect);
```

By default, an ellipse is outlined or filled using a solid pattern and the current high color. As with other `BView` drawing routines, a call to either `StrokeEllipse()` or `FillEllipse()` can include an optional second parameter that specifies a different pattern.

An alternate method of specifying the boundaries of an ellipse is to supply `StrokeEllipse()` or `FillEllipse()` with a center point and an x-axis and y-axis radius. The next snippet draws a single ellipse that is both outlined and filled. A call to `SetLowColor()` sets the low color to blue. The subsequent call to `FillEllipse()` draws an ellipse that is filled with a black and blue checkerboard pattern. Specifying the `B_MIXED_COLORS` pattern in the call to `FillEllipse()` tells the routine to fill the ellipse with a checkerboard pattern that alternates the current high color (by default, black) with the current low color (which was just set to blue). The call to `StrokeEllipse()` relies on the current high color (black) to outline the ellipse in a solid pattern (`B_SOLID_HIGH`, since no pattern was specified). Note that when both filling and outlining a single ellipse it's important to call the `StrokeEllipse()` routine after calling `FillEllipse()`. If the calling order is reversed, the call to `FillEllipse()` will obscure whatever outline was drawn by the call to `StrokeEllipse()`:

```
rgb_color  blueColor = {0, 0, 255, 255};
BPoint     center(100.0, 100.0);
float      xRadius = 40.0;
float      yRadius = 60.0;

SetLowColor(blueColor);
FillEllipse(center, xRadius, yRadius, B_MIXED_COLORS);

StrokeEllipse(center, xRadius, yRadius);
```

Polygons

A polygon is a closed shape with straight sides. Unlike a rectangle, which consists of sides that must be vertical and horizontal, a polygon can include sides that run diagonally.

Setting up a polygon

The `BPolygon` class is used to create objects that represent polygons. When creating a new `BPolygon` object, pass the `BPolygon` constructor an array of `BPoint`s that specify the points that make up the vertices of the polygon, along with an int32 value that specifies how many points are in the array. Here's how a BPolygon object representing a three-sided polygon might be defined:

```
BPoint    pointArray[3];
int32     numPoints = 3;
BPolygon  *aPolygon;

pointArray[0].Set(50.0, 100.0);
pointArray[1].Set(150.0, 20.0);
pointArray[2].Set(250.0, 100.0);

aPolygon = new BPolygon(pointArray, numPoints);
```

The previous snippet creates an array of three BPoint objects. The coordinates of each BPoint object are then assigned by calling the BPoint member function Set(). Next, new is used to create a new BPolygon object whose vertices are defined by the BPoints in the pointArray array.

Alternatively, a BPolygon can be created without immediately defining its vertices. In this case, the polygon can be defined later by calling the BPolygon member function AddPoints(), which adds an array of BPoint objects to the BPolygon object. Here, the same polygon that was defined in the previous snippet is again defined—this time using a call to AddPoints():

```
BPoint    pointArray[3];
int32     numPoints = 3;
BPolygon  *aPolygon;

aPolygon = new BPolygon();

pointArray[0].Set(50.0, 100.0);
pointArray[1].Set(150.0, 20.0);
pointArray[2].Set(250.0, 100.0);

aPolygon->AddPoints(pointArray, numPoints);
```

Drawing a polygon

Once a BPolygon object is defined, its outline can be drawn by calling the BView member function StrokePolygon(), or it can be filled in with the BView member function FillPolygon(). If the following call to StrokePolygon() is made after either of the previous two snippets execute, the result is an outlined triangle.

```
StrokePolygon(aPolygon);
```

Note that StrokePolygon() draws the lines of the polygon's edges starting at the first point in the array, and finishes by closing the polygon with a line from the last point back to the first point. Thus, the previous call to StrokePolygon() draws the following three lines.

- From point (50.0, 100.0) to point (150.0, 20.0)
- From point (150.0, 20.0) to point (250.0, 100.0)
- From point (250.0, 100.0) back to point (50.0, 100.0)

A call to `FillPolygon()` fills the previously defined polygon with a pattern. Here the triangle polygon is filled with the default pattern of `B_SOLID_HIGH`:

```
FillPolygon(aPolygon);
```

As you've come to expect from the `BView` `StrokeXxx()` and `FillXxx()` routines, both `StrokePolygon()` and `FillPolygon()` may include an optional pattern polygon that specifies a pattern to use for the outline or fill of the polygon.

Finally, you can include yet one more optional parameter in the polygon drawing routines—a `bool` value that indicates whether the final shape-closing line should or shouldn't be drawn. By default, this line is drawn to finish up the polygon. If false is passed, a series of connected lines rather than a polygon will be drawn.

Drawing a triangle

On the previous pages you saw how to define and draw a triangle using a `BPolygon` object. The BeOS also provides a shortcut for working with this special case of a polygon. It's useful because so many 3D objects are built from dozens of tiny triangles. Instead of defining an array of points and adding those points to a `BPolygon` object, you can forego the both these steps and simply call one of two `BView` member functions: `StrokeTriangle()` or `FillTriangle()`. Pass the three `BPoint` objects that define the triangle's vertices. Here, the same triangle that was drawn using a `BPolygon` object is drawn using a call to `StrokeTriangle()`:

```
BPoint  point1;
BPoint  point2;
BPoint  point3;

point1.Set(50.0, 100.0);
point2.Set(150.0, 20.0);
point3.Set(250.0, 100.0);
StrokeTriangle(point1, point2, point3);
```

As expected, `FillTriangle()` is invoked in the same manner as `StrokeTriangle()`. Again as expected, both functions accept an optional pattern parameter.

Regions

A region groups a number of rectangles together into a single `BRegion` object. This object can then be manipulated as a unit—rotated, colored, and so forth. The rectangles that make up a region can vary in size, and can be defined such that they form one continuous shape or any number of seemingly unrelated shapes. In Figure 5-14, a window has been divided into four areas. Each area holds a single region. That last point is worthy of repeating: the window holds four regions. The rectangles in the area at the bottom left of the window don't form a single, solid area, but because they've been marked to all be a part of a single `BRegion` object, they collectively make up a single region. The same applies to the rectangles in the lower right area of the window.

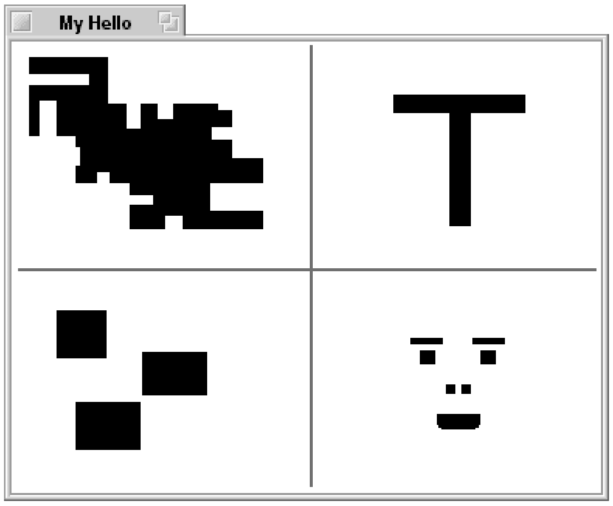


Figure 5-14. A window displaying four regions

Setting up a region

To set up a region, first create a new `BRegion` object. Then define a rectangle and add that rectangle to the `BRegion`. In the following snippet, two rectangles are defined and added to a `BRegion` object named `aRegion`:

```
BRect    aRect;
BRegion  *aRegion;

aRegion = new BRegion();

aRect.Set(20.0, 20.0, 70.0, 70.0);
aRegion->Include(aRect);

aRect.Set(50.0, 50.0, 150.0, 100.0);
aRegion->Include(aRect);
```

Drawing a region

Creating a region and adding rectangles to it defines the area (or areas) the region occupies. It doesn't, however, display the region in a view. To do that, invoke the

BView member function FillRegion(). Here, the region that was created in the previous snippet is filled with the default B_SOLID_HIGH pattern:

```
FillRegion(aRegion);
```

The BRegion object keeps track of all of the constituent rectangles and is responsible for filling each. Like other FillXxx() functions FillRegion() allows you to specify an optional pattern.

There is no StrokeRegion() member function in the BView class. Because a region can consist of any number of overlapping rectangles, outlining each individual rectangle would result in lines running through the contents of the region.

Testing for inclusion in a region

One important use of a region is to test where a point lies within an area. If a point lies in any one of the rectangles that defines a region, that point is considered a part of, or belonging to, the region. To test a point for inclusion in a region, call the BRegion member function Contains(). Pass this routine the BPoint object to test, and Contains() will return a bool value that indicates whether or not the tested point lies within the region. In this next snippet, a region consisting of just a single rectangle is defined. A point is then defined and tested for inclusion in this region:

```
BRect    aRect(20.0, 20.0, 70.0, 70.0);
BRegion  *aRegion;
BPoint   aPoint(60.0, 90.0);

aRegion = new BRegion();
aRegion->Include(aRect);
FillRegion(aRegion);

if (aRegion->Contains(aPoint))
   // do something
else
   // do something else
```

Region example project

The Region project results in a program that displays a window like the one shown in Figure 5-15.

The dark area in Figure 5-15 represents a single region composed of three rectangles. The MyDrawView member function Draw() defines and adds each rectangle in turn to the BRegion object. After all of the rectangles have been

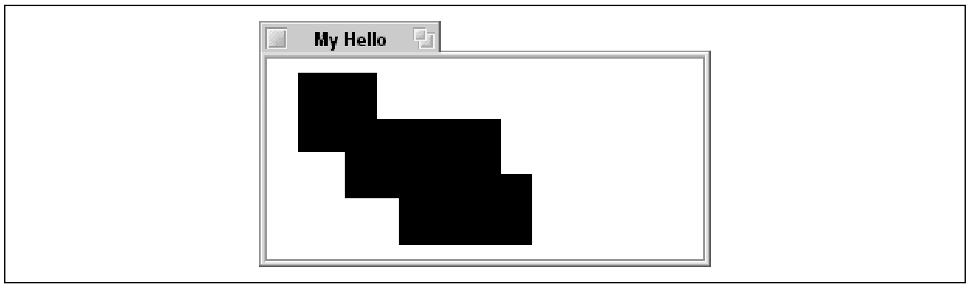


Figure 5-15. The window that results from running the Region program

added, `FillRegion()` is called to fill the entire region with the default pattern of `B_SOLID_HIGH`:

```
void MyDrawView::Draw(BRect)
{
   BRect    aRect;
   BRegion  *aRegion;

   aRegion = new BRegion();

   aRect.Set(20.0, 20.0, 70.0, 70.0);
   aRegion->Include(aRect);

   aRect.Set(50.0, 50.0, 150.0, 100.0);
   aRegion->Include(aRect);

   aRect.Set(85.0, 85.0, 170.0, 130.0);
   aRegion->Include(aRect);
   FillRegion(aRegion);
}
```

Region point testing example project

While the region shown in Figure 5-15 is filled, it doesn't have to be. By omitting a call to `FillRegion()`, it's possible to define a region without making it visible. One practical reason for doing this is for testing mouse button hits. If your program needs to find out whether the user clicked in a particular area of a window—even a very irregularly shaped area—a region that includes the entire area to test can be set up. The `BRegion` class includes a routine that tests for the inclusion of a `BPoint`. The `BView` member function `MouseDown()` provides your application with a `BPoint` object that holds the pixel location of the cursor at the time of a mouse button click. By overriding a view's `MouseDown()` function and implementing it such that it compares the cursor location with the area of the region, your program can easily respond to mouse button clicks in an area of any shape.

The RegionTest project creates a program that draws a region to a window—the same region created by the Region example project and pictured in Figure 5-15.

What's been added is the ability to test for a mouse button click in the region. If the user clicks anywhere in the dark area, the system beep is played. Clicking in the window but outside of the region produces no effect.

To let the `MyDrawView` view (which occupies the entire area of the window it resides in) respond to mouse button clicks, the program overrides `MouseDown()`. To enable a `MyDrawView` object to keep track of its region, a `BRegion` object has been added as a private data member in the `MyDrawView` class. Here's the new declaration of the `MyDrawView` class:

```
class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void  AttachedToWindow();
      virtual void  Draw(BRect updateRect);
      virtual void  MouseDown(BPoint point);

   private:
      BRegion        *fThreeRectRegion;
};
```

In the previous example (the Region project), the region was set up in the `Draw()` function. Because in that project the `MyDrawView` class didn't retain the information about the region (it didn't define a `BRegion` object as a data member), it was necessary to recalculate the region's area each time the view it resided in needed updating. Now that the view retains this information, the `BRegion` needs to be set up only once. The `MyDrawView` constructor is used for that purpose:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect    aRect;

   fThreeRectRegion = new BRegion();

   aRect.Set(20.0, 20.0, 70.0, 70.0);
   fThreeRectRegion->Include(aRect);

   aRect.Set(50.0, 50.0, 150.0, 100.0);
   fThreeRectRegion->Include(aRect);

   aRect.Set(85.0, 85.0, 170.0, 130.0);
   fThreeRectRegion->Include(aRect);
}
```

With the `BRegion` construction moved to the `MyDrawView` constructor, the implementation of the `MyDrawView` member function `Draw()` is reduced to nothing more than a call to `FillRegion()`:

```
void MyDrawView::Draw(BRect)
{
   FillRegion(fThreeRectRegion);
}
```

Implementing the `MouseDown()` routine for the `MyDrawView` class is as easy as comparing the cursor location (supplied by the system when it automatically invokes `MouseDown()`) to the area of the region. The `BRegion` member function `Contains()` handles that task:

```
   void MyDrawView::MouseDown(BPoint point)
   {
      if (fThreeRectRegion->Contains(point))
         beep();
   }
```

If the cursor is over any part of the region when the mouse button is clicked, the computer beeps. Your more sophisticated program will of course find something more interesting to do in response to the user clicking the mouse button in a region!

Pictures

Now that you know about the myriad shapes the BeOS allows you to draw, you'll appreciate the `BPicture` class. A `BPicture` object consists of any number of shapes of any type. Once defined, this combination of one, two, or hundreds of shapes can be quickly drawn and redrawn with a single function call.

Setting up a picture

A picture object is created by using `new` to allocate memory for an empty, temporary `BPicture`. The creation of the new object takes place within a call to the `BView` member function `BeginPicture()`. Doing so instructs the view to store the results of subsequent calls to `BView` drawing functions in the picture object, as opposed to drawing the results of the calls to the view (as is normally the case). When the picture is considered complete, the `BView` member function `EndPicture()` is called. This routine returns the completed temporary picture object. This temporary object should be assigned to an application-defined `BPicture` object. The following snippet provides an example:

```
   BPicture  *aPicture;

   BeginPicture(new BPicture);
      BRect  aRect;

      aRect.Set(10.0, 10.0, 30.0, 30.0);
      FillRect(aRect);
      MovePenTo(40.0, 10.0);
      StrokeLine(BPoint(60.0, 10.0));
   aPicture = EndPicture();
```

For simplicity, the previous snippet defines a picture that consists of a small filled-in rectangle and a short, horizontal line. Your own pictures may prove to be far more complex. In particular, you'll want to include calls such as `SetFont()`, `SetPenSize()`, and so forth in order to set up the drawing environment appropriate to your drawing. If such calls aren't made, the state of the view's drawing environment at the time the picture is created will be used when the picture is later drawn.

The previous snippet indents the code between the `BeginPicture()` and `EndPicture()` calls for purely aesthetic reasons. Doing so isn't necessary, but it does make it obvious just what code the picture object consists of.

Completing a picture by calling `EndPicture()` doesn't prohibit you from adding to that same picture at a later time. To add to an existing `BPicture` object, pass that object to `BeginPicture()` and start drawing. After calling `EndPicture()`, the new drawing code will be a part of the existing picture. Here the previous picture-creating snippet is repeated. After the picture is completed, it is reopened and a vertical line is added to it:

```
BPicture  *aPicture;

BeginPicture(new BPicture);
   BRect  aRect;

   aRect.Set(10.0, 10.0, 30.0, 30.0);
   FillRect(aRect);_
   MovePenTo(40.0, 10.0);
   StrokeLine(BPoint(60.0, 10.0));
aPicture = EndPicture();
...
...
BeginPicture(aPicture);
   MovePEnTo(10.0, 40.0);
   StrokeLine(BPoint(10.0, 60.0));
aPicture = EndPicture();
```

Drawing a picture

Once defined, a picture is drawn by invoking the `BView` member function `DrawPicture()`. Just pass the picture as the argument:

```
DrawPicture(aPicture);
```

Before calling `DrawPicture()`, specify the starting position of the picture by moving the pen to where the upper left corner should be. Alternately, a `BPoint` object can be passed to `DrawPicture()` to denote where drawing should start. Both techniques are shown here. The result of executing the code will be two identical pictures, one beneath the other.

```
BPicture  *aPicture;

BeginPicture(new BPicture);
   // line and shape-drawing code here
aPicture = EndPicture();

MovePenTo(100.0, 20.0);
DrawPicture(aPicture);

DrawPicture(aPicture, BPoint(100.0, 250.0);
```

Keep in mind that if a picture hasn't set up its drawing environment, the state of the view at the time the picture was created is used when the picture is drawn. If the view's graphic state happened to be in its default state (a high color of black, a pen size of 1.0, and so forth), the drawing will be made as expected. However, if any of the view's graphic settings were altered at some point before the picture was created, the drawing of the picture may occur with unpredictable and undesirable effects. Consider the previous snippet. If a call to `SetPenSize(10.0)` had been made somewhere before the call to `BeginPicture()`, any line drawing done by the `aPicture` picture would include lines with a thickness of 10 pixels. That most likely won't be the desired effect. If the picture is to draw lines with a thickness of 1 pixel, then a call to `SetPenSize(1.0)` should be made in the picture-defining code, like this:

```
BPicture  *aPicture;

BeginPicture(new BPicture);
   SetPenSize(1.0);
   // line and shape-drawing code here
aPicture = EndPicture();
```

You don't have to keep track and reverse changes that are made to the drawing environment by a picture. That is, after a call to `EndPicture()`, your code doesn't need to restore the view's environment to its pre-`BeginPicture()` state. That's because all environmental changes that are made between calls to `BeginPicture()` and `EndPicture()` apply only to the picture, not to the view.

Picture example project

The Picture project draws a number of cascading rectangles, as shown in Figure 5-16.

For this project I've added a private data member `BPicture` object named `fPicture` to the `MyDrawView` class. In the `MyDrawView` member function `AttachedToWindow()`, this picture is created and defined. A `for` loop is used to set up the numerous offset rectangles that make up the picture.

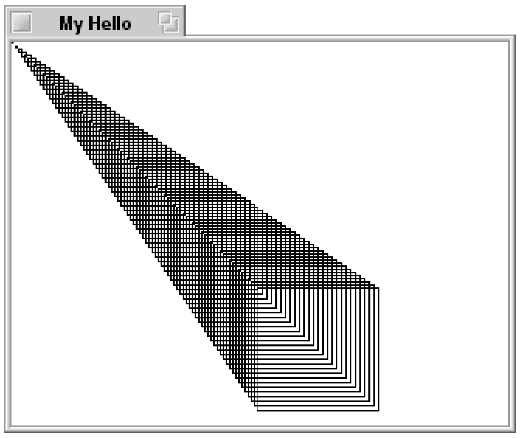


Figure 5-16. The window that results from running the Picture program

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);

   BeginPicture(new BPicture);
      BRect  aRect;
      int32  i;

      for (i=0; i<80; i++) {
         aRect.Set(i*2, i*2, i*3, i*3);
         StrokeRect(aRect);
      }
   fPicture = EndPicture();
}
```

While the temptation is to define the picture in the `MyDrawView` constructor (it is, after all, an initialization act), the code must instead appear in the `AttachedToWindow()` routine. The `BPicture` definition relies on the current state of the view the picture belongs to, and the view's state isn't completely set up until `AttachedToWindow()` executes.

Once the picture is set up and saved in the `fPicture` data member, a `MyDrawView` object can make use of it. That's done in the `Draw()` function, where a call to `MovePenTo()` ensures that the drawing will start in the top left corner of the view (and, because the view is the same size as the window, the top left corner of the window). A call to `DrawPicture()` performs the drawing:

```
void MyDrawView::Draw(BRect)
{
   MovePenTo(0.0, 0.0);
   DrawPicture(fPicture);
}
```

6 *Controls and Messages*

A control is a graphic image that resides in a window and acts as a device that accepts user input. The BeOS API includes a set of classes that make it easy to add certain predefined controls to a program. These standard controls include the button, checkbox, radio button, text field, and color control. There's also a Be-defined class that allows you to turn any picture into a control. That allows you to create controls that have the look of real-world devices such as switches and dials. Chapter 5, *Drawing*, described the color control and the `BColorControl` class used to create such controls. This chapter discusses other control types and the classes used to create each. Also discussed is the `BControl` class, the class from which all other control classes are derived.

When the user clicks on a control, the system responds by sending a message to the window that holds the control. This message indicates exactly which control has been clicked. The message is received by the window's `MessageReceived()` hook function, where it is handled. Since the `BWindow` version of `MessageReceived()` won't know how to go about responding to messages that originate from your controls, you'll override this routine. Your application then gains control of how such messages are handled, and can include any code necessary to carry out the task you want the control to perform. This chapter includes examples that demonstrate how to create controls and how to override `MessageReceived()` such that the function handles mouse clicks on controls of any of the standard types.

Introduction to Controls

When a `BWindow` object receives a message, it either handles the message itself or lets one of its views handle it. To handle a message, the window invokes a `BWindow` hook function. For example, a `B_ZOOM` message delivered to a window results in that window invoking the `BWindow` hook function `Zoom()` to shrink or enlarge the window. To allocate the handling of a message to one of its views, the window passes the message to the affected view, and the view then invokes the appropriate `BView` hook function. For example, a `B_MOUSE_DOWN` message results in the affected view invoking the `BView` hook function `MouseDown()`.

Besides being the recipient of system messages, a window is also capable of receiving application-defined messages. This lets you implement controls in your application's windows. When you create a control (such as a button object from the `BButton` class), define a unique message type that becomes associated with that one control. Also, add the control to a window. When the user operates the control (typically by clicking on it, as for a button), the system passes the application-defined message to the window. How the window handles the message is determined by the code you include in the `BWindow` member function `MessageReceived()`.

Control Types

You can include a number of different types of controls in your windows. Each control is created from a class derived from the abstract class `BControl`. The `BControl` class provides the basic features common to all controls, and the `BControl`-derived classes add capabilities unique to each control type. In this chapter, you'll read about the following control types:

Button
The `BButton` class is used to create a standard button, sometimes referred to as a push button. Clicking on a button results in some immediate action taking place.

Picture button
The `BPictureButton` class is used to create a button that can have any size, shape, and look to it. While picture buttons can have an infinite variety of looks, they act in the same manner as a push button—a mouse click results in an action taking place.

Checkbox
The `BCheckBox` class creates a checkbox. A checkbox has two states: on and off. Clicking a checkbox always toggles the control to its opposite state or value. Clicking on a checkbox usually doesn't immediately impact the program. Instead, a program typically waits until some other action takes place (such as the click of a certain push button) before gathering the current state of the checkbox. At that time, some program setting or feature is adjusted based on the value in the checkbox.

Radio button

The `BRadioButton` class is used to create a radio button. Like a checkbox, a radio button has two states: on and off. Unlike a checkbox, a radio button is never found alone. Radio buttons are grouped together in a set that is used to control an option or feature of a program. Clicking on a radio button turns off whatever radio button was on at the time of the mouse click, and turns on the newly clicked radio button. Use a checkbox in a yes or no or true or false situation. Use radio buttons for a condition that offers multiple choices that are mutually exclusive (since only one button can be on at any given time).

Text field

The `BTextControl` class is used to create a text field. A text field is a control consisting of a static string on the left and an editable text area on the right. The static text acts as a label that provides the user with information about what is to be typed in the editable text area of the control. Typing text in the editable text area of a control can have an immediate effect on the program, but it's more common practice to wait until some other action takes place (like a click on a push button) before the program reads the user-entered text.

Color control

The `BColorControl` class, shown in Chapter 5, creates color controls. A color control displays the 256 system colors, each in a small square. The user can choose a color by clicking on it. A program can, at any time, check to see which color the user has currently selected, and perform some action based on that choice. Often the selected color is used in the next, or all subsequent, drawing operation the program performs.

Figure 6-1 shows four of the six types of controls available to you. In the upper left of the figure is a button. The control in the upper right is a text field. The lower left of the figure shows a checkbox in both its on and off states, while the lower right of the figure shows a radio button in both its states. A picture button can have any size and look you want, so it's not shown. All the buttons are associated with labels that appear on or next to the controls themselves.

The sixth control type, the color control based on the `BColorControl` class, isn't shown either—it was described in detail in Chapter 5 and will only be mentioned in passing in this chapter.

A control can be in an enabled state—where the user can interact with it—or a disabled state. A disabled control will appear dim, and clicking on the control will have no effect. Figure 6-2 shows a button control in both its enabled state (leftmost in the figure) and its disabled state (rightmost in the figure). Also shown is what an enabled button looks like when it is selected using the Tab key (middle in the figure). A user can press the Tab key to cycle through controls, making each one in turn the current control. As shown in Figure 6-2, a button's label will be

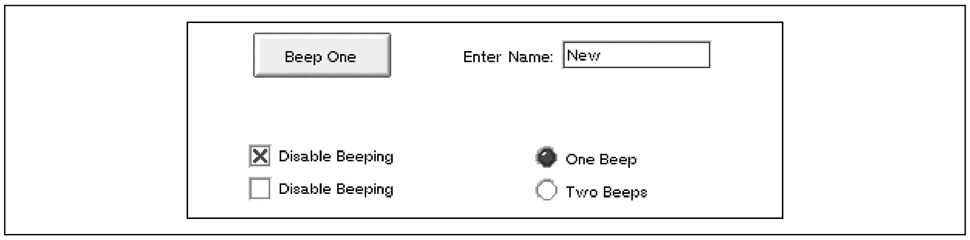


Figure 6-1. Examples of button, text field, checkbox, and radio button controls

underlined when it's current. Once current, other key presses (typically the Return and Enter key) affect that control.

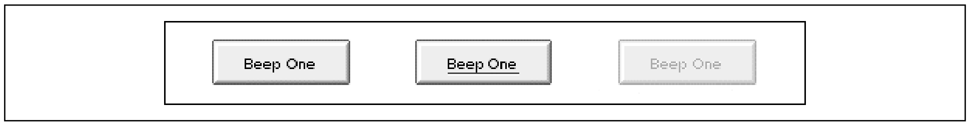


Figure 6-2. A button control that's (from left to right) enabled, current, and disabled

Creating a Control

A control is created from one of six Interface Kit classes—each of which is covered in detail in this chapter. Let us start by examining the `BControl` class from which they are derived.

The BControl class

The `BControl` class is an abstract class derived from the `BView` and `BInvoker` classes. Control objects are created from `BControl`-derived classes, so all controls are types of views.

It's possible to create controls that aren't based on the `BControl` class. In fact, the Be API does that for the `BListView` and `BMenuItem` classes. These are exceptions, though. You'll do best by basing each of your application's controls on one of the six `BControl`-derived classes. Doing so means your controls will behave as expected by the user.

`BControl` is an abstract class, so your project will create `BControl`-derived class objects rather than `BControl` objects. However, because the constructor of each `BControl`-derived class invokes the `BControl` constructor, a study of the `BControl` constructor is a worthwhile endeavor. Here's the prototype:

```
BControl(BRect       frame,
         const char  *name,
         const char  *label,
         BMessage    *message,
         uint32      resizingMode,
         uint32      flags)
```

Four of the six BControl constructor parameters match BView constructor parameters. The frame, name, resizingMode, and flags arguments get passed to the BView constructor by the BControl constructor. These parameters are discussed in Chapter 4, *Windows, Views, and Messages*, so here I'll offer only a brief recap of their purposes. The frame parameter is a rectangle that defines the boundaries of the view. The name parameter establishes a name by which the view can be identified at any time. The resizingMode parameter is a mask that defines the behavior of the view should the size of the view's parent view change. The flags parameter is a mask consisting of one or more Be-defined constants that determine the kinds of notifications (such as update) the view is to be aware of.

The remaining two BControl constructor parameters are specific to the control. The label parameter is a string that defines the text associated with it. For instance, for a button control, the label holds the words that appear on the button. The message parameter is a BMessage object that provides a means for the system to recognize the control as a unique entity. When the control is selected by the user, it is this message that the system will send to the window that holds the control.

Your project won't create BControl objects, so a sample call to the BControl constructor isn't useful here. Instead, let's look at the simplest type of BControl-derived object: the BButton.

The BButton class

Creating a new push button involves creating a new BButton object. The BButton constructor parameters are an exact match of those used by the BControl constructor:

```
BButton(BRect       frame,
        const char  *name,
        const char  *label,
        BMessage    *message,
        uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
        uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The BButton constructor invokes the BControl constructor, passing all of its arguments to that routine. The BControl constructor uses the label argument to initialize the button's label, and uses the message argument to assign a unique message to the button. The BControl constructor then invokes the BView constructor, passing along the remaining four arguments it received from the BButton constructor. The BView constructor then sets up the button as a view. After the

BControl and BView constructors have executed, the BButton constructor carries on with its creation of a button object by implementing button-specific tasks. This is, in essence, how the constructor for each of the BControl-derived classes works.

Creating a button

A button is created by defining the arguments that are passed to the BButton constructor and then invoking that constructor using new. To become functional, the button must then be added to a window. That's what's taking place in this snippet:

```
#define      BUTTON_OK_MSG   'btmg'

BRect        buttonRect(20.0, 20.0, 120.0, 50.0);
const char*  buttonName  = "OKButton";
const char*  buttonLabel = "OK";
BButton     *buttonOK;

buttonOK = new BButton(buttonRect, buttonName,
                       buttonLabel, new BMessage(BUTTON_OK_MSG));

aView->AddChild(buttonOK);
```

In the above code, the BRect variable buttonRect defines the size and location of the button. This push button will be 100 pixels wide by 30 pixels high. The buttonName string gives the button the name "OKButton." This is the name used to locate and access the button by view name using the BView member function FindView(). The name that actually appears on the button itself, "OK," is defined by the buttonLabel string. The message associated with the new button control is a new BMessage object of type BUTTON_OK_MSG. I'll explain the BMessage class in a minute. Here it suffices to say that, as shown above, creating a new message can be as easy as defining a four-character string and passing this constant to the BMessage constructor.

The BButton constructor prototype lists six parameters, yet the above invocation of that constructor passes only four arguments. The fifth and sixth parameters, resizingMode and flags, offer default values that are used when these arguments are omitted. The default resizingMode value (B_FOLLOW_LEFT | B_FOLLOW_TOP) creates a button that will remain a fixed distance from the left and top edges of the control's parent view should the parent view be resized. The default flags value (B_WILL_DRAW | B_NAVIGABLE) specifies that the control needs to be redrawn if altered, and that it can become the focus view in response to keyboard actions.

Adding a control to a window means adding the control to a view. In the above snippet, it's assumed that a view (perhaps an object of the application-defined BView-derived MyDrawView class) has already been created.

Enabling and disabling a control

When a control is created, it is initially enabled—the user can click on the control to select it. If you want a control to be disabled, invoke the control's `SetEnabled()` member function. The following line of code disables the `buttonOK` button control that was created in the previous snippet:

```
buttonOK->SetEnabled(false);
```

`SetEnabled()` can be invoked on a control at any time, but if the control is to start out disabled, call `SetEnabled()` before displaying the window the control appears in. To again enable a control, call `SetEnabled()` with an argument of `true`.

This chapter's CheckBoxNow project demonstrates the enabling and disabling of a button. The technique in that example can be used on any type of control.

Turning a control on and off

Checkboxes and radio buttons are two-state controls—they can be on or off. When a control of either of these two types is created, it is initially off. If you want the control on (to check a checkbox or fill in a radio button), invoke the `BControl` member function `SetValue()`. Passing `SetValue()` the Be-defined constant `B_CONTROL_ON` sets the control to on. Turning a control on and off in response to a user action in the control is the responsibility of the system—not your program. So after creating a control and setting it to the state you want, you will seldom need to call `SetValue()`. If you want your program to "manually" turn a control off (as opposed to doing so in response to a user action), have the control invoke its `SetValue()` function with an argument of `B_CONTROL_OFF`.

A button is a one-state device, so turning a button on or off doesn't make sense. Instead, this snippet turns on a two-state control—a checkbox:

```
requirePasswordCheckBox->SetValue(B_CONTROL_ON)
```

Creating checkboxes hasn't been covered yet, so you'll want to look at the Check-Box example project later in this chapter to see the complete code for creating and turning on a checkbox.

Technically, a button is also a two-state control. When it is not being clicked, it's off. When the control is being clicked (and before the user releases the mouse button), it's on. This point is merely an aside, though, as it's unlikely that your program will ever need to check the state of a button in the way it will check the state of a checkbox or radio button.

To check the current state of a control, invoke the `BControl` member function `Value()`. This routine returns an `int32` value that is either `B_CONTROL_ON` (which is defined to be 1) or `B_CONTROL_OFF` (which is defined to be 0). This snippet obtains the current state of a checkbox, then compares the value of the state to the Be-defined constant `B_CONTROL_ON`:

```
int32  controlState;

controlState = requirePasswordCheckBox->Value();
if (controlState == B_CONTROL_ON)
   // password required, display password text field
```

Changing a control's label

Both checkboxes and radio buttons have a label that appears to the right of the control. A text field has a label to the left of the control. The control's label is set when the control is created, but it can be changed on the fly.

The `BControl` member function `SetLabel()` accepts a single argument: the text that is to be used in place of the control's existing label. In this next snippet, a button's label is initially set to read "Click," but is changed to the string "Click Again" at some point in the program's execution:

```
BRect        buttonRect(20.0, 20.0, 120.0, 50.0);
const char   *buttonName  = "ClickButton";
const char   *buttonLabel = "Click";
BButton      *buttonClick;

buttonOK = new BButton(buttonRect, buttonName,
                       buttonLabel, new BMessage(BUTTON_CLICK_MSG));

aView->AddChild(buttonClick);
...
...
buttonClick->SetLabel("Click Again");
```

The labels of other types of controls are changed in the same manner. The last example project in this chapter, the TextField project, sets the label of a button to a string entered by the user.

Handling a Control

`BControl`-derived classes take care of some of the work of handling a control. For instance, in order to properly update a control in response to a mouse button click, your program doesn't have to keep track of the control's current state, and it doesn't have to include any code to set the control to the proper state (such as drawing or erasing the check mark in a checkbox). What action your program takes in response to a mouse button click is, however, your program's responsibility. When the user clicks on a control, a message will be delivered to the affected window. That message will be your program's prompt to perform whatever action is appropriate.

Messages and the BMessage class

When the Application Server delivers a system message to an application window, that message arrives in the form of a BMessage object. Your code determines how to handle a system message simply by overriding a BView hook function (such as MouseDown()). Because the routing of a message from the Application Server to a window and then possibly to a view's hook function is automatically handled for you, the fact that the message is a BMessage object may not have been important (or even known) to you. A control also makes use of a BMessage object. However, in the case of a control, you need to know a little bit about working with BMessage objects.

The BMessage class defines a message object as a container that holds information. Referring to the BMessage class description in the Application Kit chapter of the Be Book, you'll find that this information consists of a name, some number of bytes of data, and a type code. You'll be pleased to find out that when using a BMessage in conjunction with a control, a thorough knowledge of these details of the BMessage class isn't generally necessary (complex applications aside). Instead, all you need to know of this class is how to create a BMessage object. The snippet a few pages back that created a BButton object illustrated how that's done:

```
#define      BUTTON_OK_MSG   'btmg'

// variable declarations here

buttonOK = new BButton(buttonRect, buttonName,
                        buttonLabel, new BMessage(BUTTON_OK_MSG));
```

The only information you need to create a BMessage object is a four-character literal, as in the above definition of BUTTON_OK_MSG as 'btmg'. This value, which will serve as the what field of the message, is actually a uint32. So you can define the constant as an unsigned 32-bit integer, though most programmers find it easier to remember a literal than the unsigned numeric equivalent. This value then becomes the argument to the BMessage constructor in the BButton constructor. This newly created message object won't hold any other information.

The BMessage class defines a single public data member named what. The what data member holds the four-character string that distinguishes the message from all other message types—including system messages—the application will use. In the previous snippet, the constant btmg becomes the what data member of the BMessage object created when invoking the BButton constructor.

When the program refers to a system message by its Be-defined constant, such as B_QUIT_REQUESTED or B_KEY_DOWN, what's really of interest is the what data member of the system message. The value of each Be-defined message constant is a four-character string composed of a combination of only uppercase characters and, optionally, one or more underscore characters. Here's how Be defines a few of the system message constants:

```
enum {
   B_ABOUT_REQUESTED    = '_ABR',
   ...
   ...
   B_QUIT_REQUESTED     = '_QRQ',
   ...
   ...
   B_MOUSE_DOWN         = '_MDN',
   ...
   ...
};
```

Be intentionally uses the message constant value convention of uppercase-only characters and underscores to make it obvious that a message is a system message. You can easily avoid duplicating a Be-defined message constant by simply including one or more lowercase characters in the literal of your own application-defined message constants. And to make it obvious that a message isn't a Be-defined one, don't start the message constant name with "B_". In this book's examples, I have chosen to use a fairly informative convention in choosing symbols for application-defined control messages: start with the control type, include a word or words descriptive of what action the control results in, and end with "MSG" for "message." The value of each constant may hint at the message type (for instance, 'plSD' for "play sound"), but aside from avoiding all uppercase characters, the value is somewhat arbitrary. These two examples illustrate the convention I use:

```
#define    BUTTON_PLAY_SOUND_MSG   'plSD'
#define    CALCULATE_VALUES        'calc'
```

Messages and the MessageReceived() member function

The BWindow class is derived from the BLooper class, so a window is a type of looper—an object that runs a message loop that receives messages from the Application Server. The BLooper class is derived from the BHandler class, so a window is also a handler—an object that can handle messages that are dispatched from a message loop. A window can both receive messages and handle them.

For the most part, system messages are handled automatically; for instance, when a B_ZOOM message is received, the operating system zooms the window. But you cannot completely entrust the handling of an application-defined message to the system.

When a user selects a control, the Application Server delivers a message object with the appropriate `what` data member value to the affected `BWindow` object. You've just seen a snippet that created a `BButton` associated with a `BMessage` object. That `BMessage` had a `what` data member of 'btmg'. If the user clicked on the button that results from this object, the Application Server would deliver such a message to the affected `BWindow`. It's up to the window to include code that watches for, and responds to, this type of message. The `BWindow` class member function `MessageReceived()` is used for this purpose.

When an application-defined message reaches a window, it looks for a `MessageReceived()` function. This routine receives the message, examines the message's `what` data member, and responds depending on its value. The `BHandler` class defines such a `MessageReceived()` function. The `BHandler`-derived class `BWindow` inherits this function and overrides it. The `BWindow` version includes a call to the base class `BHandler` version, thus augmenting what `BHandler` offers. If the `BWindow` version of `MessageReceived()` can't handle a message, it passes it up to the `BHandler` version of this routine. Figure 6-3 shows how a message that can't be handled by one version of `MessageReceived()` gets passed up to the next version of this function.

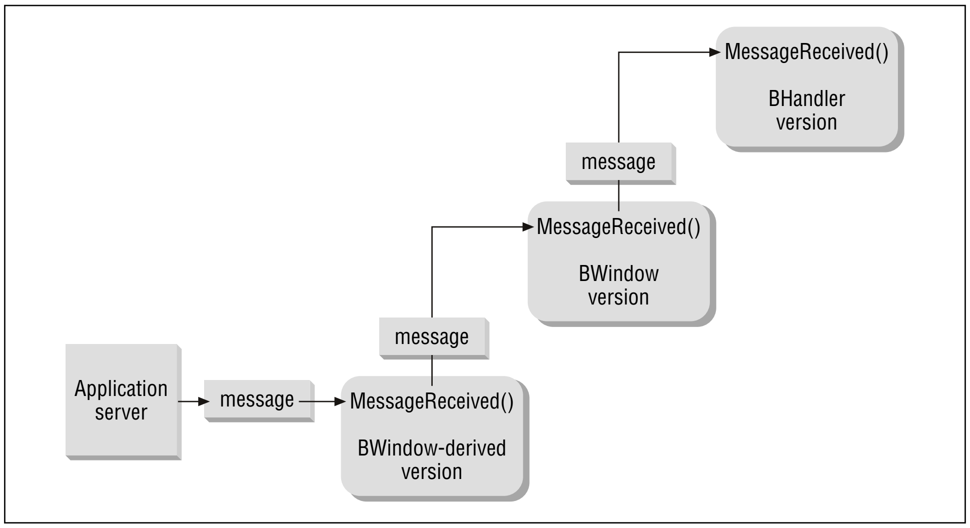


Figure 6-3. Message passed to parent class's version of MessageReceived()

Here is how the `MessageReceived()` looks in `BWindow`:

```
void BWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      // handle B_KEY_DOWN and scripting-related system messages
      default:
         BHandler::MessageReceived(message);
   }
}
```

Your project's windows won't be based directly on the BWindow class. Instead, windows will be objects of a class you derive from BWindow. While such a BWindow-derived class will inherit the BWindow version of MessageReceived(), that version of the function won't suffice—it won't know anything about the application-defined messages you've paired with the window's controls. Your BWindow-derived class should thus do what the BWindow class does: override the inherited version of MessageReceived() and, within the new implementation of this function, invoke the inherited version:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      // handle application-defined messages

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

What messages your BWindow-derived class version of MessageReceived() looks for depends on the controls you're adding to windows of that class type. If I add a single button to windows of the MyHelloWindow class, and the button's BButton constructor pairs a message object with a what constant of BUTTON_OK_MSG (as shown in previous snippets), the MyHelloWindow version of MessageReceived() would look like this:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_OK_MSG:
         // handle a click on the OK button
         break;

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

The particular code that appears under the control's case label depends entirely on what action you want to occur in response to the control being clicked. For simplicity, assume that we want a click on the OK button to do nothing more than sound a beep. The completed version of MessageReceived() looks like this:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_OK_MSG:
         beep();
         break;

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

Buttons

The `BButton` class is used to create a button—a labeled push button that is operated when the button is clicked. The previous sections used the `BButton` class and button objects for its specific examples and in its code snippets. That section provided some background on creating and working with buttons, so the emphasis here will be on incorporating the button-related code in a project.

Creating a Button

The `BButton` constructor has six parameters, each of which was described in the "The BButton class" section of this chapter:

```
BButton(BRect       frame,
        const char  *name,
        const char  *label,
        BMessage    *message,
        uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
        uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The `BButton` constructor calls the `BControl` constructor, which in turn calls the `BView` constructor. Together, these routines set up and initialize a `BButton` object.

After attaching the button to a window, the height of the button may automatically be adjusted to accommodate the height of the text of the button's label and the border of the button. If the values of the `frame` rectangle coordinates result in a button that isn't high enough, the `BButton` constructor will increase the button height by increasing the value of the `frame` rectangle's bottom value. The exact height of the button depends on the font in which the button label is displayed.

For the example button creation code, assume that a window is keeping track of `BView` and `BButton` objects in data members named `fView` and `fButton`, respectively, and that the button's message type is defined by the constant `BUTTON_MSG`:

```
#define   BUTTON_MSG   'bttn'

class MyWindow : public BWindow {
   ...
```

```
      private:
         BView     *fView;
         BButton   *fButton;
   }
```

The code that creates a new button and adds it to the view `fView` might then look like this:

```
BRect  buttonRect(20.0, 20.0, 100.0, 50.0);

fButton = new BButton(buttonRect, "MyButton",
                      "Click Me", new BMessage(BUTTON_MSG));

fView->AddChild(fButton);
```

Making a Button the Default Button

One button in a window can be made the default button—a button that the user can select either by clicking or by pressing the Enter key. If a button is the default button, it is given a wider border so that the user recognizes it as such a button. To make a button the default button, call the `BButton` member function `MakeDefault()`:

```
fButton->MakeDefault(true);
```

If the window that holds the new default button already had a default button, the old default button automatically loses its default status and becomes a "normal" button. The system handles this task to ensure that a window has only one default button.

While granting one button default status may be a user-friendly gesture, it might also not make sense in many cases. Thus, a window isn't required to have a default button.

Button Example Project

The TwoButtons project demonstrates how to create a window that holds two buttons. Looking at Figure 6-4, you can guess that a click on the leftmost button (which is the default button) results in the playing of the system sound a single time, while a click on the other button produces the beep twice.

Preparing the window class for the buttons

A few additions to the code in the *MyHelloWindow.h* file are in order. First, a pair of constants are defined to be used later when the buttons are created. The choice of constant names and values is unimportant, provided that the names don't begin with "B_" and that the constant values don't consist of all uppercase characters.

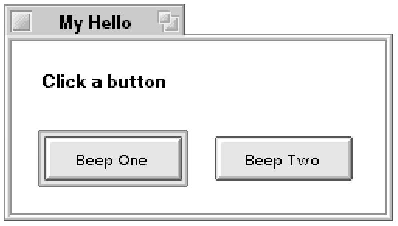


Figure 6-4. The window that results from running the TwoButtons program

```
#define   BUTTON_BEEP_1_MSG  'bep1'
#define   BUTTON_BEEP_2_MSG  'bep2'
```

To keep track of the window's two buttons, a pair of data members of type `BButton` are added to the already present data member of type `MyDrawView`. And now that the window will be receiving and responding to application-defined messages, the `BWindow`-inherited member function `MessageReceived()` needs to overridden:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool    QuitRequested();
      virtual void    MessageReceived(BMessage* message);

   private:
      MyDrawView      *fMyView;
      BButton         *fButtonBeep1;
      BButton         *fButtonBeep2;
};
```

Creating the buttons

The buttons are created and added to the window in the `MyHelloWindow` constructor. Before doing that, the constructor declares several variables that will be used in the pair of calls to the `BButton` constructor and assigns them values:

```
BRect        buttonBeep1Rect(20.0, 60.0, 110.0, 90.0);
BRect        buttonBeep2Rect(130.0, 60.0, 220.0, 90.0);
const char   *buttonBeep1Name  = "Beep1";
const char   *buttonBeep2Name  = "Beep2";
const char   *buttonBeep1Label = "Beep One";
const char   *buttonBeep2Label = "Beep Two";
```

In the past, you've seen that I normally declare a variable within the routine that uses it, just before its use. Here I've declared the six variables that are used as `BButton` constructor arguments outside of the `MyHelloWindow` constructor—but they could just as well have been declared within the `MyHelloWindow` constructor. I opted to do things this way to get in the habit of grouping all of a window's layout-defining code together. Grouping all the button boundary rectangles, names, and labels together makes it easier to lay out the buttons in relation to one another and to supply them with logical, related names and labels. This technique is especially helpful when a window holds several controls.

The buttons will be added to the `fMyView` view. Recall that this view is of the `BView`-derived application-defined class `MyDrawView` and occupies the entire content area of a `MyHelloWindow`. In the `MyHelloWindow` constructor, the view is created first, and then the buttons are created and added to the view:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtonBeep1 = new BButton(buttonBeep1Rect, buttonBeep1Name,
                              buttonBeep1Label,
                              new BMessage(BUTTON_BEEP_1_MSG));

   fMyView->AddChild(fButtonBeep1);
   fButtonBeep1->MakeDefault(true);

   fButtonBeep2 = new BButton(buttonBeep2Rect, buttonBeep2Name,
                              buttonBeep2Label,
                              new BMessage(BUTTON_BEEP_2_MSG));

   fMyView->AddChild(fButtonBeep2);

   Show();
}
```

Handling button clicks

`MessageReceived()` always has a similar format. The Application Server passes this function a message as an argument. The message data member `what` holds the message type, so that data member should be examined in a `switch` statement, with the result compared to any application-defined message types the window is capable of handling. A window of type `MyHelloWindow` can handle a `BUTTON_BEEP_1_MSG` and a `BUTTON_BEEP_2_MSG`. If a different type of message is encountered, it gets passed on to the `BWindow` version of `MessageReceived()`:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   bigtime_t  microseconds = 1000000;  // one second

   switch(message->what)
   {
      case BUTTON_BEEP_1_MSG:
         beep();
```

```
            break;

         case BUTTON_BEEP_2_MSG:
            beep();
            snooze(microseconds);
            beep();
            break;

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

Picture Buttons

A picture button is a button that has a picture on its face rather than a text label. The picture button behaves like a standard push button—clicking and releasing the mouse button while over the picture button selects it.

The `BPictureButton` class is used to create a picture button. Associated with one `BPictureButton` object are two `BPicture` objects. One of the pictures acts as the button when the button is in its normal state (that is, when the user isn't clicking on it). The other picture acts as the button when the user clicks on the button. You'll supply a `BPictureButton` object with the two pictures, and the system will be responsible for switching back and forth between the pictures in response to the user's actions.

Creating a Picture Button

A picture button is created by the `BPictureButton` constructor. As is the case for other controls, this constructor invokes the `BControl` constructor, which in turn invokes the `BView` constructor:

```
BPictureButton(BRect       frame,
               const char  *name,
               BPicture    *off,
               BPicture    *on,
               BMessage    *message,
               uint32      behavior = B_ONE_STATE_BUTTON,
               uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
               uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The `BPictureButton` constructor has eight parameters, five of which you're already familiar with. The `frame`, `name`, `resizingMode`, and `flags` parameters get passed to the `BView` constructor and are used in setting up the picture button as a view. The `message` parameter is used by the `BControl` constructor to assign a message type to the picture button. The remaining three parameters, `off`, `on`, and `behavior`, are specific to the creation of a picture button.

The `off` and `on` parameters are `BPicture` objects that define the two pictures to be used to display the button. In Chapter 5, you saw how to create a `BPicture` object using the `BPicture` member functions `BeginPicture()` and `EndPicture()`. Here I create a picture composed of a white circle within a black circle:

```
BPicture  *buttonOffPict;

fMyView->BeginPicture(new BPicture);
   BRect  aRect(0.0, 0.0, 50.0, 50.0);

   fMyView->FillEllipse(aRect, B_SOLID_HIGH);
   aRect.InsetBy(10.0, 10.0);
   fMyView->FillEllipse(aRect, B_SOLID_LOW);
buttonOffPict = fMyView->EndPicture();
```

A second `BPicture` object should then be created in the same way. These two `BPicture` objects could then be used as the third and fourth arguments to the `BPictureButton` constructor.

For more compelling graphic images, you can use bitmaps for button pictures. Once a bitmap exists, all that needs to appear between the `BeginPicture()` and `EndPicture()` calls is a call to the `BView` member function `DrawBitMap()`. Chapter 10, *Files*, discusses bitmaps.

Picture buttons are actually more versatile than described in this section. Here the picture button is treated as a one-state device—just as a standard push button is. The `BPictureButton` class can also be used, however, to create a picture button that is a two-state control. Setting the `behavior` parameter to the constant `B_TWO_STATE_BUTTON` tells the `BPictureButton` constructor to create a picture button that, when clicked on, toggles between the two pictures represented by the `BPicture` parameters `off` and `on`. Clicking on such a picture button displays one picture. Clicking on the button again displays the second picture. The displayed picture indicates to the user the current state of the button. To see a good real-world use of a two-state picture button, run the BeIDE. Then choose Find from the Edit menu. In the lower-left area of the Find window you'll find a button that has a picture of a small file icon on it. Click on the button and it will now have a picture of two small file icons on it. This button is used to toggle between two search options: search only the currently open, active file, and search all files present in the Find window list. Figure 6-5 shows both of this button's two states.

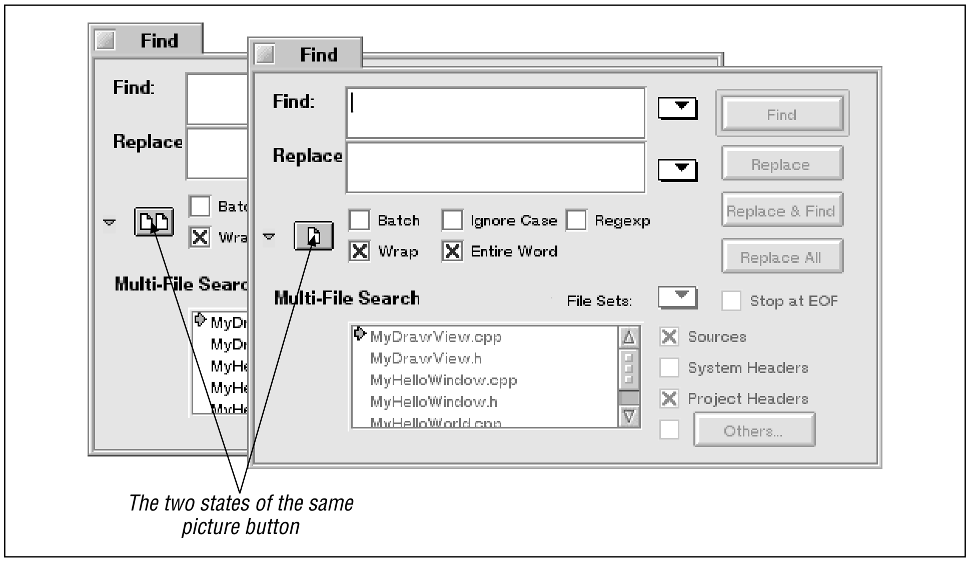


Figure 6-5. The Find window of the BeIDE provides an example of a picture button

Picture Button Example Project

The PictureButton project creates a program that displays a window that holds a single picture button. Figure 6-6 shows this one window under two different conditions. The leftmost window in the figure shows the button in its normal state. The rightmost window shows that when the button is clicked it gets slightly smaller and its center is filled in.

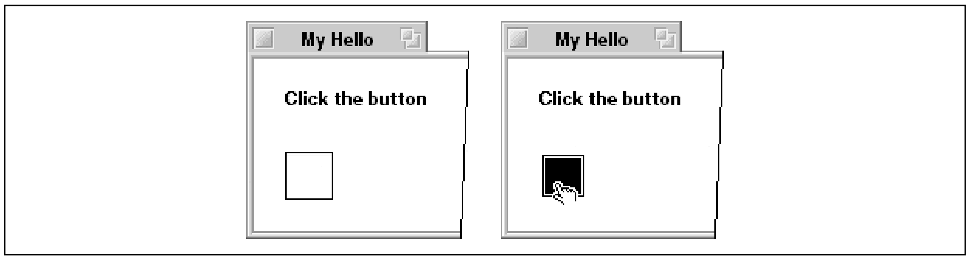


Figure 6-6. The window that results from running the PictureButton program

The picture button can include other pictures, which will be used if the program lets the button be disabled. Now that you know the basics of working with the `BPictureButton` class, the details of enhancing your picture buttons will be a quick read in the `BPictureButton` section of the Interface Kit chapter of the Be Book.

Preparing the window class for the picture button

This chapter's TwoButtons example project (presented in the "Buttons" section) provided a plan for adding a control, and support of that control, to a window. Here's how the window class header file (the *MyHelloWindow.h* file for this project) is set up for a new control:

- Define a constant to be used to represent an application-defined message type
- Override `MessageReceived()` in the window class declaration
- Add a control data member in the window class declaration

Here's how the `MyHelloWindow` class is affected by the addition of a picture button to a window of this class type:

```
#define   PICTURE_BEEP_MSG  'bep1'


class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();
      virtual void      MessageReceived(BMessage* message);

   private:
      MyDrawView        *fMyView;
      BPictureButton    *fPicButtonBeep;
};
```

I've defined the `PICTURE_BEEP_MSG` constant to have a value of `'bep1'`. Looking back at the TwoButtons example project, you'll see that this is the same value I gave to that project's `BUTTON_BEEP_1_MSG` constant. If both controls were present in the same application, I'd give one of these two constants a different value so that the `MessageReceived()` function could distinguish between a click on the Beep One push button and a click on the picture button.

Creating the picture button

The process of creating a control can also be expressed in a number of steps. All of the following affect the window source code file (the *MyHelloWindow.cpp* file in this particular example):

- Declare and assign values to the variables to be used in the control's constructor
- Create the control using `new` and the control's constructor
- Attach the control to the window by adding it to one of the window's views

Following the above steps to add a picture button to the `MyHelloWindow` constructor results in a new version of this routine that looks like this:

```
BRect        pictureBeep1Rect(20.0, 60.0, 50.0, 90.0);
const char   *pictureBeep1Name = "Beep1";


MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   BPicture  *buttonOffPict;
   BPicture  *buttonOnPict;

   fMyView->BeginPicture(new BPicture);
      BRect  offRect;

      offRect.Set(0.0, 0.0, 30.0, 30.0);
      fMyView->FillRect(offRect, B_SOLID_LOW);
      fMyView->StrokeRect(offRect, B_SOLID_HIGH);
   buttonOffPict = fMyView->EndPicture();

   fMyView->BeginPicture(new BPicture);
      BRect  onRect;

      onRect.Set(0.0, 0.0, 30.0, 30.0);
      fMyView->StrokeRect(onRect, B_SOLID_LOW);
      offRect.InsetBy(2.0, 2.0);
      fMyView->StrokeRect(onRect, B_SOLID_HIGH);
      onRect.InsetBy(2.0, 2.0);
      fMyView->FillRect(onRect, B_SOLID_HIGH);
   buttonOnPict = fMyView->EndPicture();

   fPicButtonBeep = new BPictureButton(pictureBeep1Rect, pictureBeep1Name,
                                    buttonOffPict, buttonOnPict,
                                    new BMessage(PICTURE_BEEP_MSG));
   fMyView->AddChild(fPicButtonBeep);

   Show();
}
```

The two `BPicture` objects are defined using a few of the basic drawing techniques covered in Chapter 5. As you read the following, refer back to the picture button in its off state (normal, or unclicked) and on state (being clicked) in Figure 6-5.

The off picture fills in a rectangle with the `B_SOLID_LOW` pattern (solid white) to erase the on picture that might currently be displayed (if the user has just clicked the picture button, the on picture will be serving as the picture button). Then a rectangle is outlined to serve as the off button.

The on picture erases the off picture (should it be currently drawn to the window as the picture button) by drawing a white (B_SOLID_LOW) rectangle outline with the boundaries of the off picture rectangle. That rectangle is then inset two pixels in each direction and a new rectangle is framed in black (B_SOLID_HIGH). The rectangle is then inset two more pixels, and this new area is filled with black.

Handling a picture button click

To handle a click on the picture button, MessageReceived() now looks for a message of type PICTURE_BEEP_MSG. Should that message reach the window, the computer sounds the system beep one time:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case PICTURE_BEEP_MSG:
         beep();
         break;

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

Checkboxes

The BCheckBox class is used to add checkboxes to a window. A BCheckBox object includes both the checkbox itself and a label, or title, to the right of the box. A checkbox is a two-state control: in the on state, the checkbox has an "X" in it; when off, it is empty. When a user clicks on a checkbox, its state is toggled. It's worthy of note that a checkbox label is considered a part of the checkbox control. That means that a user's click on the checkbox itself or anywhere on the checkbox label will toggle the checkbox to its opposite state.

Whether a click results in a checkbox being turned on (checked) or off (unchecked), a message is sent to the window that holds the checkbox. While a program can immediately respond to a click on a checkbox, it is more typical for the program to wait until some other action takes place before responding. For instance, the setting of some program feature could be done via a checkbox. Clicking the checkbox wouldn't, however, immediately change the setting. Instead, when the user dismisses the window the checkbox resides in, the value of the checkbox can be queried and the setting of the program feature could be performed at that time.

Creating a Checkbox

The BCheckBox constructor has six parameters:

```
BCheckBox(BRect       frame,
          const char  *name,
          const char  *label,
          BMessage    *message,
          uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
          uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The BCheckBox parameters match those used in the BButton constructor—if you know how to create a button, you know how to create a checkbox. Adding to the similarities is that after you attach the checkbox to a window, the control's height will be automatically adjusted to accommodate the height of the text of the control's label. If the values of the frame rectangle coordinates don't produce a rectangle with a height sufficient to display the checkbox label, the BCheckBox constructor will increase the checkbox boundary rectangle height by increasing the value of the frame rectangle's bottom value. The exact height of the checkbox depends on the font in which the control's label is displayed.

As for other control types, you'll define a message constant that is to be paired with the control. For instance:

```
#define   CHECKBOX_MSG   'ckbx'
```

Then, optionally, add a data member of the control type to the class declaration of the window type the control is to be added to:

```
class MyWindow : public BWindow {
   ...
   private:
      BView     *fView;
      BButton   *fCheckBox;
}
```

The following snippet is typical of the code you'll write to create a new checkbox and add that control to a view:

```
BRect   checkBoxRect(20.0, 20.0, 100.0, 50.0);

fCheckBox = new BCheckBox(checkBoxRect,"MyCheckbox"
                          "Check Me", new BMessage(CHECKBOX_MSG));

fMyView->AddChild(fCheckBox);
```

Checkbox (Action Now) Example Project

Clicking a checkbox may have an immediate effect on some aspect of the program, or it may not have an impact on the program until the user confirms the checkbox selection—usually by a click on a button. The former use of a checkbox is demonstrated in the example project described here: CheckBoxNow. For an example of the other usage, a checkbox that has an effect after another action is taken, look over the next example, the CheckBoxLater project.

The use of a checkbox to initiate an immediate action is often in practice when some area of the window the checkbox resides in is to be altered. For instance, if some controls in a window are to be rendered unusable in certain conditions, a checkbox can be used to disable (and then later enable) these controls. This is how the checkbox in the CheckBoxNow example works. The CheckBoxNow project creates a program with a window that holds two controls: a button and a checkbox. When the program launches, both controls are enabled, and the checkbox is unchecked—as shown in the top window in Figure 6-7. As expected, clicking on the Beep One button produces a single system beep. Clicking on the checkbox disables beeping by disabling the button. The bottom window in Figure 6-7 shows how the program's one window looks after clicking the Disable Beeping checkbox.

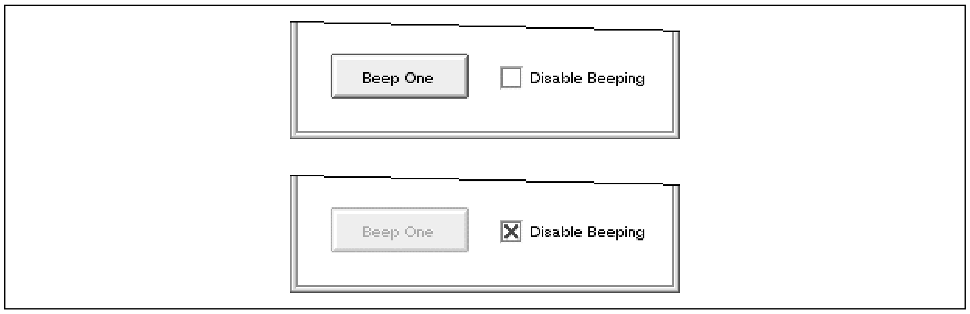


Figure 6-7. The windows that result from running the CheckBoxNow program

Preparing the Window class for the checkbox

The *MyHelloWindow.h* file prepares for the window's support of a button and a checkbox by defining a constant for each control's message:

```
#define   BUTTON_BEEP_1_MSG      'bep1'
#define   CHECKBOX_SET_BEEP_MSG  'stbp'
```

The `MyHelloWindow` class now holds three data members:

```
class MyHelloWindow : public BWindow {

   public:
                      MyHelloWindow(BRect frame);
      virtual bool    QuitRequested();
      virtual void    MessageReceived(BMessage* message);

   private:
      MyDrawView      *fMyView;
```

```
        BButton         *fButtonBeep1;
        BCheckBox       *fCheckBoxSetBeep;
};
```

Creating the checkbox

I've declared and initialized the button and checkbox boundary rectangles near one another so that I could line them up—Figure 6-6 shows that the checkbox is just to the right of the button and centered vertically with the button.

```
BRect         buttonBeep1Rect(20.0, 60.0, 110.0, 90.0);
BRect         checkBoxSetBeepRect(130.0, 67.0, 230.0, 90.0);
const char    *buttonBeep1Name      = "Beep1";
const char    *checkBoxSetBeepName  = "SetBeep";
const char    *buttonBeep1Label     = "Beep One";
const char    *checkBoxSetBeepLabel = "Disable Beeping";
```

The `MyHelloWindow` constructor creates both the button and checkbox:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtonBeep1 = new BButton(buttonBeep1Rect, buttonBeep1Name,
                              buttonBeep1Label,
                              new BMessage(BUTTON_BEEP_1_MSG));

   fMyView->AddChild(fButtonBeep1);

   fCheckBoxSetBeep = new BCheckBox(checkBoxSetBeepRect, checkBoxSetBeepName,
                                    checkBoxSetBeepLabel,
                                    new BMessage(CHECKBOX_SET_BEEP_MSG));

   fMyView->AddChild(fCheckBoxSetBeep);

   Show();
}
```

Handling a checkbox click

When the checkbox is clicked, the system will toggle it to its opposite state and then send a message of the application-defined type `CHECKBOX_SET_BEEP_MSG` to the `MyHelloWindow` `MessageReceived()` routine. In response, this message's case section obtains the new state of the checkbox and enables or disables the Beep One button as appropriate. If the Disable Beeping checkbox is checked, or on, the button is disabled by passing a value of `false` to the button's

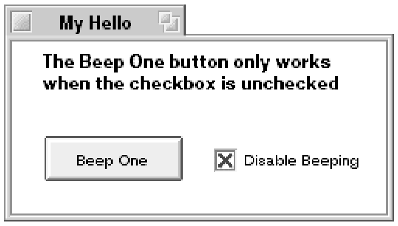
`SetEnabled()` routine. If the checkbox is unchecked, or off, a value of `true` is passed to this same routine in order to enable the button:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_1_MSG:
         beep();
         break;

      case CHECKBOX_SET_BEEP_MSG:
         int32  checkBoxState;

         checkBoxState = fCheckBoxSetBeep->Value();

         if (checkBoxState == B_CONTROL_ON)
            // Disable Beeping checkbox is checked, deactivate beep button
            fButtonBeep1->SetEnabled(false);
         else
            // Disable Beeping checkbox is unchecked, activate beep button
            fButtonBeep1->SetEnabled(true);
         break;

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

Checkbox (Action Later) Example Project

The CheckBoxNow project responds immediately to a click on a checkbox. More often, programs let users check or uncheck the checkboxes without any immediate effect. Thus, your program might reserve action until the choice is confirmed by the user's click on a button (such as OK, Done, or Accept). The CheckBox-Later project demonstrates this approach. Figure 6-7 shows that the CheckBox-Later program displays a window that looks similar to that displayed by the CheckBoxNow program. The program differs in when the state of the checkbox is queried by the program. In the CheckBoxLater program, clicking the Disable Beeping checkbox any number of times has no immediate effect on the Beep One button (in Figure 6-8, you see that the checkbox is checked, yet the button isn't disabled). It's only when the user clicks the Beep One button that the program checks to see if the Disable Beeping checkbox is checked. If it isn't checked, the button click plays the system beep. If it is checked, the button can still be clicked, but no sound will be played.

The only changes that were made to the CheckBoxNow code to turn it into the code for the CheckBoxLater project are in the `MessageReceived()` function. Here's how the new version of that routine looks:



The Beep One button only works
when the checkbox is unchecked

Figure 6-8. The window that results from running the CheckBoxLater program

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_1_MSG:
         int32  checkBoxState;

         checkBoxState = fCheckBoxSetBeep->Value();

         if (checkBoxState == B_CONTROL_ON)
            // Disable Beeping checkbox is checked, meaning DON'T beep
            ;
         else
            // Disable Beeping checkbox is unchecked, meaning DO beep
            beep();

         break;

      case CHECKBOX_SET_BEEP_MSG:
         // Here we don't do anything. Instead, we wait until the user
         // performs some other action before checking the value of the
         // checkbox break;

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

In `MessageReceived()`, the body of the `CHECKBOX_SET_BEEP_MSG` case section performs no action—a message of this type is now essentially ignored. The program would run the same even if this case section was removed, but I've left the `CHECKBOX_SET_BEEP_MSG` case label in the switch so that it's evident that `MessageReceived()` is still the recipient of such messages.

Radio Buttons

A radio button is similar to a checkbox in that it is a two-state control. Unlike a checkbox, though, a radio button always appears grouped with at least one other control of its kind.

For any given radio button group, no more than one radio button can be on at any time. When the user clicks on one button in a group, the button that was on at the time of the click is turned off and the newly clicked button is turned on. A radio button group is responsible for updating the state of its buttons—your code won't need to turn them on and off.

Creating a Radio Button

The `BRadioButton` constructor has the same six parameters described back in this chapter's "The BControl class" section:

```
BRadioButton(BRect       frame,
             const char  *name,
             const char  *label,
             BMessage    *message,
             uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
             uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

Like other types of controls, a radio button's height will be adjusted if the height specified in the frame rectangle isn't enough to accommodate the font being used to display the radio button's label.

Before creating a new radio button, define a constant to be used as the control's message type. Here's an example from a project that has two radio buttons in a window:

```
#define   RADIO_1_MSG   'rad1'
#define   RADIO_2_MSG   'rad2'
```

Access to a radio button is easiest if a data member of the control type is added to the class declaration of the window type the control is to be added to:

```
class MyWindow : public BWindow {
   ...
   private:
      BView         *fView;
      BRadioButton  *fRadio1;
      BRadioButton  *fRadio2;
}
```

The following snippet shows the creation of two radio buttons, each of which is added to the same view:

```
BRect   radio1Rect(20.0, 20.0, 100.0, 49.0);
BRect   radio2Rect(20.0, 50.0, 100.0, 79.0);

fRadio1 = new BRadioButton(radio1Rect, "MyRadio1",
                           "One", new BMessage(RADIO_1_MSG));
fMyView->AddChild(fRadio1);
```

```
    fRadio2 = new BRadioButton(radio2Rect, "MyRadio2",
                               "Two", new BMessage(RADIO_2_MSG));
    fMyView->AddChild(fRadio2);
```

By adding radio buttons to the same view, you designate that the buttons be considered a part of a radio button group. The simple act of placing a number of buttons in the same view is enough to have these buttons act in unison. A click on one radio button turns that button on, but not until that button turns off all other radio buttons in the same view.

A single window can have any number of radio button groups, or sets. For instance, a window might have one group of three buttons that provides the user with the option of displaying graphic images in monochrome, grayscale, or color. This same window could also have a radio button group that provides the user with a choice of four filters to apply to the image. In such a scenario, the window would need to include a minimum of two views—one for the group of three color-level radio buttons and another for the group of four filter radio buttons.

Radio Buttons Example Project

The RadioButtonGroup project demonstrates how to create a group of radio buttons. As shown in Figure 6-9, the RadioButtonGroup program's window includes a group of three radio buttons that allow the user to alter the behavior of the Beep push button.

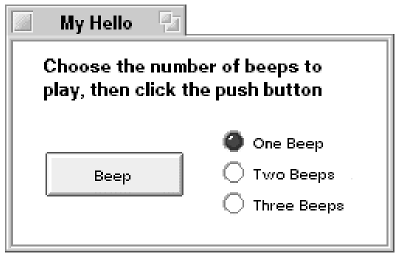

My Hello
Choose the number of beeps to play, then click the push button
Beep
One Beep
Two Beeps
Three Beeps

Figure 6-9. The window that results from running the RadioButtonGroup program

Preparing the window class for the radio buttons

The *MyHelloWindow.h* header file includes four control message constants—one per control. The push button constant has been given the value `'bEEp'` just to illustrate that an application-defined message constant can include uppercase characters (to avoid conflicting with Be-defined control message constants, it just shouldn't consist of all uppercase characters).

```
    #define   BUTTON_BEEP_1_MSG   'bEEp'
    #define   RADIO_BEEP_1_MSG    'bep1'
```

```
#define   RADIO_BEEP_2_MSG    'bep2'
#define   RADIO_BEEP_3_MSG    'bep3'
```

This project's version of the `MyHelloWindow` class includes six data members: one to keep track of the window's view, one to keep track of each of the window's four controls, and one to keep track of the number of beeps to play when the push button is clicked:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BButton        *fButtonBeep1;
      BRadioButton   *fRadioBeep1;
      BRadioButton   *fRadioBeep2;
      BRadioButton   *fRadioBeep3;
      int32          fNumBeeps;
};
```

Laying out the radio buttons

A number of radio buttons are defined as a group when they all reside in the same view. A `MyHelloWindow` object includes a view (`MyDrawView`) that occupies its entire content area—so I could add the three radio buttons to this view and have them automatically become a radio button group. That, however, isn't what I'm about to do. Instead, I'll create a new view and add it to the existing view. If at a later time I want to add a second group of radio buttons (to control some other, unrelated, option) to the window, the buttons that will comprise that group will need to be in a new view—otherwise they'll just be absorbed into the existing radio button group. By creating a new view that exists just for one group of radio buttons, I'm getting in the habit of setting up a radio button group as an isolated entity.

Placing a radio button group in its own new view also proves beneficial if it becomes necessary to make a change to the layout of all of the group's radio buttons. For instance, if I want to relocate all of the buttons in a group to another area of the window, I just redefine the group's view rectangle rather than redefining each of the individual radio button boundary rectangles. When the view moves, so do the radio buttons in it. Or, consider that I may, for aesthetic reasons, want to outline the area that holds the radio buttons. I can easily do so by framing the radio button group view.

The following variable declarations appear near the top of the *MyHelloWindow.cpp* file. Note that the boundary rectangle for each of the three radio buttons has a left coordinate of 10.0, yet the radio buttons are certainly more than 10 pixels in from the left side of the window.

Keep in mind that after being created, the radio buttons will be added to a new `BView` that is positioned in the window based on the `radioGroupRect` rectangle. Thus, the coordinate values of the radio button rectangles are relative to the new `BView`. Figure 6-10 clarifies my point by showing where the radio group view will be placed in the window. In that figure, the values 125 and 50 come from the left and top values in the `radioGroupRect` rectangle declared here:

```
BRect         radioGroupRect(125.0, 50.0, 230.0, 120.0);

BRect         radioBeep1Rect(10.0,  5.0, 90.0, 25.0);
BRect         radioBeep2Rect(10.0, 26.0, 90.0, 45.0);
BRect         radioBeep3Rect(10.0, 46.0, 90.0, 65.0);
const char    *radioBeep1Name  = "Beep1Radio";
const char    *radioBeep2Name  = "Beep2Radio";
const char    *radioBeep3Name  = "Beep3Radio";
const char    *radioBeep1Label = "One Beep";
const char    *radioBeep2Label = "Two Beeps";
const char    *radioBeep3Label = "Three Beeps";
```

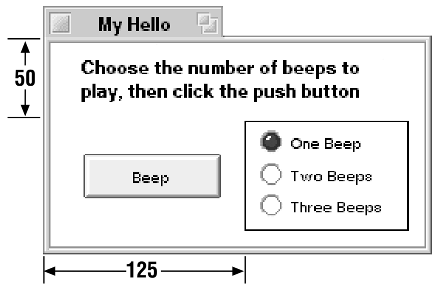

My Hello
50
Choose the number of beeps to play, then click the push button
Beep
One Beep
Two Beeps
Three Beeps
125

Figure 6-10. A group of radio buttons can reside in their own view

Creating the radio buttons

The three radio buttons are created, as expected, in the `MyHelloWindow` constructor. Before doing that, a generic view (a view of the Be class `BView`) to which the radio buttons will be added is created and added to the view of type `MyDrawView`. By default, each new radio button is turned off. A group of radio buttons must always have one button on, so after the three radio buttons are created, one of them (arbitrarily, the One Beep button) is turned on by calling the button's `SetValue()` member function. The data member `fNumBeeps` is then initialized to a value that matches the number of beeps indicated by the turned-on radio button:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtonBeep1 = new BButton(buttonBeep1Rect, buttonBeep1Name,
                              buttonBeep1Label,
                              new BMessage(BUTTON_BEEP_1_MSG));

   fMyView->AddChild(fButtonBeep1);

   BView  *radioGroupView;

   radioGroupView = new BView(radioGroupRect, "RadioView",
                              B_FOLLOW_ALL, B_WILL_DRAW);
   fMyView->AddChild(radioGroupView);

   fRadioBeep1 = new BRadioButton(radioBeep1Rect, radioBeep1Name,
                                  radioBeep1Label,
                                  new BMessage(RADIO_BEEP_1_MSG));

   radioGroupView->AddChild(fRadioBeep1);

   fRadioBeep2 = new BRadioButton(radioBeep2Rect, radioBeep2Name,
                                  radioBeep2Label,
                                  new BMessage(RADIO_BEEP_2_MSG));

   radioGroupView->AddChild(fRadioBeep2);

   fRadioBeep3 = new BRadioButton(radioBeep3Rect, radioBeep3Name,
                                  radioBeep3Label,
                                  new BMessage(RADIO_BEEP_3_MSG));

   radioGroupView->AddChild(fRadioBeep3);

   fRadioBeep1->SetValue(B_CONTROL_ON);
   fNumBeeps = 1;

   Show();
}
```

Handling a radio button click

When a radio button is clicked, a message of the appropriate application-defined type reaches the window's `MessageReceived()` function. The clicking of a radio button, like the clicking of a checkbox, typically doesn't cause an immediate action to occur. Such is the case in this example. `MessageReceived()` handles the button click by simply setting the `MyHelloWindow` data member `fNumBeeps` to the value indicated by the clicked-on radio button. When the user eventually clicks on the Beep push button, `beep()` is invoked the appropriate number of times:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   bigtime_t  microseconds = 1000000;  // one second

   switch(message->what)
   {
      case RADIO_BEEP_1_MSG:
         fNumBeeps = 1;
         break;

      case RADIO_BEEP_2_MSG:
         fNumBeeps = 2;
         break;

      case RADIO_BEEP_3_MSG:
         fNumBeeps = 3;
         break;

      case BUTTON_BEEP_MSG:
         int32  i;

         for (i = 1; i <= fNumBeeps; i++) {
            beep();
            if (i != fNumBeeps)
               snooze(microseconds);
         }
         break;

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

View Hierarchy and Controls

Chapter 4 introduced the concept of the window view hierarchy—the organization of views within a window. In this chapter's most recent example you've just seen a window that included a number of views (keeping in mind that a control is a type of view). Now that you've encountered the first example that includes several views, this a good time to revisit the topic of the view hierarchy in order to fill in some of the details. Figure 6-11 shows the view hierarchy for a window—an object of the `MyHelloWindow` class—from the RadioButtonGroup program.

Adding views to the hierarchy

A window's top view is always a "built-in" part of the window—you don't explicitly add the top view as you add other views. The `BView`-derived `fMyView` view lies directly below the top view, telling you that this view has been added to the window. The `BButton` `fButtonBeep1` view and the `BView` `radioGroupView` lie directly below the `fMyView` view, so you know that each has been added to

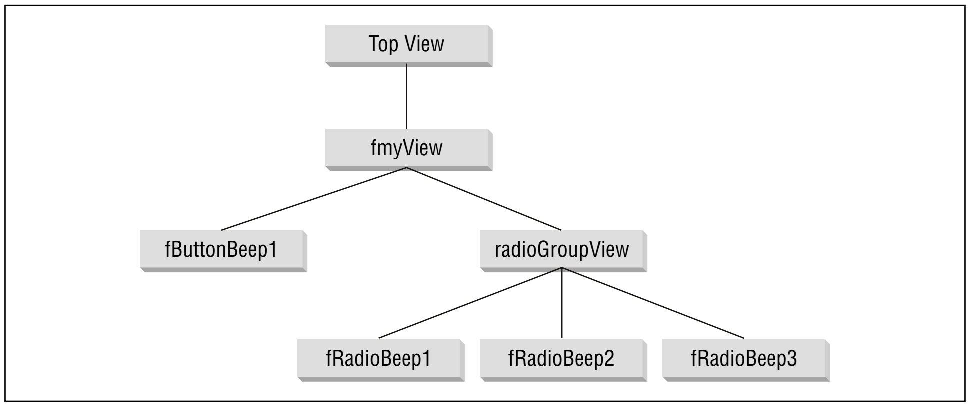


Figure 6-11. The view hierarchy for the RadioButtonGroup window

`fMyView`. Finally, the three `BRadioButton` views are beneath `radioGroupView`, telling you that these three views have been added to `radioGroupView`. You can confirm this by looking at the six `AddChild()` calls in the `MyHelloWindow` constructor—they indicate which parent view each view was added to:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   ...
   AddChild(fMyView);
   ...
   fMyView->AddChild(fButtonBeep1);
   ...
   fMyView->AddChild(radioGroupView);
   ...
   radioGroupView->AddChild(fRadioBeep1);
   ...
   radioGroupView->AddChild(fRadioBeep2);
   ...
   radioGroupView->AddChild(fRadioBeep3);
   ...
}
```

Accessing views

The names of the views in Figure 6-12 tell you how each view is referenced.

You don't draw to the top view—it merely serves as a container for organizing other views. To reference this view (as when adding a view to the window), reference the window itself. In the window's constructor, just call the desired `BWindow` member function, as in:

```
AddChild(fMyView);
```

From outside a window member function, use the `fMyWindow` data member from the `MyHelloApplication` object. If the `fMyView` view was to be added in the `BApplication` constructor after the window was created, the call to `AddChild()` would look like this:

```
fMyWindow->AddChild(fMyView);
```

The Be naming convention states that the name of a class data member should start with a lowercase "f" character. In Figure 6-11 you see that five of the six views below the top view are referenced by data members. To work with any one of these views, use the data member that references it. For instance, to invoke the `BView` member function `FillRect()` to fill a rectangle in the window's `fMyView` view, just call the routine like this:

```
fMyView->FillRect(aRect);
```

I haven't kept a data member reference in the `MyHelloWindow` class to the `BView` that groups the three radio buttons. If it became necessary to reference this view, you could call the `BView` member function `FindView()` to locate the view object and return a reference to it. Recall that when a view is created, you give it a name. For example, the radio group view was given the name "RadioView":

```
radioGroupView = new BView(radioGroupRect, "RadioView", B_FOLLOW_ALL,
                           B_WILL_DRAW);
fMyView->AddChild(radioGroupView);
```

You can find the view at any time by calling `FindView()` from the parent view. For instance, to fill a rectangle in the radio group view, call `FindView()` from that view's parent view, `fMyView`:

```
BView  *aView;

aView = fMyView->FindView("RadioView");
aView->FillRect(aRect);
```

Keep in mind that there are always two, and may be three, references, associated with one view. When a view is created, the view object is returned to the program and referenced by a variable (such as `radioGroupView` in the preceding example). When invoking the `BView` (or `BView`-derived) class constructor, a name for the view is supplied in quotes (as in "RadioView" in the preceding example). Finally, some views have a label—a name that is displayed on the view itself (as in a control such as a `BButton`—the button displays a name such as OK or Beep).

View Updating

In the RadioButtonGroup project, the `MyHelloWindow` constructor creates a view referenced by the `MyHelloWindow` data member `fMyView` and a view referenced by the local `BView` variable `radioGroupView`. This one `MyHelloWindow` constructor shows two ways of working with views, so it will be worth our while to again sidetrack from the discussion of controls in order to gain a better understanding of the very important topic of views.

BView-derived classes and the generic BView class

In the RadioButtonGroup project, and several projects preceding it, I've opted to fill the content area of a window with a view of the application-defined `BView`-derived class `MyDrawView`. One of the chief reasons for defining such a class is to let the system become responsible for updating a view. This is accomplished by having my own class override the `BView` member function `Draw()`.

A second way to work with a view is to not define a view class, but instead simply create a generic `BView` object within an application-derived routine. That's what the RadioButtonGroup project does in the `MyHelloWindow` constructor:

```
radioGroupView = new BView(radioGroupRect, "RadioView", B_FOLLOW_ALL,
                           B_WILL_DRAW);
```

After you attach the new view to an existing view, drawing can take place in the new view. For instance, if in addition to beeping, you want the program to draw a border around the three radio buttons in response to a click on the window's one push button, add the following code under the `BUTTON_BEEP_MSG case` label in the `MessageReceived()` function:

```
BView  *radioView;
BRect  radioFrame;

radioView = fMyView->FindView("RadioView");
radioFrame = radioView->Bounds();
radioView->StrokeRect(radioFrame);
```

Superficially, this approach of creating a generic `BView` and then drawing in it is simpler than defining a `BView`-derived class and then implementing a `Draw()` function for that class. But in taking this easier approach, you lose the benefit of having the system take responsibility for updating the view. Consider the above snippet. That code will nicely frame the three radio buttons. But if the window that holds the buttons ever needs updating (and if the user is allowed to move the window, of course at some point it will), the frame that surrounds the buttons won't be redrawn. The system will indeed invoke a `Draw()` function for `radioGroupView`, but it will be the empty `BView` version of `Draw()`.

Implementing a BView-derived class in the RadioButtonGroup project

What if I do want my RadioButtonGroup program to frame the radio buttons, and to do so in a way that automatically updates the frame as needed? Instead of placing the radio buttons in a generic `BView` object, I can define a new `BView`-derived class just for this purpose:

```
class MyRadioView : public BView {

   public:
                     MyRadioView(BRect frame, char *name);
      virtual void   Draw(BRect updateRect);
};
```

The `MyRadioView` constructor can be empty—just as the `MyDrawView` constructor is:

```
MyRadioView::MyRadioView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

The `MyRadioView` version of the `Draw()` function is quite simple as well:

```
void MyRadioView::Draw(BRect)
{
   BRect frame = Bounds();

   StrokeRect(frame);
}
```

While I could keep track of an instance of the `MyRadioView` class by calling the parent view's `FindView()` function as needed, I'd opt for the method of storing a reference to the view in the window that will hold the view. Here I've added such a reference to the six existing data members in the `MyHelloWindow` class of the RadioButtonGroup project:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BButton        *fButtonBeep1;
      BRadioButton   *fRadioBeep1;
      BRadioButton   *fRadioBeep2;
      BRadioButton   *fRadioBeep3;
      int32          fNumBeeps;
      MyRadioView    *fMyRadioView;
};
```

To create a MyRadioView object, I replace the generic BView creation code in the MyHelloWindow constructor with the following:

```
fMyRadioView = new MyRadioView(radioGroupRect, "RadioView");
fMyView->AddChild(fMyRadioView);
```

Now, when a radio button is created, I add it to the MyRadioView object fMyRadioView, like this:

```
fRadioBeep1 = new BRadioButton(radioBeep1Rect, radioBeep1Name,
                               radioBeep1Label,
                               new BMessage(RADIO_BEEP_1_MSG));

fMyRadioView ->AddChild(fRadioBeep1);
```

Thanks to the Draw() function of the MyRadioView class, the new radio button group will have a border drawn around it, exactly as was shown back in Figure 6-10. Better yet, obscuring the window and then bringing it back to the forefront doesn't cause the border to disappear—the update message that the Application Server sends to the MyHelloWindow window results in the calling of the Draw() function of each "out-of-date" view in the window.

If you want to see all of the code for this new version of the RadioButtonGroup program, you'll find it in the RadioButtonGroupFrame project.

If you've followed this discussion, you should be able to quickly answer the following question: in the original RadioButtonGroup project, why didn't I create a new BView-derived class like the MyRadioView class and use an object of that type to hold the radio buttons? Answer: because I use the radio button view only as a means to group the radio buttons together—I don't draw to the view. The simple approach of creating a BView on the fly works for that purpose.

Text Fields

The BTextControl class is used to add a text field to a window. A text field consists of both a static, uneditable label and an editable field that allows the user to enter a single line of text. The label appears to the left of the editable field, and is generally used to provide the user with an idea of what to enter in the editable field ("Enter your age in years:" is an example).

A text field is often handled like a checkbox or radio button: no immediate action is taken by the program in response to the user's action. Typically, the program acquires the text in the text field only when a button labeled OK, Accept, Save, or something similar is clicked on.

If your program needs to get or set the editable text of a text field as soon as the user has finished typing, the `BTextControl` accommodates you. Like other controls, a text field issues a message that will be received by the `MessageReceived()` function of the control's window. Such a message is sent when the control determines that the user has finished entering text in the editable field, as indicated by a press of the Enter, Return, or Tab key or a mouse button click in the editable field of a different text field control. In all of these instances, the text that was previously in the editable field must have been modified in order for the control to send the message. If the user, say, clicks in an editable field of a text field, then presses the Enter key, no message will be sent.

Creating a Text Field

The `BTextControl` constructor has six parameters common to all controls, along with a `text` parameter that specifies a string that is to initially appear in the editable field of the text field control:

```
BTextControl(BRect       frame,
             const char  *name,
             const char  *label,
             const char  *text,
             BMessage    *message,
             uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
             uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

Passing a string in the `text` parameter of the `BTextControl` constructor is useful if you want to alert the user that a default string or value is to be used in the event that the user doesn't enter a string or value. If a value of `NULL` is passed as the `text` parameter, no text initially appears in the editable field.

As with all control types, you must define a unique message constant that will be paired with the control. To assist in keeping track of the text field control, you can optionally include a data member in the window class in which the control is to reside:

```
#define   TEXT_FIELD_MSG   'txtf'

class MyWindow : public BWindow {
   ...
   private:
      BView         *fView;
      BTextControl  *fTextField;
}
```

To create the control, pass the `BTextControl` constructor a boundary rectangle, name, static text label, initial editable text, and a new `BMessage` object. Then add the new text field to the view the control will reside in:

```
BRect   textFieldRect(20.0, 20.0, 120.0, 50.0);

fTextField = new BTextControl(textFieldRect, "TextField",
                              "Number of dependents:", "0",
                              new BMessage(TEXT_FIELD_MSG));

fMyView->AddChild(fTextField);
```

Getting and Setting the Text

After creating a text field control, your program can obtain or set the editable field text at any time. To obtain the text currently in the text field, invoke the `BTextControl` member function `Text()`. Here, the text in a control is returned to the program and saved to a string named `textFieldText`:

```
char *textFieldText;

textFieldText = fTextField->Text();
```

The text that appears in the editable field of the control is initially set in the `BTextControl` constructor, and is then edited by the user. The contents of the editable field can also be set at any time by your program by invoking the `BTextControl` member function `SetText()`. This routine overwrites the current contents of the editable field with the string that was passed to `SetText()`. Here the current contents of a control's editable field are obtained and checked for validity. If the user-entered string isn't consistent with what's expected, the string "Invalid Entry" is written in place of the incorrect text the user entered:

```
char  *textFieldText;
bool  textValid;

textFieldText = fTextField->Text();
...
// check for validity of user-entered text that's now held in textFieldText
...
if (!textValid)
   fTextField->SetText("Invalid Entry");
```

Reproportioning the Static Text and Editable Text Areas

The `label` parameter specifies the static text that is to appear to the left of the editable field. If `NULL` is passed here, all of the control's boundary rectangle (as defined by the frame parameter) is devoted to the text field. Any `label` string other than `NULL` tells the constructor to devote half the width of the `frame` rectangle to the static text label and the other half of this rectangle to the editable text area. Consider this snippet:

```
BRect textFieldRect(20.0, 60.0, 220.0, 90.0);

fTextField = new BTextControl(textFieldRect, "TextField",
                              "Name:", "George Washington Carver",
                              new BMessage(TEXT_FIELD_MSG));

fMyView->AddChild(fTextField);
```

In this code, a text field control with a width of 200 pixels is created. By default, the static text field and the editable text field of the control each have a width that is one-half of the control's boundary rectangle, or 100 pixels. The result is shown in the top window in Figure 6-12. Because the label is a short string, and because the value that might be entered in the editable text field may be a long string, it would make sense and be more aesthetically pleasing to change the proportions of the two text areas. Instead of devoting 100 pixels to the static text label "Name:", it would be better to give that text just, say, 35 pixels of the 200 pixels that make up the control's width. Such reproportioning is possible using the `BTextControl` member function `SetDivider()`.

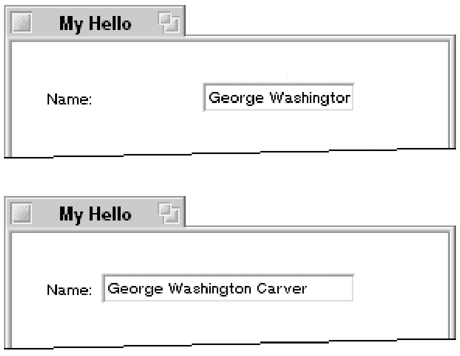


Figure 6-12. The two areas of a text field control can be proportioned in different ways

When passed a floating-point value, `SetDivider()` re-establishes the dividing point between the two text areas of a text field control. `SetDivider()` uses the `BTextControl` object's local coordinate system (meaning that the left edge of the text edit control has a value of 0.0, regardless of where the control is positioned in a window). The following addition to the above snippet changes the ratio to 35 pixels for the static text field and 165 pixels for the editable text field. The bottom window in Figure 6-12 shows the result. Notice that the placement and overall width of the control are unaffected by the call to `SetDivider()`.

```
float  xCoordinate = 35.0;

fTextField->SetDivider(xCoordinate);
```

Text Field Example Project

The TextField project demonstrates how to include a text field in a window, obtain that control's user-entered string, and make use of that string elsewhere. When the user clicks the window's button, the program gets the string from the text field and uses that string as the new label for the push button. Figure 6-13 shows the program's window after the button has been clicked.

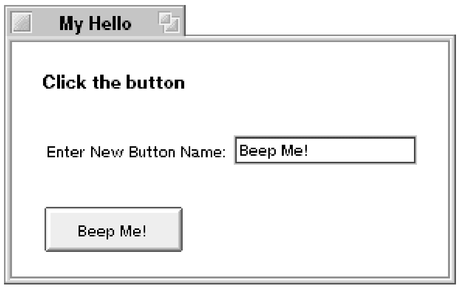


Figure 6-13. The window that results from running the TextField program

Preparing the window class for the text field

The *MyHelloWindow.h* header file is edited to include a control message constant for the window's two controls.

```
#define   BUTTON_BEEP_1_MSG   'bep1'
#define   TEXT_NEW_TITLE_MSG  'newT'
```

To keep track of the window's views, the `MyHelloWindow` class now holds three data members:

```
class MyHelloWindow : public BWindow {

   public:
                       MyHelloWindow(BRect frame);
      virtual bool     QuitRequested();
      virtual void     MessageReceived(BMessage* message);

   private:
      MyDrawView       *fMyView;
      BButton          *fButtonBeep1;
      BTextControl     *fTextNewTitle;
};
```

Creating the text field

The text field and button information are defined together just before the implementation of the `MyHelloWindow` constructor. The push button label will initially be "Beep One" and the text that will appear initially in the editable field of the text field control is the string "Beep Me!":

```
BRect        buttonBeep1Rect(20.0, 105.0, 110.0, 135.0);
BRect        textNewTitleRect(20.0, 60.0, 260.0, 90.0);
const char   *buttonBeep1Name   = "Beep1";
const char   *textNewTitleName  = "TextTitle";
const char   *buttonBeep1Label  = "Beep One";
const char   *textNewTitleLabel = "Enter New Button Name:";
const char   *textNewTitleText  = "Beep Me!";
```

The `MyHelloWindow` constructor creates the window's main view, then creates and adds the button control and text field control to that view:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_RESIZABLE)
{
   frame.OffsetTo(B_ORIGIN);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtonBeep1 = new BButton(buttonBeep1Rect, buttonBeep1Name,
                              buttonBeep1Label,
                              new BMessage(BUTTON_BEEP_1_MSG));

   fMyView->AddChild(fButtonBeep1);

   fTextNewTitle = new BTextControl(textNewTitleRect, textNewTitleName,
                                    textNewTitleLabel, textNewTitleText,
                                    new BMessage(TEXT_NEW_TITLE_MSG));

   fMyView->AddChild(fTextNewTitle);

   Show();
}
```

Handling a text field entry and a button click

A program can make use of a text field control in two ways. First, a window can obtain the user-entered text from a text field control at any time—without regard for whether the text field has issued a message. Second, a window can await a message sent by the text field and then respond. The `MessageReceived()` function demonstrates both these options.

When the window's push button is clicked, the window receives a message from the button. At that time, the editable text of the text field control is retrieved and used in a call to the button's `SetLabel()` function. While the retrieving of the text field control's editable text takes place in response to a message, that message is one issued by the push button—not the text field control.

When the user clicks in the text field, that control becomes the focus view. Recall from Chapter 4 that a window can have only one focus view, and that view becomes the recipient of keystrokes. Once a text field is the focus view (as indicated by the editable text field being highlighted and the I-beam appearing in it), and once that control's editable text has been altered in any way, the control is capable of sending a message. That happens when the user presses the Return, Enter, or Tab key. In response to a message sent by the text field control, `MessageReceived()` resets the push button's label to its initial title of "Beep One" (as defined by the `buttonBeep1Label` constant):

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_1_MSG:
         char *textFieldText;

         textFieldText = fTextNewTitle->Text();
         fButtonBeep1->SetLabel(textFieldText);
         beep();
         break;

      case TEXT_NEW_TITLE_MSG:
         fButtonBeep1->SetLabel(buttonBeep1Label);
         break;

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

Multiple Control Example Project

The numerous example projects in this chapter demonstrated how to incorporate one, or perhaps two, types of controls in a window. Your real-world program might very well include numerous controls. ControlDemo is such a program—its one window holds the six controls shown in Figure 6-14.

To use ControlDemo, enter a number in the range of 1 to 9 in the text field control, click a radio button control to choose one of three drawing colors, then click the Draw button. The ControlDemo program responds by drawing colored, overlapping circles. The number of circles drawn is determined by the value entered in the text field. Before drawing the circles, ControlDemo erases the drawing area—so you can try as many combinations of circles and colors as you want. You can

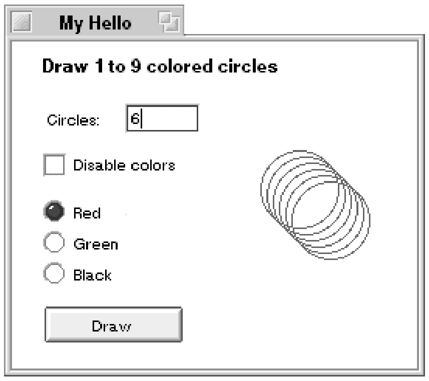


Figure 6-14. The window that results from running the ControlDemo program

also click the Disable colors checkbox to disable the radio buttons and force drawing to take place in the last selected color.

Preparing the Window Class for the Controls

The program's window holds six controls, so you can expect to see six application-defined message constants in the *MyHelloWindow.h* file:

```
#define   BUTTON_DRAW_MSG         'draw'
#define   RADIO_RED_MSG           'rred'
#define   RADIO_GREEN_MSG         'rgrn'
#define   RADIO_BLACK_MSG         'rblk'
#define   TEXT_NUM_CIRCLES_MSG    'crcl'
#define   CHECKBOX_SET_COLOR_MSG  'setc'
```

Each control is kept track of by a data member in the `MyHelloWindow` class. There's also the familiar data member that exists to keep track of the window's main view:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BTextControl   *fTextNumCircles;
      BCheckBox      *fCheckBoxSetColor;
      BRadioButton   *fRadioRed;
      BRadioButton   *fRadioGreen;
      BRadioButton   *fRadioBlack;
      BButton        *fButtonDraw;
};
```

Creating the Controls

All of the variables that are to be used as arguments to the control constructors are declared together in *MyHelloWindow.cpp*. Each of the controls is then created in the `MyHelloWindow` constructor. There are no surprises here—just use `new` with the appropriate control constructor and assign the resulting object to the proper `MyHelloWindow` data member.

You should be quite familiar with this process by now. To see the complete `MyHelloWindow` constructor listing, refer to *MyHelloWindow.cpp*.

Handling the Messages

All application-defined control messages are handled in the body of the `switch` statement in `MessageReceived()`. The checkbox message is handled by first checking the control's value, then disabling or enabling the three radio buttons as appropriate:

```
case CHECKBOX_SET_COLOR_MSG:
   int32  checkBoxState;

   checkBoxState = fCheckBoxSetColor->Value();
   if (checkBoxState == B_CONTROL_ON) {
      fRadioRed->SetEnabled(false);
      fRadioGreen->SetEnabled(false);
      fRadioBlack->SetEnabled(false);
   }
   else {
      fRadioRed->SetEnabled(true);
      fRadioGreen->SetEnabled(true);
      fRadioBlack->SetEnabled(true);
   }
   break;
```

Each of the radio buttons does nothing more than set the high color to an RGB color that matches the button's label:

```
case RADIO_RED_MSG:
   fMyView->SetHighColor(255, 0, 0, 255);
   break;

case RADIO_GREEN_MSG:
   fMyView->SetHighColor(0, 255, 0, 255);
   break;

case RADIO_BLACK_MSG:
   fMyView->SetHighColor(0, 0, 0, 255);
   break;
```

Clicking the Draw button results in a number of colored circles being drawn. Here the text field value is obtained to see how many circles to draw, and the high color is used in the drawing of those circles. Before drawing the circles, any old drawing is erased by whiting out an area of the window that is at least as big as the drawing area:

```
case BUTTON_DRAW_MSG:
   int32       numCircles;
   int32       i;
   BRect       areaRect(160.0, 70.0, 270.0, 180.0);
   BRect       circleRect(160.0, 70.0, 210.0, 120.0);
   const char  *textFieldText;

   textFieldText = fTextNumCircles->Text();
   numCircles = (int32)textFieldText[0] - 48;

   if ((numCircles < 1) || (numCircles > 9))
      numCircles = 5;

   fMyView->FillRect(areaRect, B_SOLID_LOW);

   for (i=1; i<=numCircles; i++) {
      fMyView->StrokeEllipse(circleRect, B_SOLID_HIGH);
      circleRect.OffsetBy(4.0, 4.0);
   }
   break;
```

The text field message is completely ignored. The program obtains the editable text when the user clicks the Draw button. As written, the program checks the text field input to see if the user entered a numeric character in the range of 1 to 9. If any other value (or character or string) has been entered, the program arbitrarily sets the number of circles to draw to five. This is a less-than-perfect way of doing things in that it allows the user to enter an invalid value. One way to improve the program would be to have the program react to a text field control message. That type of message is delivered to `MessageReceived()` when the user ends a typing session (that is, when the user clicks elsewhere, or presses the Enter, Return, or Tab key). `MessageReceived()` could then check the user-entered value and, if invalid, set the editable text area to a valid value (and, perhaps, post a window that informs the user what has taken place).

Modifying the ControlDemo Project

What the program draws isn't important to the demonstration of how to include a number of controls in a window. With the graphics information found in Chapter 4 you should be able to modify ControlDemo so that it draws something far more interesting. Begin by enlarging the window so that you have some working room. In the *MyHelloWorld.cpp* file, change the size of the `BRect` passed to the

MyHelloWindow constructor. Here I'm setting the window to have a width of 500 pixels and a height of 300 pixels:

```
aRect.Set(20, 30, 520, 430);
fMyWindow = new MyHelloWindow(aRect);
```

Limiting drawing to only three colors isn't a very user-friendly thing to do, so your next change might be to include a BColorControl object that lets the user choose any of the 256 system colors. The details of working with a color control were covered back in Chapter 5. Recall that you can easily support this type of control by first adding a BColorControl data member to the MyHelloWindow class:

```
class MyHelloWindow : public BWindow {
      ...
      ...
      BColorControl   *fColorControl;
};
```

Then, in the MyHelloWindow constructor, create the control and add it to the window. You'll need to determine the appropriate coordinates for the BPoint and a suitable Be-defined constant for the color_control_layout in order to position the color control in the window you're designing:

```
BPoint               leftTop(20.0, 50.0);
color_control_layout matrix = B_CELLS_4x64;
long                 cellSide = 16;

fColorControl = new BColorControl(leftTop, matrix, cellSide, "ColorControl");
AddChild(fColorControl);
```

Finally, when it comes time to draw, check to see which color the user has selected from the color control. You can do that when the user clicks the Draw button:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_DRAW_MSG:
         rgb_color  userColorChoice;

         userColorChoice = fColorControl->ValueAsColor();
         SetHighColor(userColorChoice);

         // now load up this section with plenty of graphics-drawing code
         break;

      ...
      ...

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

Now, review Chapter 4 to come up some ideas for drawing some really interesting graphics. Then add them under the `BUTTON_DRAW_MSG case` label in `MessageReceived()`!

7 Menus

In this chapter:
- *Menu Basics*
- *Working with Menus*
- *Multiple Menus*
- *Pop-up Menus*
- *Submenus*

Menus are the interface between the user and the program, and are the primary means by which a user carries out tasks. A Be program that makes use of menus usually places a menubar along the top of the content area of each application window—though it's easy enough to instead specify that a menubar appear elsewhere in a window.

A menu is composed of menu items, and resides in a menubar. You'll rely on the `BMenuBar`, `BMenu`, and `BMenuItem` classes to create menubar, menu, and menu item objects. Early in this chapter, you'll see how to create objects of these types and how to interrelate them to form a functioning menubar. After these menubar basics are described, the chapter moves to specific menu-related tasks such as changing a menu item's name during runtime and disabling a menu item or entire menu.

To offer the user a number of related options, create a single menu that allows only one item to be marked. Such a menu is said to be in radio mode, and places a checkmark beside the name of the most recently selected item. If these related items all form a subcategory of a topic that is itself a menu item, consider creating a submenu. A submenu is a menu item that, when selected, reveals still another menu. Another type of menu that typically holds numerous related options is a pop-up menu. A pop-up menu exists outside of a menubar, so it can be placed anywhere in a window. You'll find all the details of how to put a menu into radio mode, create a submenu, and create a pop-up menu in this chapter.

Menu Basics

A Be application can optionally include a menubar within any of its windows, as shown in Figure 7-1. In this figure, a document window belonging to the

StyledEdit program includes a menubar that holds four menus. As shown in the Font menu, a menu can include nested menus (submenus) within it.

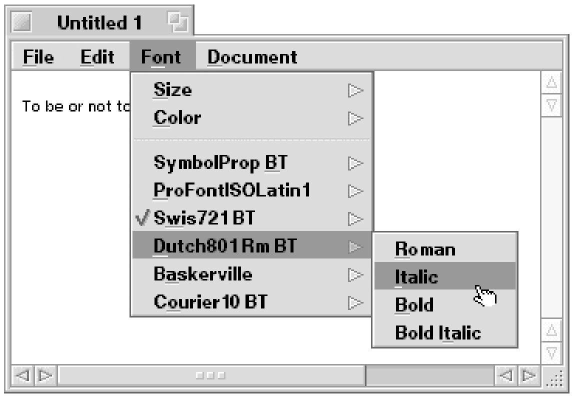


Figure 7-1. An application window can have its own menubar

Menus can be accessed via the keyboard rather than the mouse. To make the menubar the focus of keyboard keystrokes, the user presses both the Command and Escape keys. Once the menubar is the target of keystrokes, the left and right arrow keys can be used to drop, or display, a menu. Once displayed, items in a menu can be highlighted using the up and down arrow keys. The Enter key selects a highlighted item.

A second means of navigating menus and choosing menu items from the keyboard is through the use of *triggers*. One character in each menu name and in each menu item name is underlined. This trigger character is used to access a menu or menu item. After making the menubar the focus of the keyboard, pressing a menu's trigger character drops that menu. Pressing the trigger character of an item in that menu selects that item.

The topics of menubars, menus, and menu items are intertwined in such a way that moving immediately into a detailed examination of each in turn doesn't make sense. Instead, it makes more sense to conduct a general discussion of menu basics: creating menu item, menu, and menubar objects, adding menu item objects to a menu object, and adding a menu object to a menubar. That's what takes place on the next several pages. Included are a couple of example projects that include the code to add a simple menubar to a window. With knowledge of the interrelationship of the various menu elements, and a look at the code that implements a functional menubar with menus, it will be appropriate to move on to studies of the individual menu-related elements.

Adding a Menubar to a Window

The menubar, menu, and menu item are represented by objects of type BMenuBar, BMenu, and BMenuItem, respectively. To add these menu-related elements to your program, carry out the following steps:

1. Create a BMenuBar object to hold any number of menus.
2. Add the BMenuBar object to the window that is to display the menu.
3. For each menu that is to appear in the menubar:
    a. Create a BMenu object to hold any number of menu items.
    b. Add the BMenu object to the menubar that is to hold the menu.
    c. Create a BMenuItem object for each menu item that is to appear in the menu.

A menubar must be created before a menu can be added to it, and a menu must be created before a menu item can be added to it. However, the attaching of a menubar to a window and the attaching of a menu to a menubar needn't follow the order shown in the above list. For instance, a menubar, menu, and several menu items could all be created before the menu is added to a menubar.

When an example project in this book makes use of a menubar, its code follows the order given in the above list. It's worth noting that you will encounter programs that do things differently. Go ahead and rearrange the menu-related code in the MyHelloWindow constructor code in this chapter's first example project to prove that it doesn't matter when menu-related objects are added to parent objects.

Creating a menubar

The menubar is created through a call to the BMenuBar constructor. This routine accepts two arguments: a BRect that defines the size and location of the menubar, and a name for what will be the new BMenuBar object. Here's an example:

```
#define   MENU_BAR_HEIGHT   18.0

BRect     menuBarRect;
BMenuBar  *menuBar;

menuBarRect = Bounds();
menuBarRect.bottom = MENU_BAR_HEIGHT;

menuBar = new BMenuBar(menuBarRect, "MenuBar");
```

Convention dictates that a menubar appear along the top of a window's content area. Thus, the menubar's top left corner will be at point (0.0, 0.0) in window coordinates. The bottom of the rectangle defines the menu's height, which is typically 18 pixels. Because a window's menubar runs across the width of the window—regardless of the size of the window—the rectangle's right boundary can be set to the current width of the window the menubar is to reside in. The call to the `BWindow` member function `Bounds()` does that. After that, the bottom of the rectangle needs to be set to the height of the menu (by convention it's 18 pixels).

Creating a menubar object doesn't automatically associate that object with a particular window object. To do that, call the window's `BWindow` member function `AddChild()`. Typically a window's menubar will be created and added to the window from within the window's constructor. Carrying on with the above snippet, in such a case the menubar addition would look like this:

```
AddChild(menuBar);
```

Creating a menu

Creating a new menu involves nothing more than passing the menu's name to the `BMenu` constructor. For many types of objects, the object name is used strictly for "behind-the-scenes" purposes, such as in obtaining a reference to the object. A `BMenu` object's name is also used for that purpose, but it has a second use as well—it becomes the menu name that is displayed in the menubar to which the menu eventually gets attached. Here's an example:

```
BMenu  *menu;

menu = new BMenu("File");
```

Because one thinks of a menubar as being the organizer of its menus, it may be counterintuitive that the `BMenuBar` class is derived from the `BMenu` class—but indeed it is. A menubar object can thus make use of any `BMenu` member function, including the `AddItem()` function. Just ahead you will see how a menu object invokes `AddItem()` to add a menu item to itself. Here you see how a menubar object invokes `AddItem()` to add a menu to itself:

```
menuBar->AddItem(menu);
```

Creating a menu item

Each item in a menu is an object of type `BMenuItem`. The `BMenuItem` constructor requires two arguments: the menu item name as it is to appear listed in a menu, and a BMessage object. Here's how a menu item to be used as an Open item in a File menu might be created:

```
#define    MENU_OPEN_MSG   'open'

BMenuItem  *menuItem;

menuItem = new BMenuItem("Open", new BMessage(MENU_OPEN_MSG));
```

Add the menu item to an existing menu by invoking the menu object's BMenu member function AddItem(). Here menu is the BMenu object created in the previous section:

```
menu->AddItem(menuItem);
```

While the above method of creating a menu item and adding it to a menu in two steps is perfectly acceptable, the steps are typically carried out in a single action:

```
menu->AddItem(new BMenuItem("Open", new BMessage(MENU_OPEN_MSG)));
```

Handling a Menu Item Selection

Handling a menu item selection is so similar to handling a control click that if you know one technique, you know the other. You're fresh from seeing the control (you either read Chapter 6, *Controls and Messages*, before this chapter, or you just jumped back and read it now, right?), so a comparison of menu item handling to control handling will serve well to cement in your mind the practice used in each case: message creation and message handling.

In Chapter 6, you read that to create a control, such as a button, you define a message constant and then use new along with the control's constructor to allocate both the object and the model message—as in this snippet that creates a standard push button labeled "OK":

```
#define    BUTTON_OK_MSG   'btmg'

BRect      buttonRect(20.0, 20.0, 120.0, 50.0);
BButton    *buttonOK;

buttonOK = new BButton(buttonRect, "OKButton",
                       "OK", new BMessage(BUTTON_OK_MSG));
```

Menu item creation is similar: define a message constant and then create a menu item object:

```
#define    MENU_OPEN_MSG   'open'

BMenuItem  *menuItem;

menuItem = new BMenuItem("Open", new BMessage(MENU_OPEN_MSG));
```

An application-defined message is sent from the Application Server to a window. The window receives the message in its `MessageReceived()` function. So the recipient window's `BWindow`-derived class must override `MessageReceived()`—as demonstrated in Chapter 6 and again here:

```
class MyHelloWindow : public BWindow {

   public:
                       MyHelloWindow(BRect frame);
      virtual bool     QuitRequested();
      virtual void     MessageReceived(BMessage* message);
   ...
   ...
};
```

The implementation of `MessageReceived()` defines the action that occurs in response to each application-defined message. You saw several examples of this in Chapter 6, including a few projects that simply used `beep()` to respond to a message. Here's how `MessageReceived()` would be set up for a menu item message represented by a constant named `MENU_OPEN_MSG`:

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

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

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

Menubar Example Project

The SimpleMenuBar project generates a window that includes a menubar like the one in Figure 7-2. Here you see that the window's menubar extends across the width of the window, as expected, and holds a menu with a single menu item in it. Choosing the Beep Once item from the Audio menu sounds the system beep.

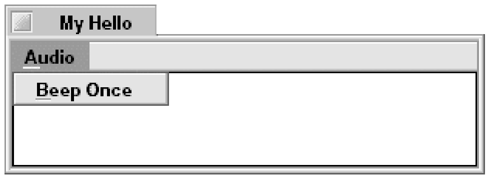


Figure 7-2. The SimpleMenuBar program's window

Preparing the window class for a menubar

If a window is to let a user choose items from a menubar, its `BWindow`-derived class must override `MessageReceived()`. Additionally, you may opt to keep track of the window's menubar by including a `BMenuBar` data member in the class. You can also include `BMenu` and `BMenuItem` data members in the class declaration, but keeping track of the menubar alone is generally sufficient. As demonstrated later in this chapter, it's a trivial task to find any menu or menu item and get a reference to it by way of a menubar reference. Here's how the window class header file (the *MyHelloWindow.h* file for this project) is set up for menu item handling:

```
#define   MENU_BEEP_1_MSG    'bep1'

class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
      BMenuBar       *fMenuBar;
};
```

Creating the menubar, menu, and menu item

By default, the height of a menubar will be 18 pixels (though the system will automatically alter the menubar height to accommodate a large font that's used to display menu names). So we'll document the purpose of this number by defining a constant:

```
#define   MENU_BAR_HEIGHT   18.0
```

After the constant definition comes the `MyHelloWindow` constructor. In past examples, the first three lines of this routine created a view that occupies the entire content area of the new window. This latest version of the constructor uses the same three lines, but also inserts one new line after the call to `OffsetTo()`:

```
frame.OffsetTo(B_ORIGIN);
frame.top += MENU_BAR_HEIGHT + 1.0;

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

The `frame` is the `BRect` that defines the size and screen location of the new window. Calling the `BRect` function `OffsetTo()` with an argument of `B_ORIGIN` redefines the values of this rectangle's boundaries so that the rectangle remains the same overall size, but has a top left corner at window coordinate (0.0, 0.0). That's perfect for use when placing a new view in the window. Here, however, I want the view to start not at the window's top left origin, but just below the menubar that will soon be created. Bumping the top of the `frame` rectangle down the height of the menu, plus one more pixel to avoid an overlap, properly sets up the rectangle for use in creating the view.

If you work on a project that adds a view and a menubar to a window, and mouse clicks on the menubar's menus are ignored, the problem most likely concerns the view. It's crucial to reposition a window's view so that it lies below the area that will eventually hold the menubar. If the view occupies the area that the menubar will appear in, the menubar's menus may not respond to mouse clicks. (Whether a menu does or doesn't respond will depend on the order in which the view and menubar are added to the window.) If the view overlaps the menubar, mouse clicks may end up directed at the view rather than the menubar.

The menubar is created by defining the bar's boundary and then creating a new `BMenuBar` object. A call to the `BWindow` function `AddChild()` attaches the menubar to the window:

```
BRect   menuBarRect;

menuBarRect = Bounds();
menuBarRect.bottom = MENU_BAR_HEIGHT;

fMenuBar = new BMenuBar(menuBarRect, "MenuBar");
AddChild(fMenuBar);
```

The menubar's one menu is created using the `BMenu` constructor. A call to the `BMenu` function `AddItem()` attaches the menu to the existing menubar:

```
BMenu   *menu;

menu = new BMenu("Audio");
fMenuBar->AddItem(menu);
```

A new menu is initially devoid of menu items. Calling the `BMenu` function `AddItem()` adds one item to the menu:

```
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
```

Subsequent calls to `AddItem()` append new items to the existing ones. Because the menu item won't be referenced later in the routine, and as a matter of convenience, the creation of the new menu item object is done within the call to `AddItem()`. We could expand the calls with no difference in the result. For instance, the above line of code could be written as follows:

```
BMenuItem  *theItem;

theItem = new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG));
menu->AddItem(theItem);
```

Here, in its entirety, is the `MyHelloWindow` constructor for the SimpleMenuBar project:

```
#define   MENU_BAR_HEIGHT   18.0

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("Audio");
   fMenuBar->AddItem(menu);

   menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));

   Show();
}
```

Handling a menu item selection

To respond to the user's menu selection, all I did on this project was copy the `MessageReceived()` function that handled a click on a control in a Chapter 6 example project. The simplicity of this code sharing is further proof that a menu item selection is handled just like a control:

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

   switch(message->what)
   {
      case MENU_BEEP_1_MSG:
         beep();
         break;

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

Window resizing and views

The SimpleMenuBar example introduces one topic that's only partially related to menus: how the resizing of a window affects a view attached to that window. A titled or document window (a window whose constructor contains a third parameter value of either `B_TITLED_WINDOW` or `B_DOCUMENT_WINDOW`) is by default resizable. (Recall that a `BWindow` constructor fourth parameter of `B_NOT_RESIZABLE` can alter this behavior.) The SimpleMenuBar window is resizable, so the behavior of the views within the window isn't static.

Like anything you draw, a menubar is a type of view. When a window that displays a menubar is resized, the length of the menubar is automatically altered to occupy the width of the window. This is a feature of the menubar, not your application-defined code.

Unlike a menubar, a `BView`-derived class needs to specify the resizing behavior of an instance of the class. This is done by supplying the appropriate Be-defined constant in the `resizingMode` parameter (the third parameter) to the `BView` constructor. In past examples, the `B_FOLLOW_ALL` constant was used for the `resizingMode`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
}
```

The `B_FOLLOW_ALL` constant sets the view to be resized in conjunction with any resizing that takes place in the view's parent. If the view's parent is the window (technically, the window's top view) and the window is enlarged, the view will be enlarged proportionally. Likewise, if the window size is reduced, the view size is reduced. As a window is resized, it requires constant updating—so the `Draw()` function of each view in the window is repeatedly invoked. This may not always be desirable, as the SimpleMenuBar example demonstrates. This program's window is filled with a view of the class `MyDrawView`. In this project, the `Draw()` function for the `MyDrawView` class draws a rectangle around the frame of the view:

```
void MyDrawView::Draw(BRect)
{
   BRect  frame = Bounds();

   StrokeRect(frame);
}
```

If the `MyDrawView` view has a `resizingMode` of `B_FOLLOW_ALL`, the result of enlarging a window will be a number of framed rectangles in the window—one rectangle for each automatic call that's been made to `Draw()`. Figure 7-3 illustrates this.

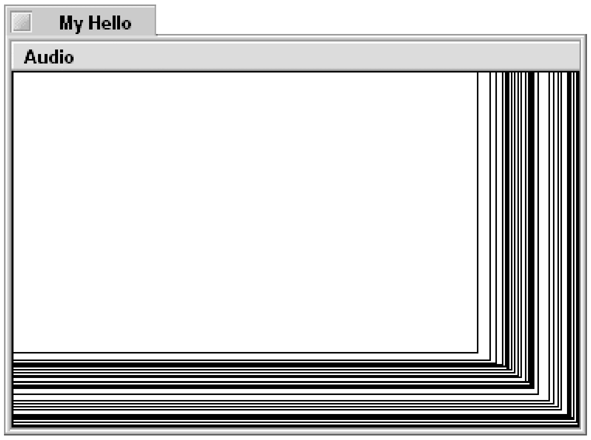


Figure 7-3. A view's resizing mode needs to be coordinated with window resizing

The SimpleMenuBar project avoids the above phenomenon by using the `B_FOLLOW_NONE` constant for the `resizingMode`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_NONE, B_WILL_DRAW)
{
}
```

This constant sets the view to remain a fixed size and at a fixed location in its parent—regardless of what changes take place in the parent's size. Figure 7-4 shows how the view in the SimpleMenuBar project's window looks when the program's window is enlarged.

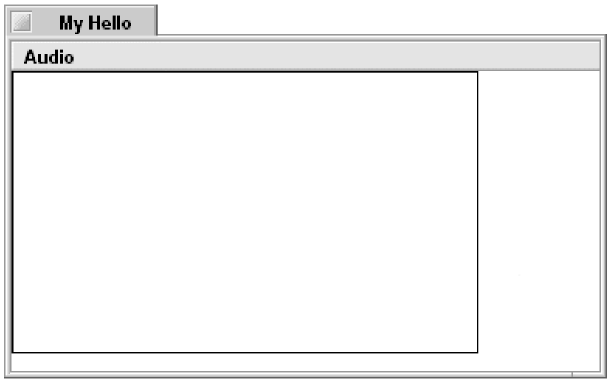


Figure 7-4. A fixed-size view is unaffected by window resizing

Menubar and Control Example Project

Now that you know how to add controls and menus to a window, there's a strong likelihood that you may want to include both within the same window. The MenuAndControl project demonstrates how to do this. As Figure 7-5 shows, the MenuAndControl program's window includes the same menubar that was introduced in the previous example (the SimpleMenuBar project). Sounding the system beep is accomplished by either choosing the one menu item or by clicking on the button. The view, which in this program doesn't occupy the entire window content area, remains empty throughout the running of the program. Here the view is used to elaborate on last section's discussion of window resizing and views. In this chapter's TwoMenus project, the view displays one of two drawings.

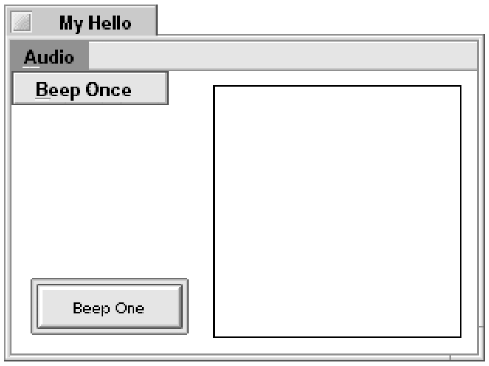


Figure 7-5. The MenuAndControl application window

Preparing the window class for a menubar and control

Both the push button control and the one menu item require the definition of a message constant:

```
#define   BUTTON_BEEP_MSG    'beep'
#define   MENU_BEEP_1_MSG    'bep1'
```

To handle messages from both the push button and the menu item, override `MessageReceived()`. Data members for the control and menubar appear in the `BWindow`-derived class as well:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage* message);

   private:
      MyDrawView     *fMyView;
```

```
        BButton         *fButtonBeep;
        BMenuBar        *fMenuBar;
};
```

Creating the menu-related elements and the control

The MyHelloWindow constructor begins with the customary creation of a view for the window. Here, however, the view doesn't occupy the entire content area of the window. Recall that the SimpleMenuBar project set up the view's area like this:

```
frame.OffsetTo(B_ORIGIN);
frame.top += MENU_BAR_HEIGHT + 1.0;
```

The MenuAndControl project instead sets up the view's area as follows:

```
frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);
```

Figure 7-5 shows that the resulting view occupies the right side of the program's window. Since a view in past examples occupied the entire content area of a window, items were added to the view in order to place them properly. For instance, if the MyHelloWindow defined a BButton data member named fButtonBeep and a MyDrawView data member named fMyView, the addition of the button to the window would look like this:

```
fMyView->AddChild(fButtonBeep);
```

The MyHelloWindow class declared in the MenuAndControl project does in fact include the two data members shown in the above line of code. This project's MyHelloWindow constructor, however, adds the button directly to the window rather than to the window's view. A call to the BButton function MakeDefault() serves to outline the button:

```
AddChild(fButtonBeep);
fButtonBeep->MakeDefault(true);
```

Looking back at Figure 7-5, you can see that in this project it wouldn't make sense to add the button to the window's view. If I did that, the button would end up being placed not on the left side of the window, but on the right side.

After adding the button to window, we create the menubar and add it to the window, create the menu and add it to the menubar, and create the menu item and add it to the menu. The menu-related code is identical to that used in the previous example (the SimpleMenuBar project). Note that there is no significance to my placing the control-related code before the menu-related code—the result is the same regardless of which component is added to the window first:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);
   fMyView = new MyDrawView(frame, "MyDrawView");
   AddChild(fMyView);

   fButtOnBeep = new BButton(buttonBeepRect, buttonBeepName,
                             buttonBeepLabEl, new BMessage(BUTTON_BEEP_MSG));
   AddChild(fButtonBeep);
   fButtonBeep->MakeDefault(true);

   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("Audio");
   fMenuBar->AddItem(menu);

   menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));

   Show();
}
```

Handling a menu item selection and a control click

It's important to keep in mind that "a message is a message"—a window won't distinguish between a message issued by a click on a control and a message generated by a menu item selection. So the same `MessageReceived()` function handles both message types:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         beep();
         break;

      case MENU_BEEP_1_MSG:
         beep();
         break;

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

Because a click on the Beep One button and a selection of the Beep Once menu item both result in the same action—a sounding of the system beep—I could have defined a single message constant. For instance, instead of defining both the `BUTTON_BEEP_MSG` and the `MENU_BEEP_1_MSG` constants, I could have simply defined, say, a `BEEP_MSG`:

```
#define   BEEP_MSG    'beep'
```

The One View Technique

Now you've seen numerous examples that establish one window-encompassing view, and one example that doesn't. Which method should you use? Sorry, but the answer is an ambiguous "It depends." It depends on whether your program will be making "universal" changes to a window, but it behooves you to get in the habit of always including a window-filling view in each of your `BWindow`-derived classes. If, much later in project development, you decide a window needs to be capable of handling some all-encompassing change or changes, you can just issue appropriate calls to the view and keep changes to your project's code to a minimum.

As an example of a universal change to a window, consider a window that displays several controls and a couple of drawing areas. If for some reason all of these items need to be shifted within the window, it would make sense to have all of the items attached to a view within the window rather than to the window itself. Then a call to the `BView` `MoveBy()` or `MoveTo()` member function easily shifts the window's one view, and its contents, within the window.

The second reason to include a window-filling view—programmer's preference—is related to the first reason. For each `BWindow`-derived class you define, you might prefer as a matter of habit to also define a `BView`-derived class:

```
class MyFillView : public BView {

   public:
                       MyDrawView(BRect frame, char *name);
      virtual void     AttachedToWindow();
      virtual void     Draw(BRect updateRect);
};
```

If you have no immediate plans for the view, simply implement the view class member functions as empty:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_NONE, B_WILL_DRAW)
{
}

void MyDrawView::AttachedToWindow()
{
}

void MyDrawView::Draw(BRect)
{
}
```

—Continued—

In the window's constructor, create and add a view. Then add all of the window's controls and other views to this main view:

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

   fMyView = new MyFillView(frame, "MyFillWindowView");
   AddChild(fMyView);

        fButton = new BButton(buttonRect, buttonName,
                        buttonLabel, new BMessage(BUTTON_MSG));
   fMyView->AddChild(fButton);
   ...
   ...
}
```

The `BButton` constructor would then make use of this new message constant:

```
fButtonBeep = new BButton(buttonBeepRect, buttonBeepName,
                          buttonBeepLabel, new BMessage(BEEP_MSG));
```

The creation of the menu item makes use of this same message constant:

```
menu->AddItem(new BMenuItem("Beep Once", new BMessage(BEEP_MSG)));
```

A click on the button or a selection of the menu item would both result in the same type of message being sent to the window, so the `MessageReceived()` function would now need only one case label:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BEEP_MSG:
         beep();
         break;

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

This scenario further demonstrates the notion that a window isn't interested in the source of a message—it cares only about the type of the message (as defined by the message constant). That's all well and good, but what's the likelihood of a real-world application having a window that includes both a control and a menu item that produce the same action? Perhaps higher than you might guess. It's a common practice in many programs to include a control window (usually referred to as a palette) that as a matter of convenience holds a number of buttons that mimic the actions of commonly used menu items.

Window resizing and the view hierarchy

This chapter's first example, the SimpleMenuBar project, illustrated how window resizing affects a view. Including a control in the window of the MenuAndControl project provides an opportunity to illuminate resizing further.

A view created with a `resizingMode` of `B_FOLLOW_ALL` is one that is resized along with its resized parent. A `resizingMode` of `B_FOLLOW_NONE` fixes a view in its parent—even as the parent is resized. A view can also be kept fixed in size, but move within its parent. How it moves in relationship to the parent is dependent on which of the Be-defined constants `B_FOLLOW_RIGHT`, `B_FOLLOW_LEFT`, `B_FOLLOW_BOTTOM`, or `B_FOLLOW_TOP` are used for the `resizingMode`. Each constant forces the view to keep its present distance from one parent view edge. Constants can also be used in combination with one another by using the OR operator (|). In the MenuAndControl project, the `MyDrawView` constructor combines `B_FOLLOW_RIGHT` and `B_FOLLOW_BOTTOM`:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_RIGHT | B_FOLLOW_BOTTOM, B_WILL_DRAW)
{
}
```

The result of this pairing of constants is a view that remains fixed to the parent view's (the window here) right and bottom. In Figure 7-6, the MenuAndControl window's size has been reduced horizontally and increased vertically, yet you see that the view has kept its original margin of about ten pixels from the window's right side and about ten pixels from the window's bottom edge.

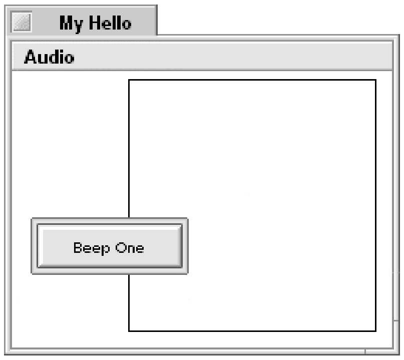


Figure 7-6. A view that keeps a constant-size right and bottom border in its parent

A complete description of all the `resizingMode` constants is found in the `BView` section of the Interface Kit chapter of the Be Book.

Figure 7-6 raises an interesting issue regarding the window's view hierarchy. In the figure, you see that the view appears to be drawn behind the button. As of this writing, the view hierarchy determines the drawing order of the views in a window. When a window requires updating, each view's `Draw()` function is automatically invoked. The order in which the `Draw()` functions are called is first dependent on the view hierarchy, starting from the window's top view down to its bottom views. For views on the same view hierarchy level, the order in which their `Draw()` functions are invoked depends on the order in which the views were added, or attached, to their parent view. The first view added becomes the first view redrawn. Such is the case with the `BButton` view and the `MyDrawView`. Each was added to the window, so these two views are at the same level of the view hierarchy, just under the window's top view. The `MyDrawView` was added first, so it is updated first. After its `Draw()` function is called, the `BButton` `Draw()` routine is called—thus giving the button the appearance of being in front of the `MyDrawView`:

```
MyHelloWindow::MyHelloWindow(BRect frame)
     : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   frame.Set(130.0, MENU_BAR_HEIGHT + 10.0, 290.0, 190.0);

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

   fButtonBeep = new BButton(buttonBeepRect, buttonBeepName,
                             buttonBeepLabel, new BMessage(BUTTON_BEEP_MSG));
   AddChild(fButtonBeep);
   ...
   ...
}
```

If the order of the two calls to `AddChild()` were switched, you would expect the button to be redrawn first, and the `MyDrawView` to be updated next. Give it a try by editing the *MyHelloWindow.cpp* file of the MenuAndControl project. When you do that, you'll see that running the program and shrinking the window results in the `MyDrawView` obscuring the button.

Notice that the discussion of view updating order starts of with "As of this writing…". There is no guarantee that this order based on view hierarchy will always be in effect. In short, don't make assumptions about view updating order. Instead, make an effort not to overlap views.

Working with Menus

Your program can get by with simple, static menus and menu items—but why stop there? The menubar, menus, and menu items of a program should reflect the current state of a program. You can make sure they do that by implementing menus so that they give the user visual cues as to what is being done, and what can and can't be done. For instance, a menu item can be marked with a checkmark to let the user know the item is currently in force. Or a menu item's name can be changed, if appropriate, to what is currently taking place in the program. A menu item—or an entire menu—can be disabled to prevent the user from attempting to perform some action that doesn't make sense at that point in the program is at. These and other menu-altering techniques are covered in this section.

Creating a Menu Item

Each menu item in a menu is an object based on the BMenuItem class. Menu item objects were introduced earlier in this chapter—here they're studied in much greater detail.

The BMenuItem class

A menu item is created using the BMenuItem constructor, the prototype of which is shown here:

```
BMenuItem(const char *label,
          BMessage   *message,
          char       shortcut = 0,
          uint32     modifiers = 0)
```

The first BMenuItem parameter, label, assigns the item its name, which is displayed as the item's label when the user pulls down the menu in which the item appears.

The message parameter assigns a message of a particular type to the menu item. When the user chooses the item, the message is delivered to the window that holds the menubar containing the menu item. That window's

`MessageReceived()` function becomes responsible for carrying out the action associated with the menu item.

The third `BMenuItem` constructor parameter, `shortcut`, is optional. The default value used by the constructor is `0`, but if a character is passed, that character becomes the menu item's keyboard shortcut. When the user presses the shortcut key in conjunction with a modifier key, the menu item is considered selected—just as if the user chose it from the menu. The fourth parameter, `modifiers`, specifies what key is considered the modifier key. A keyboard shortcut *must* include the Command key (which by default is the Alt key on a PC and the Command key on a Macintosh) as its modifier key, but it can also require that one or more other modifier keys be pressed in order to activate the keyboard shortcut. Any of the Be-defined modifier key constants, including `B_COMMAND_KEY`, `B_SHIFT_KEY`, `B_OPTION_KEY`, and `B_CONTROL`, can be used as the `modifiers` parameter. For instance, to designate that a key combination of Command-Q represent a means of activating a Quit menu item, pass `'Q'` as the shortcut parameter and `B_COMMAND_KEY` as the modifiers parameter. To designate that a key combination of Alt-Shift-W (on a PC) or Command-Shift-W (on a Mac) represents a means of closing all open windows, pass `'W'` as the `shortcut` parameter, and the ored constants `B_COMMAND_KEY | B_SHIFT_KEY` as the `modifiers` parameter.

Creating a BMenuItem object

A menu item is often created and added to a menu in one step by invoking the `BMenuItem` constructor from right within the parameter list of a call to the `BMenu` function `AddItem()`:

```
menu->AddItem(new BMenuItem("Start", new BMessage(START_MSG)));
```

Alternatively, a menu item can be created and then added to a menu in a separate step:

```
menuItem = new BMenuItem("Start", new BMessage(START_MSG));
menu->AddItem(menuItem);
```

Regardless of the method used, to this point the `BMenuItem` constructor has been passed only two arguments. To assign a keyboard shortcut to a menu item, include arguments for the optional third and fourth parameters. Here a menu item named "Start" is being given the keyboard shortcut Command-Shift-S (with the assumption that the slightly more intuitive keyboard shortcut Command-S is perhaps already being used for a "Save" menu item):

```
menu->AddItem(new BMenuItem("Start", new BMessage(START_MSG), 'S',
                            B_COMMAND_KEY | B_SHIFT_KEY));
```

If a menu item is associated with a keyboard shortcut, and if that shortcut uses a modifier key, a symbol for that modifier key appears to the right of the menu item.

The symbol provides the user with an indication of what key should be pressed in conjunction with the character key that follows the symbol. Figure 7-7 provides several examples. In that figure, I've set up a menu with four items. The name I've given each item reflects the modifier key or keys that need to be pressed in order to select the item. For instance, the first menu item is selected by pressing Command-A. This next snippet provides the code necessary to set up the menu shown in Figure 7-7:

```
menu->AddItem(new BMenuItem("Command", new BMessage(A_MSG), 'A',
                                                  B_COMMAND_KEY));
menu->AddItem(new BMenuItem("Command-Shift", new BMessage(B_MSG), 'B',
                            B_COMMAND_KEY | B_SHIFT_KEY));
menu->AddItem(new BMenuItem("Command-Shift-Option", new BMessage(C_MSG), 'C',
                            B_COMMAND_KEY | B_SHIFT_KEY | B_OPTION_KEY));
menu->AddItem(new BMenuItem("Command-Shift-Option-Control",
                            new BMessage(D_MSG), 'D',
                            B_COMMAND_KEY | B_SHIFT_KEY |
                            B_OPTION_KEY | B_CONTROL_KEY));
```

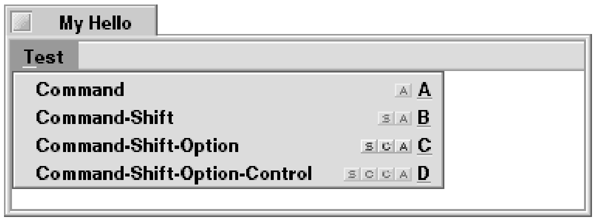

My Hello
Test
Command A A
Command-Shift S A B
Command-Shift-Option S C A C
Command-Shift-Option-Control S C C A D

Figure 7-7. A menu that includes items that use several shortcut modifier keys

As illustrated in the preceding snippet and Figure 7-7, menu items are displayed in a menu in the order in which they were added to the `BMenu` object. To reposition items, simply rearrange the order of the calls to `AddItem()`.

A separator is a special menu item that is nothing more than an inactive gray line. It exists only to provide the user with a visual cue that some items in a menu are logically related, and are thus grouped together. To add a separator item, invoke the `BMenu` function `AddSeparatorItem()`:

```
menu->AddSeparatorItem();
```

Figure 7-15, later in this chapter, includes a menu that has a separator item.

Accessing a Menu Item

If a program is to make a change to a menu item, it of course needs access to the item. That can be accomplished by storing either the `BMenuItem` object or the `BMenuBar` object as a data member.

Storing a menu item in a data member

If you need access to a menu item after it is created, just store it in a local variable:

```
BMenu      *menu;
BMenuItem  *menuItem;
...
menu->AddItem(menuItem = new BMenuItem("Beep", new BMessage(BEEP_MSG)));
```

This snippet creates a menu item and adds it to a menu—as several previous examples have done. Here, though, the menu item object created by the `BMenuItem` constructor is assigned to the `BMenuItem` variable `menuItem`. Now the project has access to this one menu item in the form of the `menuItem` variable.

This menu item access technique is of limited use. Access to a menu item may need to take place outside of the function which created the menu item. If that's the case, a window class data member can be created to keep track of an item for the life of the window:

```
class MyHelloWindow : public BWindow {
   ...

   private:
      ...
      BMenuItem   *fMenuItem;
};
```

Now, when the menu item is created, store a reference to it in the `fMenuItem` data member:

```
menu->AddItem(fMenuItem = new BMenuItem("Beep", new BMessage(BEEP_MSG)));
```

Using this technique for creating the menu item, the menu item can be accessed from any member function of the window class.

Storing a menubar in a data member

If several menu items are to be manipulated during the running of a program, it may make more sense to keep track of just the menubar rather than each individual menu item:

```
class MyHelloWindow : public BWindow {
   ...

   private:
      ...
      BMenuBar   *fMenuBar;
};
```

If the menubar can be referenced, the `BMenu` member function `FindItem()` can be used to access any menu item in any of its menus. Pass `FindItem()` a menu's label (the string that represents the menu item name that is displayed to the user), and the function returns that menu item's BMenuItem object:

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Beep");
```

Marking Menu Items

When selected, a menu item can be given a check mark to the left of the item name. When selected again, this same menu item can become unchecked. Figure 7-8 shows a menu with two marked items.

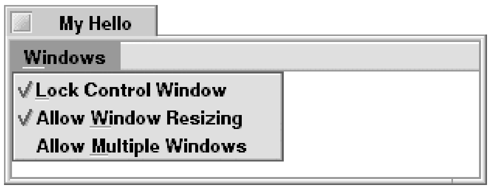


Figure 7-8. A menu with marked, or checked, menu items

Marking a menu item

To mark or unmark a menu item, invoke the item's BMenuItem function SetMarked(). Passing this function a value of true marks the item, while passing a value of false unmarks it. Attempting to mark an already marked item or unmark an already unmarked item has no effect. This next snippet sets up the Windows menu items shown in Figure 7-8. Assume that data members exist to keep track of each of the three Windows menu items, and that the menubar and menu have already been created:

```
BMenu      *menu;
BMenuItem  *menuItem;

menu->AddItem(fLockWindMenuItem = new BMenuItem("Lock Control Window",
              new BMessage(LOCK_WIND_MSG)));
fLockWindMenuItem->SetMarked(true);
menu->AddItem(fResizeWindMenuItem = new BMenuItem("Allow Window Resizing",
              new BMessage(RESIZE_WIND_MSG)));
fResizeWindMenuItem ->SetMarked(true)
menu->AddItem(fMultipleWindMenuItem = new BMenuItem("Allow Multiple Windows",
              new BMessage(MULTIPLE_WIND_MSG)));
```

If a menu item is to be initially marked (as the Lock Control Window and Allow Window Resizing items are in the above snippet), save a reference to the item when creating it. Then use that BMenuItem object to mark the item.

To find out whether a menu item is marked, call the `BMenuItem` function `IsMarked()`. For instance, after the user selects a menu item, call `IsMarked()` to determine whether to pass `SetMarked()` a value of `true` or `false` and thus toggle the menu's mark. The updating of an item's mark can take place in the `MessageReceived()` function, as shown here for the first menu item from the Windows menu of Figure 7-8:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case LOCK_WIND_MSG:
         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Lock Control Window");
         if (theItem->IsMarked()) {
            theItem->SetMarked(false);
            // code to unlock a window (allow user to move it)
         }
         else {
            theItem->SetMarked(true);
            // code to lock a window (prevent user from moving it)
         }
      ...
      ...
      default:
         BWindow::MessageReceived(message);
   }
}
```

Marking a menu item in a menu of related items

A menu may treat all of its items as related options, with the intent of allowing only one item to be in force at any time. The Audio menu in Figure 7-9 provides an example of such a menu. Here the user is expected to choose one of the two beeping options. A subsequent click on the Beep button sounds the system beep either once or twice, depending on the currently selected menu item.

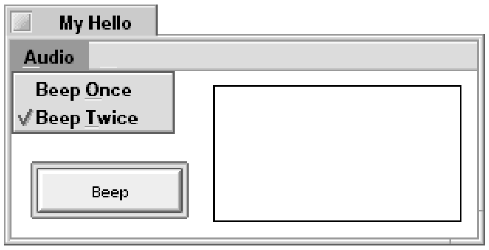


Figure 7-9. A menu with related options

If all of the items in a menu are to act as a single set of options, you can set up the menu to automatically handle the checking and unchecking of its items. The BMenu function SetRadioMode() instructs a menu to allow only one of its items to be marked at any time. Additionally, setting a menu to radio mode provides the menu with the power to automatically mark whatever item the user chooses, and to unmark the previously selected item. To set a menu to radio mode, pass SetRadioMode() a value of true. This next snippet sets up the Audio menu pictured in Figure 7-9, marks the Beep Twice item, and sets the Audio menu to radio mode:

```
BMenu      *menu;
BMenuItem  *menuItem;
...
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
menu->AddItem(menuItem = new BMenuItem("Beep Twice",
                                   new BMessage(MENU_BEEP_2_MSG)));
menuItem->SetMarked(true);
menu->SetRadioMode(true);
```

While a menu in radio mode will properly update its item mark in response to menu item selections, it is your responsibility to check which item is to be initially marked. As shown above, that's accomplished via a call to SetMarked(). If a menu item is checked, your code should also make sure that the feature or option to be set is in fact set.

The BMenu function FindMarked() returns the menu item object of the currently marked item in a menu. When a menu is in radio mode, only one item can be marked at any time. In the next snippet, the Audio menu shown in Figure 7-9 is kept track of by a BMenu data member named fAudioMenu. Calling FindMarked() on this item returns the BMenuItem object of the currently marked item:

```
BMenuItem  *theItem;

theItem = fAudioMenu->FindMarked();
```

FindMarked() can be used on a menu that isn't set to radio mode, too—but its usefulness is then diminished because there may be more than one item marked. If more than one item is marked, FindMarked() returns a reference to the first marked item encountered (it starts at the first item in a menu and moves down).

Changing a Menu Item's Label

A menu item's label can be changed at any time. To do that, gain access to the menu item and then invoke the BMenuItem function SetLabel(). In the next snippet, a menu item named “Start Simulation” is being renamed to “Stop Simulation.”

```
fSimMenuItem->SetLabel("Stop Simulation");
```

To find out the current label of a menu item, call the `BMenuItem` function `Label()`. A call to this routine can be made prior to a call to `SetLabel()` to determine the item’s current label before changing it to a new string. The type of label-changing shown in the above snippet is a good candidate for the use of both `Label()` and `SetLabel()`. If the user starts some sort of action by choosing a menu item, that menu item’s label might change in order to provide the user with a means of stopping the action. The `MessageReceived()` function holds the label-changing code:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_SIMULATION_RUN_MSG:
         const char  *menuItemLabel;

         menuItemLabel = fSimMenuItem->Label();
         if ((strcmp(menuItemLabel, "Start Simulation") == 0)) {
            fSimMenuItem->SetLabel("Stop Simulation");
            // invoke application-defined routine to start simulation
         }
         else {
            fSimMenuItem->SetLabel("Start Simulation");
            // invoke application-defined routine to stop simulation
         }
         break;

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

This snippet is the first in this book to make use of a standard C library function. While the member functions of the classes of the BeOS take care of many of your programming needs, they of course can’t account for every task your program is to perform. Before writing an application-defined routine, don’t forget to fall back on your C and C++ background to select a standard library function where appropriate. Here I use the string comparison routine `strcmp()` to compare the characters in a menu item’s label to the string “Start Simulation.” If the strings are identical, `strcmp()` returns a value of `0`. In such a case, the menu item label needs to be changed to signal that the simulation is running and to allow the user to halt the action. As shown in the two comments in the above code, whatever it is that is to be simulated is left as an exercise for the reader!

Disabling and Enabling Menus and Menu Items

When a menu is created, the menu and each of its items are all initially enabled. The entire menu—including the menu's name in the menubar and all its items—or any individual menu item can be disabled.

Disabling and enabling a menu item

A disabled menu item appears dim and is inactive—the user can see the item and read its label, but can't select it (releasing the mouse button while the cursor is over the item has no effect). A menu item can be disabled or re-enabled by invoking the `BMenuItem` member function `SetEnabled()`. An argument of `true` enables the item, while an argument of `false` disables the item. By default, a menu item is enabled upon creation. To disable a newly created menu item, call `SetEnabled()` just after the menu item is created. Assuming that the menu's window keeps track of its menubar in a data member named `fMenuBar`, the following snippet could be used to disable a menu item named "Start":

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Start");
theItem->SetEnabled(false);
```

The current state of a menu item can be found by invoking the item's `IsEnabled()` function. This routine returns a value of `true` if the item is presently enabled, `false` if it's disabled.

If the enabling or disabling of a menu item is to take place in response to a message (whether initiated by a different menu item selection or a control click), include the menu item enabling/disabling code in `MessageReceived()`. Here a menu item named Advanced Options is enabled or disabled in response to an application-defined `TOGGLE_OPTIONS_MSG` message:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case TOGGLE_OPTIONS_MSG:
         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Advanced Options");
         if (theItem->IsEnabled())
            theItem->SetEnabled(false);
         else
            theItem->SetEnabled(true);
      ...
      ...
      default:
         BWindow::MessageReceived(message);
   }
}
```

What happens when `SetEnabled()` or `IsEnabled()` is invoked depends on the state of the menu the item resides in. If the menu itself is disabled, attempting to enable an individual item in that menu will fail. Thus, even if your code hasn't explicitly disabled a specific menu item, `IsEnabled()` could return a value of `false` for that item.

Disabling and enabling a menu

When an entire menu is disabled, its name appears dim in the menubar. Clicking on the menu opens the menu and displays its items, but each of the items will be disabled. Now that you know of the `BMenuItem()` function `SetEnabled()`, it should come as no surprise that there is also a `BMenu` version of this routine. Passing a value of `false` to a menu object's `SetEnabled()` function disables the entire menu. If a program keeps track of an Audio menu in a `BMenu` data member named `fAudioMenu`, you could disable that menu with just this line of code:

```
fAudioMenu->SetEnabled(false);
```

The current state of a menu can be found by invoking the item's `IsEnabled()` function. Like the `BMenuItem` version, the `BMenu` version of this routine returns a value of `true` if the object is presently enabled.

A menu's state usually changes in response to a message. If that's the case, include the menu enabling or disabling code in `MessageReceived()`.

Menu and Menu Item Keyboard Access

Menu items can be accessed by the mouse, of course, but they can also be accessed from the keyboard.

Shortcut keys

This chapter's "Creating a Menu Item" section demonstrated how to assign a new `BMenuItem` a keyboard shortcut. Here the third and fourth arguments to the `BMenuItem` constructor set the new menu item to have a keyboard shortcut of Command-M:

```
menu->AddItem(new BMenuItem("Command", new BMessage(A_MSG), 'M',
                              B_COMMAND_KEY));
```

In most instances the above method works fine for establishing a keyboard shortcut. However, your program may want to assign a keyboard shortcut on the fly. This is particularly true if your program gives the user the power to modify menu item keyboard shortcuts. If an existing menu item doesn't have a keyboard shortcut, or your program needs to change its currently defined keyboard shortcut, invoke the item's `SetShortcut()` function. The two arguments to this routine are identical to the third and fourth arguments that can be passed to the `BMenuItem` constructor. Here the keyboard shortcut for an Open menu item is being set to Command-Shift-O:

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Open");
theItem->SetShortcut('O', B_COMMAND_KEY | B_SHIFT_KEY)
```

To determine the current keyboard shortcut of a menu item, invoke the item's `Shortcut()` function. Pass this function a pointer to a 32-bit unsigned integer variable. `Shortcut()` will fill this variable with a mask that consists of all of the modifier keys that are a part of the shortcut for the menu item. `Shortcut()` will also return the shortcut character for the menu item:

```
uint32  *shortcutModifiers;
char    shortcutChar;

shortcutChar = theItem->Shortcut(shortcutModifiers);
```

To determine which modifier key or keys are a part of the shortcut, perform a bitwise AND on the `uint32` parameter. In the next snippet, a menu item named Calculate is examined to determine its shortcut key. If the character returned by `Shortcut()` is null, the item has no shortcut key. If the menu item does have a shortcut, the code goes on to determine which modifier keys are involved. Because all shortcut key combinations must include the Command key, no check is made to see if that key is a modifier. The code does, however, check to see if the Shift or Option keys are included in the shortcut key combination:

```
BMenuItem  *theItem;
uint32     *shortcutModifiers;
char       shortcutChar;
bool       hasShortcut = true;
bool       shiftKeyModifier = false;
bool       optionKeyModifier = false;

theItem = fMenuBar->FindItem("Calculate");
shortcutChar = theItem->Shortcut(shortcutModifiers);

if (shortcutChar == '\0')          // menu item doesn't have a shortcut key
   hasShortcut = false;
else {                             // menu item has a shortcut key
   if (*shortcutModifiers & B_SHIFT_KEY)
      shiftKeyModifier = true;
   if (*shortcutModifiers & B_OPTION_KEY)
      optionKeyModifier = true;
}
```

Keyboard triggers

A menu item can optionally be supplied with a shortcut key to benefit users who prefer the keyboard over the mouse. But every menu item is supplied with a *trigger* for the same reason. A trigger is a single character the user types in order to select a menu item. A trigger differs from a shortcut key in two ways. First, the trigger doesn't involve the use of a modifier key—simply pressing the trigger key is enough to choose the menu item. The second difference is that the trigger works only when the menu that holds the item in question is open, or dropped. Once a menu is open onscreen, the user can simply press a trigger key to select an item.

The trigger is one of the characters in the menu item name, and is indicated by being underlined. Typically, the trigger is the first character of one of the words that make up the menu item's name. Looking back at Figure 7-9 reveals two examples—there you see that the Audio menu's two items, Beep Once and Beep Twice, have triggers of "O" and "T," respectively.

Because a trigger can be used only on an open menu, different menus in the same menubar can have items with the same trigger. A menu item's trigger is assigned to the item by the system, so your program doesn't have to worry about which items end up with which triggers. If you do want responsibility for assigning a menu item a particular trigger, invoke that item's `SetTrigger()` function. Simply pass `SetTrigger()` the character that is to serve as the new trigger. Here a menu item named Jump is given a trigger of "u":

```
BMenuItem  *theItem;

theItem = fMenuBar->FindItem("Jump");
theItem->SetTrigger('u');
```

The character you pass to `SetTrigger()` must be either the menu item's shortcut key or a character in the menu item name. Failing both of these, the item will not be given a trigger—and whatever character had previously been assigned to the item won't be used either (so passing `SetTrigger()` an invalid character provides the exception to the rule that every menu item must have a trigger).

You can verify that a call to `SetTrigger()` worked according to plan by invoking the item's `Trigger()` function. Double-check to see if the above snippet works by following it with a call to `Trigger()`:

```
char  theTrigger;

theTrigger = theItem->Trigger();
```

With the exception of a menu item that's been given a trigger via a call to `SetTrigger()`, the system doesn't assign menu items triggers until runtime. That's done to avoid duplication of triggers within a menu. For this reason, calling `Trigger()` on a menu item that hasn't been manually assigned a trigger (by your project invoking the item's `SetTrigger()` function) serves little purpose—`Trigger()` will simply return `NULL` in such cases.

Figure 7-9 shows that the menu itself has a trigger—the "A" key serves as the trigger for the Audio menu. Like a menu item trigger, the system automatically assigns a trigger to a menu. Again like a menu item, your project can override the system-supplied trigger character. To do that, invoke the `BMenu` version of `SetTrigger()` on a menu object. Here the Audio menu is created and its trigger set to "U":

```
BMenu      *menu;
...
menu = new BMenu("Audio");
fMenuBar->AddItem(menu);
menu->SetTrigger('U');
// now add menu items to menu
```

Menu Characteristics Example Projects

In this chapter's example projects folder you'll find three projects that alter the characteristics of menus: TwoItemMenu, DisableMenuItem, and FindItemByMark. Each contains only a few lines of new code, so I'll forego thorough code walk-throughs and describe each only briefly.

Adding and altering menu items example project

The menu in each of this chapter's previous example projects consisted of just a single item. The TwoItemMenu project adds a second item. This project also adds a shortcut key to each item—Command-1 for the Beep Once item and Command-2 for the Beep Twice item. Figure 7-10 shows that the system has assigned each item a trigger that is the same character as that used in the item's shortcut key. Finally, the project demonstrates a menu set to radio mode—selecting one menu item checks that item and unchecks the other item.

Most of the code included in the TwoMenuItem project will be quite familiar to you. The code that's pertinent to the menu item topics in this section comes from the `MyHelloWindow` constructor:

```
BMenu      *menu;
BMenuItem  *menuItem;
...
menu = new BMenu("Audio");
```

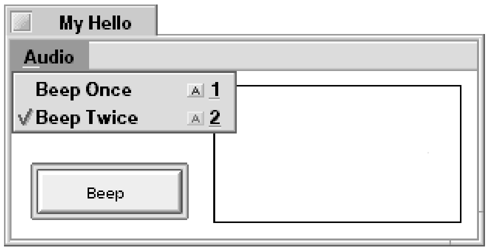


Figure 7-10. Menu items with shortcut keys

```
fMenuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG),
                            '1', B_COMMAND_KEY));
menu->AddItem(menuItem = new BMenuItem("Beep Twice", new BMessage(MENU_BEEP_
2_MSG),
                                       '2', B_COMMAND_KEY));
menuItem->SetMarked(true);
menu->SetRadioMode(true);
```

Menu item disabling and enabling example project

The DisableMenuItem project adds a few lines of code to the TwoMenuItem project to demonstrate how your program can toggle a menu item's state based on a message sent by a control. Clicking on the Beep button disables the Beep Once menu item in the Audio menu. Clicking on the Beep button again enables the same item. This menu-related code is found in the `MessageReceived() case` section for the message issued by the button control:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         // code to beep the appropriate number of times goes here

         BMenuItem  *theItem;

         theItem = fMenuBar->FindItem("Beep Once");
         if (theItem->IsEnabled())
            theItem->SetEnabled(false);
         else
            theItem->SetEnabled(true);
         break;

      case MENU_BEEP_1_MSG:
         fNumBeeps = 1;
         break;

      case MENU_BEEP_2_MSG:
         fNumBeeps = 2;
```

```
            break;

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

Accessing a menu item from a menu object

The preceding two projects use a `MyHelloWindow` class data member named `fNumBeeps` to keep track of how many times the system beep should sound in response to a click on the Beep button. The FindItemByMark project omits this data member, and doesn't keep track of which menu item is currently selected. Instead, it waits until the user clicks the Beep button before determining which menu item is currently marked. Clicking the Beep button results in the issuing of a `BUTTON_BEEP_MSG` that reaches the `MessageReceived()` function. Here the `BMenu` member function `FindMarked()` is used to find the currently checked menu item. Once the item object is obtained, its place in the menu is found by calling the `BMenu` function `IndexOf()`. A menu's items are indexed starting at 0, so adding 1 to the value returned by `IndexOf()` provides the number of beeps to play. The following snippet is from the `MessageReceived()` function of the project's `MyHelloWindow` class:

```
case BUTTON_BEEP_MSG:
   bigtime_t  microseconds = 1000000;  // one second
   int32      i;

   BMenuItem  *theItem;
   int32      itemIndex;
   int32      numBeeps;

   theItem = fAudioMenu->FindMarked();
   itemIndex = fAudioMenu->IndexOf(theItem);
   numBeeps = itemIndex + 1;

   for (i = 1; i <= numBeeps; i++)
   {
      beep();
      if (i != numBeeps)
         snooze(microseconds);
   }

   break;
```

Multiple Menus

Rather than jumping right into new topics, I'll provide a bit of a transition by presenting an example that includes two menus in its menubar. The example isn't entirely gratuitous, though—much of its code will reappear in upcoming discussions. Figure 7-11 shows the window, and the new Visual menu that's been added to the existing Audio menu, for the TwoMenus program. Choosing Draw Circles from the Visual menu draws a number of concentric circles in the window, while Draw Squares draws, yes, a number of concentric squares!

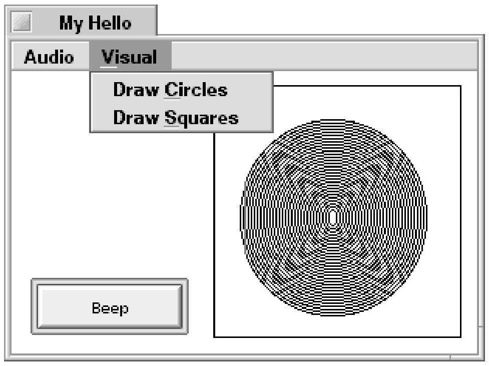


Figure 7-11. The TwoMenus application window

The *MyHelloWindow.h* header file in the TwoMenus project defines five message constants—one for the window's button and one for each of the four menu items.

```
#define   BUTTON_BEEP_MSG          'beep'
#define   MENU_BEEP_1_MSG          'bep1'
#define   MENU_BEEP_2_MSG          'bep2'
#define   MENU_DRAW_CIRCLES_MSG    'circ'
#define   MENU_DRAW_SQUARES_MSG    'squa'
```

The `MyHelloWindow` class holds four data members. `fMyView` is used for drawing the circles or squares, and `fNumBeeps` holds the number of times the system beep is to be played when the Beep button is clicked. Once the button and menubar are created, they aren't accessed outside of the `MyHelloWindow` constructor. Thus, I could have declared `BButton` and `BMenuBar` variables local to that routine rather than making each a data member. However, I've opted to set the project up from the start with the assumption that it will grow in complexity well beyond this trivial version. If I later need to add features that alter either the button or menu items (such as disabling and so forth), I'm all set.

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();
      virtual void      MessageReceived(BMessage* message);

   private:
      MyDrawView        *fMyView;
      BButton           *fButtonBeep;
```

```
        BMenuBar       *fMenuBar;
        int32          fNumBeeps;
};
```

Some of the `MyHelloWindow` constructor code is familiar to you, so I won't show the routine in its entirety. Here's the constructor without the view and button code:

```
MyHelloWindow::MyHelloWindow(BRect frame)
      : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   ...
   ...
   BMenu      *menu;
   BMenuItem  *menuItem;
   BRect      menuBarRect;

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

   menu = new BMenu("Audio");
   fMenuBar->AddItem(menu);
   menu->AddItem(new BMenuItem("Beep Once", new BMessage(MENU_BEEP_1_MSG)));
   menu->AddItem(menuItem = new BMenuItem("Beep Twice",
                                       new BMessage(MENU_BEEP_2_MSG)));
   menu->SetRadioMode(true);
   menuItem->SetMarked(true);
   fNumBeeps = 2;

   menu = new BMenu("Visual");
   fMenuBar->AddItem(menu);
   menu->AddItem(new BMenuItem("Draw Circles",
                            new BMessage(MENU_DRAW_CIRCLES_MSG)));
   menu->AddItem(new BMenuItem("Draw Squares",
                            new BMessage(MENU_DRAW_SQUARES_MSG)));
   Show();
}
```

The `MessageReceived()` routine handles each of the five types of messages the window might receive. A selection of either of the items from the Visual menu results in the application-defined function `SetViewPicture()` being called, followed by a call to the `BView` routine `Invalidate()` to force the view to update.

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case BUTTON_BEEP_MSG:
         // beep fNumBeeps times
         break;

      case MENU_BEEP_1_MSG:
         fNumBeeps = 1;
```

```
            break;

         case MENU_BEEP_2_MSG:
            fNumBeeps = 2;
            break;

         case MENU_DRAW_CIRCLES_MSG:
            fMyView->SetViewPicture(PICTURE_CIRCLES);
            fMyView->Invalidate();
            break;

         case MENU_DRAW_SQUARES_MSG:
            fMyView->SetViewPicture(PICTURE_SQUARES);
            fMyView->Invalidate();
            break;

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

The drawing code could have been kept in the `MessageReceived()` function, but I've decided to place it in a `MyDrawView` member function in order to keep `MessageReceived()` streamlined. The `SetViewPicture()` function defines a `BPicture` object based on the `int32` argument passed to the routine. The value of that argument is in turn based on the menu item selected by the user:

```
void MyDrawView::SetViewPicture(int32 pictureNum)
 {
   BRect  aRect;
   int32  i;

   switch (pictureNum)
   {
      case PICTURE_SQUARES:
         BeginPicture(fPicture);
            aRect.Set(15.0, 20.0, 140.0, 150.0);
            for (i = 0; i < 30; i++) {
               aRect.InsetBy(2.0, 2.0);
               StrokeRect(aRect);
         }
         fPicture = EndPicture();
         break;

      case PICTURE_CIRCLES:
         BeginPicture(fPicture);
            aRect.Set(15.0, 20.0, 140.0, 150.0);
            for (i = 0; i < 30; i++) {
               aRect.InsetBy(2.0, 2.0);
               StrokeEllipse(aRect);
            }
         fPicture = EndPicture();
         break;
   }
}
```

The picture defined in `SetViewPicture()` is a new data member that's been added to the `MyDrawView` class. Here you see the `BPicture` data member and the newly added declaration of the `SetViewPicture()` function:

```
#define   PICTURE_SQUARES   1
#define   PICTURE_CIRCLES   2


class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);
      void           SetViewPicture(int32 pictureNum);

   private:
      BPicture       *fPicture;
};
```

The whole purpose of storing the circles or squares `BPicture` object in a `MyDrawView` data member is so that the picture will be automatically updated whenever the view it's drawn in needs updating. That's accomplished by adding a call to the `BView` function `DrawPicture()` to the `MyDrawView` member function `Draw()`:

```
void MyDrawView::Draw(BRect)
{
   BRect  frame = Bounds();

   StrokeRect(frame);
   DrawPicture(fPicture);
}
```

Pop-up Menus

A pop-up menu is a menu that exists within the content area of a window rather than within a menubar. The pop-up menu can be positioned anywhere in a window (or anywhere in a view in a window). Like a menu in a menubar, a pop-up menu's content is displayed when the user clicks on the menu. Figure 7-12 shows a pop-up menu, both before and after being clicked, that holds two items.

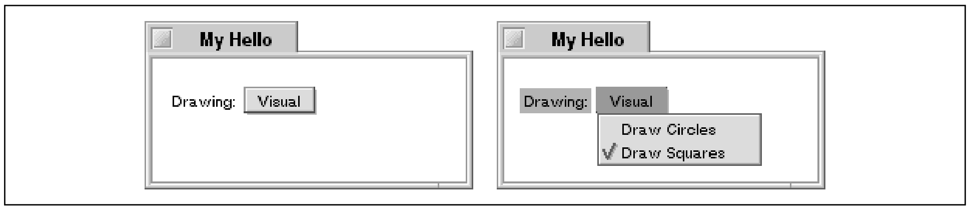


Figure 7-12. An example of a pop-up menu

A pop-up menu's default state is radio mode—the most recently selected item in the menu appears checked when the menu pops up. Figure 7-12 illustrates this for a menu with two items in it. A pop-up menu is most often in radio mode, so such a menu should be used to hold a related set of options. If the menu is to hold items that aren't directly related to one another, the menu should be housed within a menubar rather than existing as a pop-up menu. A context-sensitive pop-up menu is an exception—it behaves like a normal menu, albeit one that is not tied to a specific location in a window.

The BPopUpMenu Class

A pop-up menu is an object of the class `BPopUpMenu`. The `BPopUpMenu` class is derived from a class you've already studied—the `BMenu` class. Here's the `BPopUpMenu` constructor:

```
BPopUpMenu(const char  *name,
           bool        radioMode = true,
           bool        labelFromMarked = true,
           menu_layout layout = B_ITEMS_IN_COLUMN)
```

Like any menu, a pop-up menu has a name. The name is defined by the first `BPopUpMenu` constructor parameter and is present on the pop-up menu when the menu initially appears in a window. This pop-up menu name, however, can be changed to reflect the user's selection from the menu. The `labelFromMarked` parameter determines if that is to be the case.

The value of the `radioMode` parameter sets the pop-up menu's radio mode setting. By default, `radioMode` has a value of `true`. A value of `true` here means the same as it does for any other menu: choosing one item checks that item and unchecks whatever item was previously selected. The pop-up menu's radio mode value can be toggled by calling the `BMenu` function `SetRadioMode()`.

If the `labelFromMarked` parameter is set to `true`—as it is by default—the user's menu item choice from the pop-up menu determines the name the pop-up menu takes on. The original name won't reappear during the life of the window to which the pop up is attached. In Figure 7-12, for instance, the pop-up menu's name is Visual. If the user chooses, say, the Draw Squares item, the pop-up menu's name will change to Draw Squares. Setting `labelFromMarked` to `true` has the interesting side effect of automatically setting the pop-up menu to radio mode. That is, regardless of the value passed as the `radioMode` parameter of the `BPopUpMenu` constructor, the menu will be set to radio mode. If `labelFromMarked` is `false`, the pop-up menu's name will be fixed at its initial name (as defined by the value passed in as the first parameter) and its radio mode state will be determined by the value of the `radioMode` parameter.

The last BPopUpMenu parameter defines the layout of the pop-up menu. By default, a pop-up menu's items appear in a column—just like a menu held in a menubar. To instead have the items appear in a row, replace the B_ITEMS_IN_COLUMN value with another Be-defined constant: B_ITEMS_IN_ROW.

The BMenuField Class

A pop-up menu won't be placed in a menubar, so it doesn't have to be added to a BMenuBar object. A pop-up menu does, however, need to be added to an object capable of controlling the menu. The BMenuField class exists for this purpose. When a BMenuField object is created, a BPopUpMenu object is associated with it. Here's the BMenuField constructor:

```
BMenuField(BRect       frame,
           const char  *name,
           const char  *label,
           BMenu       *menu,
           uint32      resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
           uint32      flags = B_WILL_DRAW | B_NAVIGABLE)
```

The BMenuField is derived from the BView class. When a BMenuField object is created, four of the six BMenuField constructor parameters (frame, name, resizingMode, and flags) are passed on to the BView constructor.

The frame is a rectangle that defines the size of the BMenuField, which includes both a label and a pop-up menu. In Figure 7-12, you saw an example of a BMenuField object that has a label of "Drawing:" and a menu with the name "Visual." Recall that the source of the menu's name is the name parameter of the BPopUpMenu object. The BMenuField name parameter serves as a name for the BMenuField view, and isn't displayed onscreen. The resizingMode parameter specifies how the BMenuField is to be resized as its parent view is resized. The default value of B_FOLLOW_LEFT | B_FOLLOW_TOP means that the distance from the menu field's left side and its parent's parent's left side will be fixed, as will the distance from the menu field's top and its parent's top. The flags parameter specifies the notification the menu field is to receive. The default flags value of B_WILL_DRAW | B_NAVIGABLE means that the menu field view contains drawing, and should thus be subject to automatic updates, and that the menu field is capable of receiving and responding to keyboard input.

The BMenuField label parameter defines an optional label for the menu field. If you pass a string here, that string is displayed to the left of the pop-up menu that is a part of the menu field. To omit a label, pass NULL as the label parameter.

The menu parameter specifies the pop-up menu that is to be controlled by the menu field. While the class specified is BMenu, it is most likely that you'll pass a BPopUpMenu object here (BPopUpMenu is derived from BMenu, so it can be used).

Creating a Pop-up Menu

To create a pop-up menu, you first create a BPopUpMenu object, and then create a BMenuField object. Three of the four BPopUpMenu constructor parameters have default values, and those values generally suffice when creating a pop-up menu object—so creating the BPopUpMenu object often involves passing only a single argument to the BPopUpMenu constructor. Here a BPopUpMenu object for a pop-up menu named "Visual" is being created:

```
BPopUpMenu  *popUpMenu;

popUpMenu = new BPopUpMenu("Visual");
```

For a "regular" menu—one that resides in a menubar—the next step would typically be to add the new menu object to the existing menubar object with a call to AddItem(). A pop-up menu won't be placed in a menubar, so the above step is unnecessary. The pop-up menu does, however, need to be added to a menu field. The next steps are to create a BMenuField object that incorporates the BPopUpMenu object and then add this new menu field to a view (or window):

```
BMenuField  *menuField;
BRect       popUpMenuRect(10.0, 40.0, 105.0, 70.0);

menuField = new BMenuField(popUpMenuRect, "VisualPopUp", "Drawing",
                           popUpMenu);
AddChild(menuField);
```

The third argument to the BMenuField constructor specifies that the menu field have a label of "Drawing." In the previous snippet, the sole argument to the BPopUpMenu constructor specified that the pop-up menu have the name "Visual." The result of executing the above two snippets would be the menu field shown on the left side of Figure 7-12. The pop-up menu would be devoid of any items. To add a menu item to a pop up, have the pop-up menu invoke the BMenu function AddItem(). BPopUpMenu is derived from BMenu, and BPopUpMenu doesn't override the BMenu version of AddItem()—so adding menu items to a pop-up menu is handled in the exact same way as adding menu items to a "normal" menu that resides in a menubar. Here two items are added to the pop-up menu that was just created:

```
popUpMenu->AddItem(new BMenuItem("Draw Circles",
                                 new BMessage(MENU_DRAW_CIRCLES_MSG)));
popUpMenu->AddItem(new BMenuItem("Draw Squares",
                                 new BMessage(MENU_DRAW_SQUARES_MSG)));
```

At this point the menu field, and the pop-up menu that is a part of the menu field, match those shown on the right of Figure 7-12.

One reason the menu field label exists is to provide the user with information regarding the purpose of the menu field's pop-up menu. Once the user chooses an item from the pop-up menu, the pop-up menu's name disappears, so the menu field label may then be of help. If the contents of the pop-up menu make the pop-up menu's purpose obvious, you may choose to forego the menu field label. To do that, simply pass `NULL` as the third argument to the `BMenuField` constructor. Compare this `BMenuField` object creation with the one created a couple of snippets back:

```
menuField = new BMenuField(popUpMenuRect, "VisualPopUp", NULL, popUpMenu);
```

The left side of Figure 7-13 shows how the menu field looks now. The middle part of the figure shows a menu item selection being made, while the right side of the figure shows how the menu field looks after choosing an item.

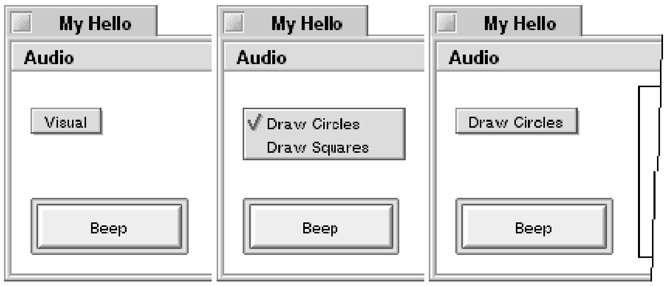


Figure 7-13. A menu field before, during, and after a menu item selection

You'll find the code that generates the window shown in Figure 7-13 in the MenuAndPopup project. The code varies little from that shown in the coverage of this chapter's TwoMenus project. But the `MyHelloWindow` constructors are different. The MenuAndPopup project uses the following code for adding a second menu to the menubar:

```
BMenuField   *menuField;
BPopUpMenu   *popUpMenu;
BRect        popUpMenuRect(10.0, 40.0, 120.0, 70.0);
const char   *popUpName = "VisualPopUp";
const char   *popUpLabel = NULL;

popUpMenu = new BPopUpMenu("Visual");
menuField = new BMenuField(popUpMenuRect, popUpName, popUpLabel, popUpMenu);
AddChild(menuField);
popUpMenu->AddItem(new BMenuItem("Draw Circles",
                                  new BMessage(MENU_DRAW_CIRCLES_MSG)));
popUpMenu->AddItem(new BMenuItem("Draw Squares",
                                  new BMessage(MENU_DRAW_SQUARES_MSG)));
```

Altering the Label/Pop-up Menu Divider

While no physical vertical line divides a menu field's label from its pop-up menu, there is indeed a defined boundary. By default, half of a menu field's width is devoted to the label, and half is assigned to the menu. Consider this snippet:

```
BPopUpMenu  *popUpMenu;

popUpMenu = new BPopUpMenu("Click Here");
popUpMenu->AddItem(new BMenuItem("Small", new BMessage(SML_MSG)));
popUpMenu->AddItem(new BMenuItem("Medium", new BMessage(MED_MSG)));
popUpMenu->AddItem(new BMenuItem("Large", new BMessage(LRG_MSG)));
popUpMenu->AddItem(new BMenuItem("Extra Large", new BMessage(XLG_MSG)));

BMenuField  *menuField;
BRect       popUpMenuRect(30.0, 25.0, 150.0, 50.0);

menuField = new BMenuField(popUpMenuRect, "PopUp", "Size", popUpMenu);
AddChild(menuField);
```

This code creates a pop-up menu and adds it to a menu field. The `BMenuField` object has a width of 120 pixels (150.0 – 30.0). Thus the menu field divider, which is given in coordinates local to the menu field's view, would be 60.0. As shown in the top two windows of Figure 7-14, this 1:1 ratio isn't always appropriate. Here the menu field label "Size" requires much less space than the pop-up menu name of "Click Here." Because the pop-up menu is constrained to half the menu field width, the pop-up name is automatically condensed—as is the "Extra Large" menu item after it is selected.

To devote more or less of a menu field to either the label or pop-up menu, use the `BMenuField` function `SetDivider()`. Pass this routine a floating-point value to be used as the new divider. This one argument should be expressed in coordinates local to the menu field. Consider our current example, which produces a menu field with a width of 120 pixels. To move the divider from its halfway point of 60 pixels to 30 pixels from the left edge of the menu field, pass a value of 30.0 to `SetDivider()`:

```
menuField->SetDivider(30.0);
```

The bottom two windows in Figure 7-14 show how the menu field looks after moving its divider. The pop-up menu now starts 30 pixels from the left of the menu field—just a few pixels to the left of the "Size" label. The pop-up menu now has room to expand horizontally; rather than being limited to 60 pixels in width, the menu can now occupy up to 90 of the menu field's 120 pixels.

You could use trial and error to find the amount of room appropriate for your pop up's label. But of course you'll instead rely on the `BView` function `StringWidth()`—the `BMenuField` class is derived from the `BView` class, so any

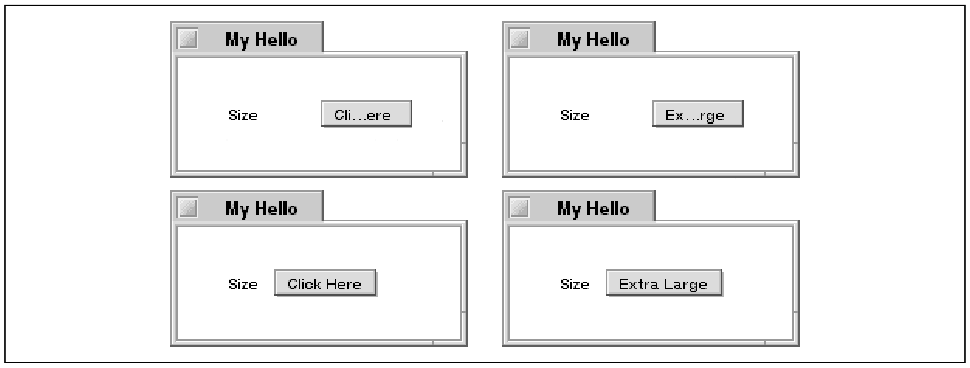


Figure 7-14. A menu field with its default divider (top) and an adjusted divider (bottom)

view member function can be invoked by a pop-up menu object. When passed a string, `StringWidth()` returns the number of pixels that string requires (based on the characteristics of the font currently used by the `BMenuField` object). For instance:

```
#define     LABEL_MARGIN5.0
float       labelWidth;

labelWidth = menuField->StringWidth("Size");
menuField->SetDivider(labelWidth + LABEL_MARGIN);
```

The above snippet determines the width of the string "Size" (the label used in the previous snippets), then uses that pixel width in setting the width of the space used to hold the label. Because the label starts a few pixels in from the left edge of the area reserved for the label, a few pixels are added as a margin, or buffer. If that wasn't done, the divider would be placed somewhere on the last character in the label, cutting a part of it off.

Submenus

A menu item can act as a *submenu*, or *hierarchical menu*—a menu within a menu. To operate a submenu, the user simply clicks on the submenu name, exposing a new menu of choices. To choose an item from the submenu, the user keeps the mouse button held down, slides the cursor onto the item, and releases the mouse button. Figure 7-15 provides an example of a submenu. Here a separator item and a submenu have been added to the Visual menu that was introduced in this chapter's TwoMenus project. The Number of Shapes submenu consists of three items: 10, 20, and 30.

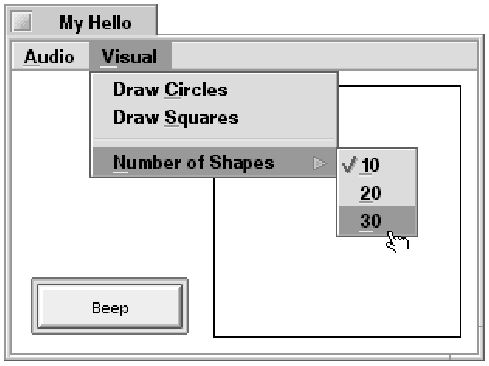


Figure 7-15. An example of a submenu

Creating a Submenu

A submenu is nothing more than a BMenu object that is added to another BMenu object in place of a menu item. Consider an Animals menu that has five types of animals for its menu items: armadillo, duck, labrador, poodle, and shepherd. Because three of the five animal types fall into the same category—dogs—this example would be well served by grouping the three dog items into a submenu. Figure 7-16 shows what the Animals menu would look like with a submenu, and this next snippet shows the code needed to produce this menu:

```
BMenu     *menu;
BMenu     *subMenu;

menu = new BMenu("Animals");
menuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Armadillo", new BMessage(ARMADILLO_MSG)));
subMenu = new BMenu("Dogs");
menu->AddItem(subMenu);
subMenu->AddItem(new BMenuItem("Labrador", new BMessage(LAB_MSG)));
subMenu->AddItem(new BMenuItem("Poodle", new BMessage(POODLE_MSG)));
subMenu->AddItem(new BMenuItem("Shepherd", new BMessage(SHEPHERD_MSG)));
menu->AddItem(new BMenuItem("Duck", new BMessage(DUCK_MSG)));
```

Notice in this snippet that while I've given the variable used to represent the submenu the name subMenu, it really is nothing more than a BMenu object. The items in the Dogs submenu were added the same way as the items in the Animal menu—by invoking the BMenu member function AddItem().

Submenu Example Project

The MenusAndSubmenus project builds an application that displays the window shown back in Figure 7-15. Most of the code in this project comes from the

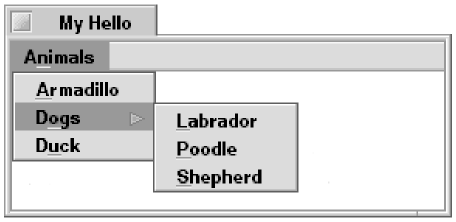


Figure 7-16. Categorizing things using a menu and submenu

TwoMenus project, along with new code supporting the new submenu. The `MyHelloWindow` class now holds a new `int32` data member named `fNumShapes` that keeps track of the number of circles or squares that should be used when Draw Circles or Draw Squares is selected:

```
class MyHelloWindow : public BWindow {
   ...
   ...
   private:
      MyDrawView     *fMyView;
      BButton        *fButtonBeep;
      BMenuBar       *fMenuBar;
      int32          fNumBeeps;
      int32          fNumShapes;
};
```

The `MyHelloWindow` constructor includes new code that adds a separator item to the Visual menu and creates and initializes the Number of Shapes submenu that's now housed as the last item in the Visual menu. Here's a part of the `MyHelloWindow` constructor:

```
BMenu      *menu;
BMenu      *subMenu;
BMenuItem  *menuItem;

// create menubar and add to window

// create Audio menu, add to menubar, add items to it, set
// to radio mode and mark one item

menu = new BMenu("Visual");
fMenuBar->AddItem(menu);
menu->AddItem(new BMenuItem("Draw Circles",
                             new BMessage(MENU_DRAW_CIRCLES_MSG)));
menu->AddItem(new BMenuItem("Draw Squares",
                             new BMessage(MENU_DRAW_SQUARES_MSG)));

menu->AddSeparatorItem();

subMenu = new BMenu("Number of Shapes");
menu->AddItem(subMenu);
```

```
subMenu->AddItem(menuItem = new BMenuItem("10",
                                new BMessage(MENU_10_SHAPES_MSG)));
subMenu->AddItem(new BMenuItem("20", new BMessage(MENU_20_SHAPES_MSG)));
subMenu->AddItem(new BMenuItem("30", new BMessage(MENU_30_SHAPES_MSG)));

subMenu->SetRadioMode(true);
menuItem->SetMarked(true);
fNumShapes = 10;
```

As shown, a submenu can be set to radio mode, and an item in the submenu can be marked, just as is done for a menu that's added to a menubar.

The `MessageReceived()` function needs three new `case` sections—one to handle each of the three new messages that result from the submenu item selections:

```
case MENU_10_SHAPES_MSG:
   fNumShapes = 10;
   break;

case MENU_20_SHAPES_MSG:
   fNumShapes = 20;
   break;

case MENU_30_SHAPES_MSG:
   fNumShapes = 30;
   break;
```

Two of the existing `case` sections in `MessageReceived()` need modification. Now the number of shapes to use in the drawing of the concentric circles or squares gets passed to the `MyDrawView` member function `SetViewPicture()`:

```
case MENU_DRAW_CIRCLES_MSG:
fMyView->SetViewPicture(PICTURE_CIRCLES, fNumShapes);
   fMyView->Invalidate();
   break;

case MENU_DRAW_SQUARES_MSG:
fMyView->SetViewPicture(PICTURE_SQUARES, fNumShapes);
   fMyView->Invalidate();
   break;
```

The `MyDrawView` member function `SetViewPicture()` makes use of the new parameter as the index that determines how many times `InsetRect()` and `StrokeRect()` are called.

8

In this chapter:
- *Fonts*
- *Simple Text*
- *Editable Text*
- *Scrolling*

Text

The BeOS makes it simple to display text in a view—you've seen several examples of calling the BView functions SetFont() and DrawString() to specify which font a view should use and then draw a line of text. This approach works fine for small amounts of plain text; your application, however, is more likely to be rich in both graphics and text—so you'll want to take advantage of the BFont, BStringView, BTextView, BScrollBar, and BScrollView classes.

The BFont class creates objects that define the characteristics of fonts. You create a BFont object based on an existing font, then alter any of several characteristics. The BeOS is quite adept at manipulating fonts. You can alter basic font features such as size and spacing, but you can also easily change other more esoteric font characteristics such as shear and angle of rotation. You can use this new font in subsequent calls to DrawString(), or as the font in which text is displayed in BStringView, BTextView, or BScrollView objects.

A BStringView object displays a line of text, as a call to the BView function DrawString() does. Because the text of a BStringView exists as an object, this text knows how to update itself—something that the text produced by a call to DrawString() doesn't know how to do.

More powerful than the BStringView class is the BTextView class. A BTextView object is used to display small or large amounts of editable text. The user can perform standard editing techniques (such as cut, copy, and paste) on the text of a BTextView object. And the user (or the program itself) can alter the font or font color of some or all of the text in such an object.

If the text of a BTextView object extends beyond the content area of the object, a scrollbar simplifies the user's viewing. The BScrollBar class lets you add a scrollbar to a BTextView. Before adding that scrollbar, though, you should consider creating a BScrollView object. As its name implies, such an object has built-in support for scrollbars. Create a BTextView object to hold the text, then create a BScrollView object that names the text view object as the scroll view's target. Or, if you'd like to scroll graphics rather than text, name a BView object as the target and then include a BPicture in that BView. While this chapter's focus is on text, it does close with an example adding scrollbars to a view that holds a picture.

Fonts

In the BeOS API, the BFont class defines the characteristics of a font—its style, size, spacing, and so forth. While the BFont class has not been emphasized in prior chapters, it has been used throughout this book. Every BView object (and thus every BView-derived object) has a current font that affects text displayed in that view. In previous examples, the BView-derived MyDrawView class used its AttachedToWindow() function to call a couple of BView functions to adjust the view's font: SetFont() to set the font, and SetFontSize() to set the font's size:

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(24);
}
```

A view's current font is used in the display of characters drawn using the BView function DrawString(). Setting a view's font characteristics in the above fashion affects text produced by calls to DrawString() in each MyDrawView object.

The above snippet illustrates that the examples to this point have done little to alter the look of a font. Making more elaborate modifications is an easy task. Later in this chapter, you'll use some of the following techniques on text displayed in text view objects—editable text objects based on the BTextView class.

System Fonts

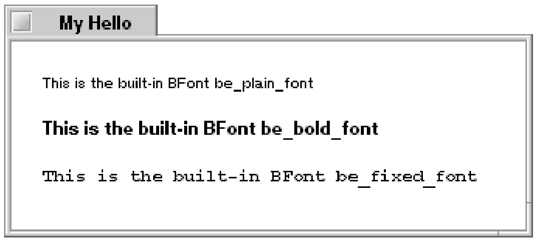
When designing the interface for your application, you'll encounter instances where you want a consistent look in displayed text. For example, your application may have a number of windows that include instructional text. In such a case, you'll want the text to have the same look from window to window. To ensure that your application can easily do this, the BeOS defines three fonts guaranteed to exist and remain constant for the running of your application.

The three global system fonts

The three constant fonts, or *global system fonts*, are BFont objects. When an application launches, these BFont objects are created, and three global pointers are assigned to reference them. Table 8-1 shows these global BFont objects. Figure 8-1 shows a window running on my machine; the figure includes a line of text written in each of the three system fonts.

Table 8-1. Global Fonts and Their Usage

BFont Global Pointer	Common Font Usage
be_plain_font	Controls, such as checkboxes and buttons, have their labels displayed in this font. Menu items also appear in this font.
be_bold_font	Window titles appear in this font.
be_fixed_font	This font is used for proportional, fixed-width characters.



This is the built-in BFont be_plain_font

This is the built-in BFont be_bold_font

```
This is the built-in BFont be_fixed_font
```

Figure 8-1. An example of text produced from the three global fonts

Contradictory as it sounds, the user can change the font that's used for any of the global system fonts. Figure 8-2 shows that the FontPanel preferences program lets the user pick a different plain, bold, or fixed font. This means that your application can't count on a global font pointer (such as be_plain_font) always representing the same font on all users' machines. You can, however, count on a global font pointer to always represent only a single font on any given user's machine—regardless of which font that is. So while you may not be able to anticipate what font the user will view when you make use of a global font pointer in your application, you are assured that the user will view the same font each time that global font pointer is used by your application.

Using a global system font

You've already seen how to specify one of the global fonts as the font to be used by a particular view: just call the BView function SetFont() within one of the view's member functions. The AttachedToWindow() snippet that appears above provides an example. That method initializes all of the objects of a particular class to use the same font. In the above example, all MyDrawView objects will initially display text in the font referenced by be_bold_font. For a particular view to have its current font set to a different system font, have that view call SetFont() after the view has been created:

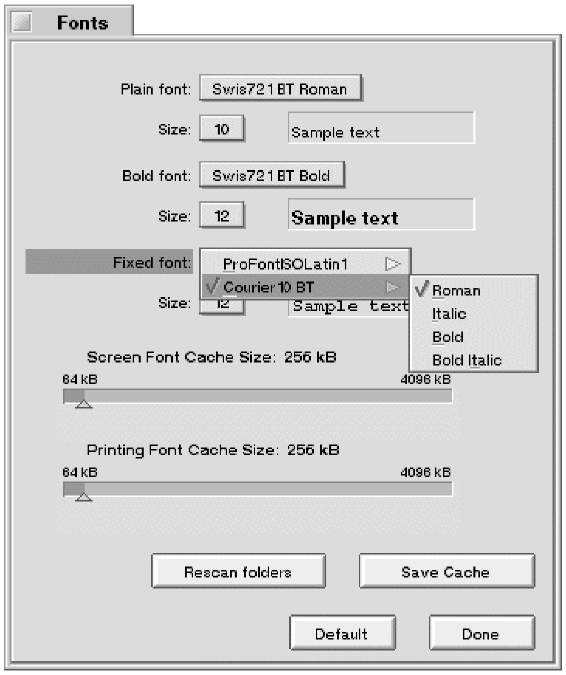

Plain font: Swis721 BT Roman
Size: 10
Bold font: Swis721 BT Bold
Size: 12
Fixed font: ✓ Courier 10 BT, ✓ Roman
Size: 12
Screen Font Cache Size: 256 kB
Printing Font Cache Size: 256 kB

Figure 8-2. The FontPanel preferences application window

```
    MyDrawView  *theDrawView;

    theDrawView = new MyDrawView(frameRect, "MyDrawView");
    theDrawView->SetFont(be_plain_font);
```

While a BeOS machine may have more than the three system fonts installed, your application shouldn't make any font-related assumptions. You can't be sure every user has a non-system font your application uses; some users may experience unpredictable results when running your application. If you want your program to display text that looks different from the global fonts (such as a very large font like 48 points), you can still use a global font to do so, as the next section illustrates.

Your program shouldn't force the user to have a particular non-system font on his or her machine, but it can give the user the option of displaying text in a non-system font. Consider a word processor you're developing. The default font should be `be_plain_font`. But your application could have a Font menu that allows for the display of text in any font on the user's computer. Querying the user's machine for available fonts is a topic covered in the `BFont` section of the Interface Kit chapter of the Be Book.

Global fonts are not modifiable

A global font is an object defined to be constant, so it can't be altered by an application. If a program could alter a global font, the look of text in other applications would be affected. Instead, programs work with copies of global fonts. While calling a `BView` function such as `SetFontSize()` may seem to be changing the size of a font, it's not. A call to `SetFontSize()` simply specifies the point size at which to display characters. The font itself isn't changed—the system simply calculates a new size for each character and displays text using these new sizes. Consider this snippet:

```
MyDrawView  *drawView1;
MyDrawView  *drawView2;

drawView1 = new MyDrawView(frameRect1, "MyDrawView1");
drawView1->SetFont(be_bold_font);
drawView1->SetFontSize(24);

drawView2 = new MyDrawView(frameRect2, "MyDrawView2");
drawView2->SetFont(be_bold_font);

drawView1->MoveTo(20.0, 20.0);
drawView1->DrawString("This will be bold, 24 point text");

drawView2->MoveTo(20.0, 20.0);
drawView2->DrawString("This will be bold, 12 point text");
```

This code specifies that the `MyDrawView` object `drawView1` use the `be_bold_font` in the display of characters. The code also sets this object to display these characters in a 24-point size. The second `MyDrawView` object, `drawView2`, also uses the `be_bold_font`. When drawing takes place in `drawView1`, it will be 24 points in size. A call to `DrawString()` from `drawView2` doesn't result in 24-point text, though. That's because the call to `SetFontSize()` didn't alter the font `be_bold_font` itself. Instead, it only marked the `drawView2` object to use 24 points as the size of text it draws.

Making global fonts unmodifiable is a good thing, of course. Having a global font remain static means that from the time your application launches until the time it terminates, you can always rely on the font having the same look. Of course, there will be times when your application will want to display text in a look that varies from that provided by any of the three global fonts. That's the topic of the next section.

Altering Font Characteristics

If you want to display text in a look that doesn't match one of the system fonts, and you want to be able to easily reuse this custom look, create your own `BFont` object. Pass the BFont constructor one of the three global system fonts and the constructor will return a copy of it to your application:

```
BFont  theFont(be_bold_font);
```

The BFont object theFont is a copy of the font referenced by be_bold_font, so theFont can be modified. To do that, invoke the BFont member function appropriate for the characteristic to change. For instance, to set the font's size, call SetSize():

```
theFont.SetSize(15.0);
```

A look at the BFont class declaration in the *Font.h* BeOS API header file hints at some of the other modifications you can make to a BFont object. Here's a partial listing of the BFont class:

```
class BFont {
   public:
                  BFont();
                  BFont(const BFont &font);
                  BFont(const BFont *font);

      void        SetFamilyAndStyle(const font_family family,
                                  const font_style style);
      void        SetFamilyAndStyle(uint32 code);
      void        SetSize(float size);
      void        SetShear(float shear);
      void        SetRotation(float rotation);
      void        SetSpacing(uint8 spacing);
      ...
      void        GetFamilyAndStyle(font_family *family, font_style *style)
                     const;
      uint32      FamilyAndStyle() const;
      float       Size() const;
      float       Shear() const;
      float       Rotation() const;
      uint8       Spacing() const;
      uint8       Encoding() const;
      uint16      Face() const;
      ...
      float       StringWidth(const char *string) const;
      ...
}
```

For each member function that sets a font trait, there is a corresponding member function that returns the same trait. An examination of a few of these font characteristics provides a basis for understanding how fonts are manipulated.

Font size

An example of setting a BFont object's point size was shown above. An example of determining the current point size of that same BFont object follows.

```
float  theSize;

theSize = theFont.Size();
```

You've already seen that in order for a view to make use of a font, that font needs to become the view's current font. The BView function SetFont() performs that task. Numerous examples have demonstrated this routine's use in setting a view's font to one of the global system fonts, but you can use SetFont() with any BFont object. Here, one view is having its font set to the global system font be_plain_font, while another is having its font set to an application-defined BFont object:

```
BFont  theFont(be_bold_font);

theFont.SetSize(20.0);
drawView1->SetFont(&theFont);

drawView2->SetFont(be_plain_font);
```

This snippet demonstrates how to replace whatever font a view is currently using with another font—the drawView1 view was making use of some font before the call to SetFont(). There will be times when you won't want to replace a view's font, but rather simply alter one or more of the traits of the view's current font. To do that, call the BView function GetFont() to first get a copy of the view's current font. Make the necessary changes to this copy, then call SetFont() to make it the view's new current font. Here, a view's current font has its size changed:

```
BFont  theFont;

theDrawView->GetFont(&theFont);
theFont.SetSize(32.0);
theDrawView->SetFont(&theFont);
```

Font shear

A font's shear is the slope, or angle, at which the font's characters are drawn. Pass the BFont function SetShear() a value in degrees and the routine will use it to adjust the amount of slope characters have. The range of values SetShear() accepts is 45.0 to 135.0. As Figure 8-3 shows, this angle is relative to the baseline on which characters are drawn. You'll also note that the degrees are measured clockwise. A value of 45.0 produces the maximum slant to the left, while a value of 135.0 produces the maximum slant to the right. The following code generates the three strings shown in Figure 8-3:

```
BFont theFont(be_plain_font);

theFont.SetSize(24.0);
theFont.SetShear(45.0);
theView->SetFont(&theFont);
theView->MovePenTo(110.0, 60.0);
```

```
    theView->DrawString("Shear 45");

    theFont.SetShear(90.0);
    theView->SetFont(&theFont);
    theView->MovePenTo(110.0, 140.0);
    theView->DrawString("Shear 90");

    theFont.SetShear(135.0);
    theView->SetFont(&theFont);
    theView->MovePenTo(110.0, 220.0);
    theView->DrawString("Shear 135");
```

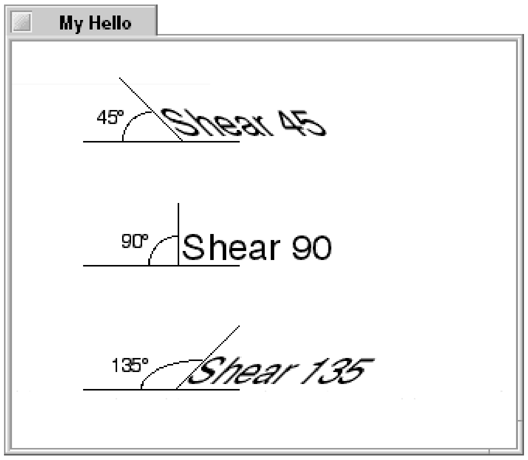


Figure 8-3. Output of text when the font's shear is varied

Font rotation

The `SetRotation()` function in the `BFont` class makes it easy to draw text that's rotated to any degree. Pass `SetRotation()` a value in degrees, and subsequent text drawn to the view will be rotated. The degrees indicate how much the baseline on which text is drawn should be rotated. Figure 8-4 shows that the angle is relative to the original, horizontal baseline. Degrees are measured clockwise: a positive rotation means that subsequent text will be drawn at an angle upward, while a negative rotation means that text will be drawn at an angle downward. This next snippet produces the text shown in the window in Figure 8-4:

```
    BFont theFont(be_plain_font);

    theFont.SetSize(24.0);
    theFont.SetRotation(45.0);
    theView->SetFont(&theFont);
    theView->MovePenTo(70.0, 110.0);
    theView->DrawString("Rotate 45");

    theFont.SetRotation(-45.0);
    theView->SetFont(&theFont);
    theView->MovePenTo(190.0, 110.0);
    theView->DrawString("Rotate -45");
```

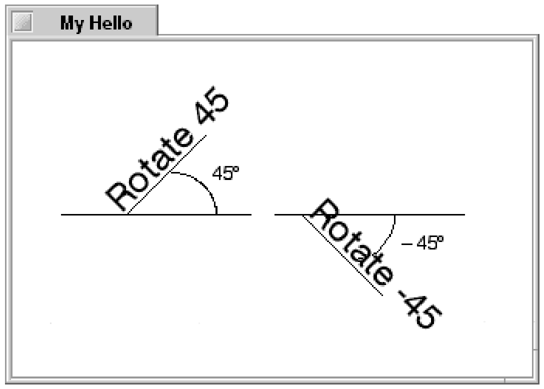


Figure 8-4. Output of text when the font's rotation is varied

Fonts Example Project

The FontSetting project demonstrates how to create `BFont` objects and use them as a view's current font. As Figure 8-5 shows, this example also demonstrates how to set the angle at which text is drawn, as well as how to rotate text.

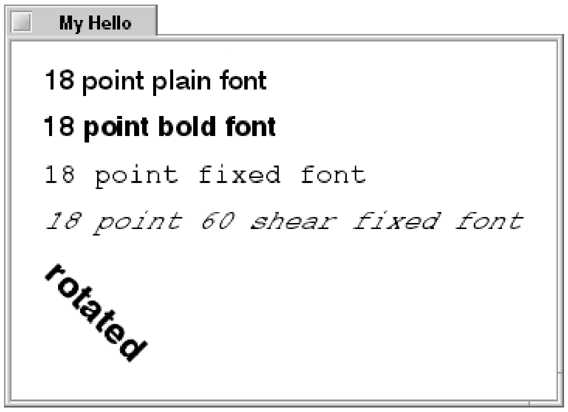


Figure 8-5. The FontSetting example program's window

I won't need a sophisticated program to show off a few of the things that can be done with fonts; a single menuless window will do. The FontSetting project's `MyHelloWindow` class has only one data member: the familiar drawing view `fMyView`. The `MyDrawView` class has no data members. Both the `MyDrawView` constructor and the MyDrawView function AttachedToWindow() are empty. The only noteworthy function is the MyDrawView routine Draw(), shown here:

```
void MyDrawView::Draw(BRect)
{
   SetFont(be_plain_font);
   SetFontSize(18);
   MovePenTo(20, 30);
   DrawString("18 point plain font");

   SetFont(be_bold_font);
   SetFontSize(18);
   MovePenTo(20, 60);
   DrawString("18 point bold font");

   SetFont(be_fixed_font);
   SetFontSize(18);
   MovePenTo(20, 90);
   DrawString("18 point fixed font");

   BFont font;
   GetFont(&font);
   font.SetShear(120.0);
   SetFont(&font);
   MovePenTo(20, 120);
   DrawString("18 point 60 shear fixed font");

   SetFont(be_bold_font);
   GetFont(&font);
   font.SetSize(24.0);
   font.SetRotation(-45.0);
   SetFont(&font);
   MovePenTo(20, 150);
   DrawString("rotated");
}
```

The code in Draw() falls into five sections, each section ending with a call to DrawString(). Each of the first three sections:

- Sets the view's font to one of the three system fonts
- Sets the view to draw text in 18-point size
- Moves the pen to the starting location for drawing
- Draws a string

To draw each of the first three lines of text in 18-point size, note that after each call to SetFont(), SetFontSize() needs to be called. That's because a call to SetFont() uses all of the characteristics of the passed-in font. Thus, the second call to SetFont()—the call that sets the drawing view to draw in be_bold_font—sets the view to draw text in whatever point size the user defines for the be_bold_font (defined for the bold font in the FontPanel preferences window).

The fourth code section demonstrates how to change one aspect of a view's current font without affecting the font's other attributes. A call to `GetFont()` returns a copy of the view's current font. A call to the `BFont` function `SetShear()` alters the shear of the font. A call to `SetFont()` then establishes this font as the view's new current font.

The final section of code provides a second example of changing some characteristics of a view's current font without overwriting all of its traits. Here the view's font is set to `be_bold_font`, a copy is retrieved, and the size and rotation of the copied font are changed. This new font is then used as the view's current font before drawing the string "rotated."

Simple Text

Throughout this book you've seen that you can draw a string in any view by invoking the `BView`'s `DrawString()` function. `DrawString()` is a handy routine because it's easy to use—just call `MovePenTo()` or `MovePenBy()` to establish the starting point for a string, then pass `DrawString()` the text to draw. Drawing text with `DrawString()` has one distinct shortcoming, though. Unless the call is made from within the view's `Draw()` function, the text drawn by `DrawString()` won't automatically be updated properly whenever all or part of the text comes back into view after being obscured. A call to `DrawString()` simply draws text—it doesn't create a permanent association between the text and the view, and it doesn't create any kind of string object with the power to update itself. The `BStringView` class exists to overcome these deficiencies.

A `BStringView` object draws a single line of text, just as `DrawString()` does. Unlike the `DrawString()` text, however, the `BStringView` object's text automatically gets updated whenever necessary. While the text displayed by the `BStringView` object can be changed during runtime (see the "Setting the text in a string" section ahead), it isn't user-editable. It also doesn't word-wrap, and it can't be scrolled. That makes a `BStringView` object ideal for creating simple, static text such as that used for a label, but undesirable for displaying large amounts of text or user-editable text. For working with more sophisticated text objects, refer to the description of the `BTextView` class in this chapter's "Editable Text" section.

The BStringView Class

Create a `BStringView` object by invoking the `BStringView` constructor. The `BStringView` class is derived from the `BView` class. In creating a new string view object, the `BStringView` constructor passes all but its `text` parameter on to the `BView` constructor:

```
BStringView(BRect frame,
            const char *name,
            const char *text,
            uint32 resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
            uint32 flags = B_WILL_DRAW)
```

The `frame` parameter is a rectangle that defines the boundaries of the view. The text displayed by the `BStringView` object won't word wrap within this rectangle, so it must have a width sufficient to display the entire string. The `name` parameter defines a name by which the view can be identified at any time. The `resizingMode` parameter specifies the behavior of the view in response to a change in the size of the string view's parent view. The `flags` parameter is a mask consisting of one or more Be-defined constants that determine the kinds of notifications the view is to respond to.

The `text` parameter establishes the text initially displayed by the `BStringView` object. The text can be passed between quotes or, as shown below, a variable of type `const char *` can be used as the `text` argument. After creating the string view object, call `AddChild()` to add the new object to a parent view:

```
BStringView  *theString;
BRect        stringFrame(10.0, 10.0, 250.0, 30.0);
const char   *theText = "This string will be automatically updated";

theString = new BStringView(stringFrame, "MyString", theText);
AddChild(theString);
```

For simplicity, this snippet hardcodes the string view's boundary. Alternatively, you could rely on the `StringWidth()` function to determine the pixel width of the string and then use that value in determining the coordinates of the view rectangle. In Chapter 7, *Menus*, this routine was introduced and discussed as a `BView` member function. Here you see that the `BFont` class also includes such a function. By default, a new `BStringView` object uses the `be_plain_font` (which is a global `BFont` object), so that's the object to use when invoking `StringWidth()`. Here, I've modified the preceding snippet to use this technique:

```
#define      FRAME_LEFT    10.0

BStringView  *theString;
BRect        stringFrame;
const char   *theText = "This string will be automatically updated";
float        textWidth;

textWidth = be_plain_font->StringWidth(theText);
stringFrame.Set(FRAME_LEFT, 10.0, FRAME_LEFT + textWidth, 30.0);

theString = new BStringView(stringFrame, "MyString", theText);
AddChild(theString);
```

Manipulating the Text in a String

Once a string view object is created, its text can be altered using a variety of BStringView member functions.

Setting the text in a string

The text of a BStringView object isn't directly editable by the user, but the program can change it. To do that, invoke the BStringView function SetText(), passing the new text as the only parameter. In the following snippet, the text of the string view object created in the previous snippet is changed from "This string will be automatically updated" to "Here's the new text":

```
theString->SetText("Here's the new text");
```

To obtain the current text of a string view object, call the BStringView member function Text():

```
const char  *stringViewText;

stringViewText = theString->Text();
```

Aligning text in a string

By default, the text of a BStringView object begins at the left border of the object's frame rectangle. You can alter this behavior by invoking the BStringView member function SetAlignment(). This routine accepts one of three Be-defined alignment constants: B_ALIGN_LEFT, B_ALIGN_RIGHT, or B_ALIGN_CENTER. Here the left-aligned default characteristic of the text of the BStringView object theString is altered such that it is now right-aligned:

```
theString->SetAlignment(B_ALIGN_RIGHT);
```

You can obtain the current alignment of a BStringView object's text by invoking the BStringView function Alignment(). This routine returns a value of type alignment. Unsurprisingly, the constants B_ALIGN_LEFT, B_ALIGN_RIGHT, and B_ALIGN_CENTER are of this type, so you can compare the returned value to one or more of these constants. Here, the alignment of the text in a BStringView object is checked to see if it is currently centered:

```
alignment  theAlignment;

theAlignment = theString->Alignment();

if (theAlignment == B_ALIGN_CENTER)
   // you're working with text that is centered
```

Changing the look of the text in the string

A new `BStringView` object's text is displayed in black and in the system plain font. A `BStringView` object is a `BView` object, so `BView` member functions such as `SetHighColor()`, `SetFont()`, and `SetFontSize()` can be invoked to change the characteristics of a string view object's text. Here, the color of the text of a `BStringView` object is changed from black to red by altering the string view's high color. The text's font and size are changed as well:

```
rgb_color  redColor = {255, 0, 0, 255};
theString->SetHighColor(redColor);
theString->SetFont(be_bold_font);
theString->SetFontSize(18);
```

You can make more sophisticated changes to the look of the text displayed in a `BStringView` object by creating a `BFont` object, modifying any of the font's characteristics (using the techniques shown in this chapter's "Fonts" section), and then using that font as the `BStringView` object's font. Here, the font currently used by a string view object is retrieved, its shear changed, and the altered font is again used as the string view object's font:

```
BFont theFont;

theString->GetFont(&theFont);
theFont.SetShear(100.0);
theString->SetFont(&theFont);
```

String View Example Project

The StringView project produces the window shown in Figure 8-6. The "Here's the new text" string is a `BStringView` object, so the text is automatically redrawn after the user obscures the window and then reveals it again. The Text menu holds a single item named `Test` that, when selected, does nothing more than generate a system beep. Subsequent examples in this chapter add to this menu.

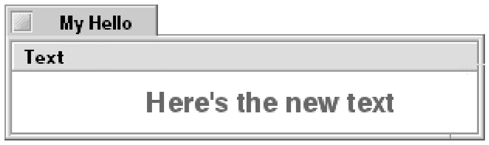


Figure 8-6. The StringView example program's window

The `BStringView` object will be added to the window's main view—the window's one `MyDrawView` object. To make it easy for you to manipulate the string later in the program, I keep track of the string by making it a data member in the `MyDrawView` class.

```
class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);

   private:
      BStringView    *fString;
};
```

The `BStringView` object's frame rectangle has a left boundary of 10 pixels. The `BStringView` object's parent view is the window's `fMyView` view. The width of the `MyDrawView` `fMyView` is the same as the window, so the default state for the `BStringView` text has the text starting 10 pixels from the left edge of the window. Figure 8-6 makes it clear that this isn't the starting point of the text. A call to `SetAlignment()` is responsible for this discrepancy—the string view object's text has been changed to right-aligned. The text's look has been changed from its default state by calling the `BView` functions `SetFont()` and `SetFontSize()`. You can't tell from Figure 8-6 that the text appears in red rather than black. It's a call to `SetHighColor()` that makes this color change happen. Here's the StringView project's `MyDrawView` constructor, which shows all the pertinent code:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect   stringFrame(10.0, 10.0, 250.0, 30.0);

   fString = new BStringView(stringFrame, "MyString",
                        "This string will be automatically updated");
   AddChild(fString);

   fString->SetText("Here's the new text");

   fString->SetAlignment(B_ALIGN_RIGHT);

   rgb_color  redColor = {255, 0, 0, 255};
   fString->SetHighColor(redColor);

   fString->SetFont(be_bold_font);
   fString->SetFontSize(18);
}
```

Editable Text

A `BStringView` object is ideal for displaying a small amount of uneditable text. When your application needs to display a larger amount of text that is user-editable, though, it's time to switch to a `BTextView` object. A `BTextView` object automatically implements keyboard editing, and makes it easy to add menu editing. And while a text view object initially displays all its text in a single font and a single color, you can easily alter the object to support multiple fonts and multiple colors—even within the same paragraph.

The BTextView Class

The `BTextView` class used to create an editable text object is derived from the `BView` class. So, as expected, several of the `BTextView` constructor parameters will be immediately familiar to you:

```
BTextView(BRect       frame,
          const char  *name,
          BRect       textRect,
          uint32      resizingMode,
          uint32      flags)
```

The `frame`, `name`, and `resizingMode` parameters serve the same purposes as they do for the `BView` class. The `flags` parameter is made up of one or more Be-defined constants that determine the kinds of notifications the view is to respond to. Regardless of which constant or constants you pass as the `flags` parameter, the `BTextView` constructor goes behind your back to add a couple more constants before forwarding `flags` to the `BView` constructor it invokes. These two `BTextView`-added constants are `B_FRAME_EVENTS`, to allow the `BTextView` object to reformat its text when it is resized, and `B_PULSE_NEEDED`, to allow the text insertion caret to blink properly.

The one `BTextView` constructor parameter unique to the `BTextView` class is `textRect`. This rectangle specifies the boundaries for the text that will eventually be placed in the `BTextView` object.

BTextView frame and text rectangles

At first glance, the purpose of the `BTextView` constructor's `textRect` rectangle may seem to be redundant—the `frame` parameter is also a boundary-defining rectangle. Here's the difference: the `frame` rectangle defines the size of the `BTextView` object, as well as where the `BTextView` object resides in its parent view. The `textRect` parameter defines the size of the text area within the `BTextView` object, and where within the `BTextView` object this text area is to be situated. By default, a `BTextView` object has a frame the size of the `frame` rectangle drawn around it. The `textRect` rectangle doesn't have a frame drawn around it. Thus, the `textRect` rectangle provides for a buffer, or empty space, surrounding typed-in text and the `BTextView` object's frame. Figure 8-7 illustrates this.

In Figure 8-7, the dark-framed rectangle represents the `frame` rectangle, the first parameter to the `BTextView` constructor. The light-framed rectangle represents the `textRect` rectangle. Neither of these rectangles would be visible to the user; I've shown them in the figure only to make it obvious where their boundaries are in

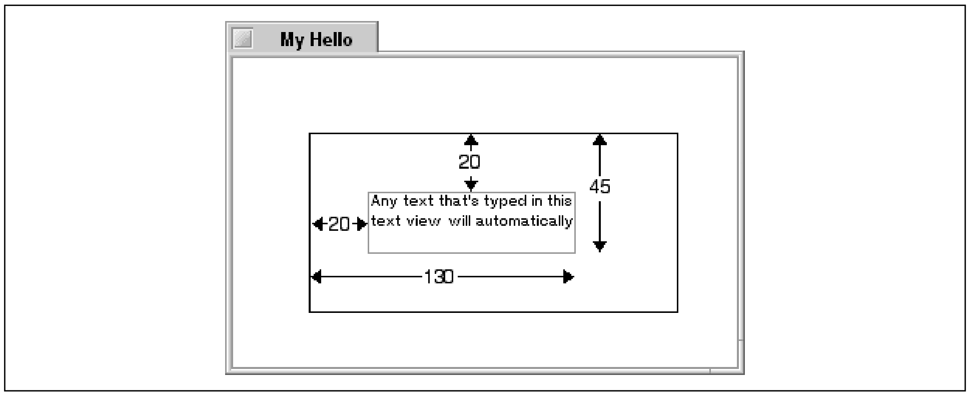


Figure 8-7. A BTextView object consists of two rectangles

this particular example. The arrows would not be in the window either—I've added them to make it clear that the coordinates of the `textBounds` rectangle are relative to the `viewFrame` rectangle. Here's the code that sets up a `BTextView` object like the one shown in Figure 8-7:

```
BTextView  *theTextView;
BRect      viewFrame(30.0, 30.0, 200.0, 110.0);
BRect      textBounds(20.0, 20.0, 130.0, 45.0);

theTextView = new BTextView(viewFrame, "TextView", textBounds,
                            B_FOLLOW_NONE, B_WILL_DRAW);
AddChild(theTextView);
```

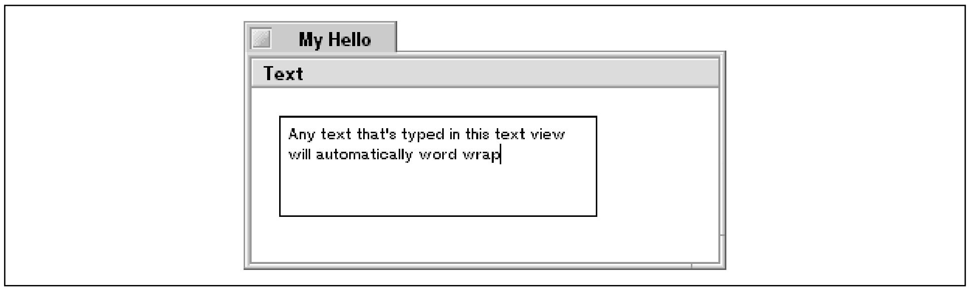
In this snippet, the `viewFrame` rectangle defines the text view object frame to be 170 pixels wide by 80 pixels high. The `textBounds` rectangle specifies that the first character typed into the text view object will have 20 pixels of white space between the object's left edge and the character and 20 pixels of white space between the object's top edge and the top of the character. The `textBounds` rectangle's right boundary, 130, means there will be 40 pixels of white space between the end of a line of text and the text object's right boundary (see Figure 8-7).

While I've discussed at length the `BTextView` constructor parameters, I'm compelled to elaborate just a bit more on the two rectangles. Figure 8-7 and the accompanying code snippet exhibit a text object whose text area rectangle provides large and non-uniform borders between it and the text object itself. But it's much more typical to define a text area rectangle that has a small, uniform border. This example exaggerated the border size simply to make the relationship between the two rectangles clear.

Another point to be aware of is that the top and bottom coordinates of the text area rectangle become unimportant as the user enters text that exceeds the size of the text area rectangle. The bottom coordinate of the text area rectangle is always ignored—the text view object will accept up to 32K of text and will automatically scroll the text as the user types, always displaying the currently typed characters. And as the text scrolls, the top coordinate of the text area rectangle becomes meaningless; the text view object will display the top line of scrolling text just a pixel or so away from the top of the text view object.

Text view example project

The TextView project displays a window like the one shown in Figure 8-8. To make the text view object's boundaries clear, the program outlines the object with a line one pixel in width. As it did for the StringView project, the Text menu holds a single item named Test. Choosing this item simply generates a system beep.



Any text that's typed in this text view
will automatically word wrap

Figure 8-8. The TextView example program's window

The text view object will be added to the window-filling `MyDrawView`, so I've added a `BTextView` data member to the `MyDrawView` class:

```
class MyDrawView : public BView {

   public:
                        MyDrawView(BRect frame, char *name);
      virtual void      AttachedToWindow();
      virtual void      Draw(BRect updateRect);

   private:
      BTextView         *fTextView;
};
```

The normally empty `MyDrawView` constructor now holds the code to create a `BTextView` object. The `viewFrame` rectangle defines the size and placement of the text view object. This rectangle is declared outside of the `MyDrawView` constructor because, as you see ahead, it sees additional use in other `MyDrawView` member functions. The `TEXT_INSET` constant is used in establishing the boundaries of the text view object's text area; that area will have a 3-pixel inset from each side of the text view object itself:

```
#define   TEXT_INSET   3.0

BRect  viewFrame(20.0, 20.0, 220.0, 80.0);
```

```
MyDrawView::MyDrawView(BRect rect, char *name)
     : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "TextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   AddChild(fTextView);

   viewFrame.InsetBy(-2.0, -2.0);
}
```

After using `viewFrame` to establish the size and placement of the text view object, this rectangle's size is expanded by 2 pixels in each direction (recall from Chapter 5, *Drawing*, that a negative number as an argument to the `BView` member function `InsetBy()` moves the affected view's frame outward in one direction). This is done in preparation for drawing a border around the text view area.

Clicking on a text view object causes a blinking insertion point caret to appear in the text area of that object. The programmer can "jump start," or force, this caret to appear in a text view object by making the object the focus view. The final setup work for a `MyDrawView` object takes place in the `AttachedToWindow()` member function, so that's an appropriate enough place to make a call to the `BView` function `MakeFocus()`:

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);

   fTextView->MakeFocus();
}
```

The `AttachedToWindow()` calls to `SetFont()` and `SetFontSize()` don't affect the font used in the text view object. They're called by the `MyDrawView` object, so they affect text drawn directly in such an object (there just doesn't happen to be any text drawn in this example's `MyDrawView` object). To change the font of a text view object, invoke the `BTextView` function `SetFontAndColor()` from the text view object. Refer to "Text Characteristics," in this chapter.

The `MyDrawView` constructor ended with the coordinates of the rectangle `viewFrame` being enlarged a couple of pixels in each direction. This was done to define a rectangle with boundaries just outside the boundaries of the text view object. When used as an argument to `StrokeRect()`, this rectangle provides a frame for the text view object. I've placed the call to `StrokeRect()` in the `MyDrawView Draw()` function so that this frame always gets appropriately updated:

```
void MyDrawView::Draw(BRect)
{
   StrokeRect(viewFrame);
}
```

You might be tempted to try to surround a text view object with a frame by simply calling `StrokeRect()` from the text view object. This won't work, because the text view object holds text, not graphics. Instead, draw the frame in the text view object's parent view as I've done above. The `fTextView` object was added to the `MyDrawView` object, so I draw the text view object's border in the `MyDrawView` object.

Text Editing

By default, the user can select and edit some or all of the text that appears in a text view object. `BTextView` member functions, along with several Be-defined message constants, provide you with a great degree of control over the level of editing you want to allow in each text view object in a window.

Default text editing

Regardless of which editing menu items you choose to include or not include in the menubar of the text object object's parent window, the following text editing shortcut keys are automatically supported:

- Command-x: Cut
- Command-c: Copy
- Command-v: Paste
- Command-a: Select All

You can verify that this notion of automatic text editing is true in practice by running the previous example program, TextView. Then type a few characters, select some or all of it, and press the Command and "X" keys. Even though the TextView project includes no text editing menu items and no text editing code, the selected text will be cut.

You can deny the user the ability to edit text in the text view object by calling the `BTextView` function `MakeEditable()`, passing a value of `false`:

```
fTextView->MakeEditable(false);
```

After disabling text editing, you can again enable editing by calling `MakeEditable()` with an argument of `true`. You can check the current editing state of a text object by calling `IsEditable()`:

```
bool  canEdit;

canEdit = fTextView->IsEditable();
```

If you disable text editing for a text object, you may also want to disable text selection. Like text editing, by default, text in a text object can be selected by clicking and dragging the mouse. If you disable text editing, the user will be able to select any number of characters in the text object. Since the user will be able to select and copy text, but won't be able to paste copied text back into the view, this could lead to some confusion. To prevent the user from selecting text by invoking the `BTextView` member function `MakeSelectable()`, pass a value of `false` as the sole argument:

```
fTextView->MakeSelectable(false);
```

You can enable text selection by again calling `MakeSelectable()`, this time with an argument of `true`. You can check the current text selection state of a text view object by calling `IsSelectable()`:

```
bool  canSelect;

canSelect = fTextView->IsSelectable();
```

Menu items and text editing

Users may not intuitively know that a text object automatically handles keyboard shortcuts for copying, cutting, pasting, and selecting all of the object's text. When it comes time to perform text editing, the user will no doubt look in the menus of a window's menubar for the basic editing menu items: the Cut, Copy, Paste, and Select All items. If you include one or more `BTextView` objects in a window of your program, you'd be wise to include these four menu items in an Edit menu.

As you've seen, a `BTextView` object automatically provides shortcut key editing—you don't need to write any code to enable the shortcut key combinations to work. The system also automatically supports menu item editing—menu item editing is easy to enable on a text view object, but you do need to write a little of your own code. While you don't do any work to give a text view object shortcut key editing, you do need to do a little work to give that same object menu editing. All you need to do is build a menu with any or all of the four basic editing items. If you include the proper messages when creating the menu items, editing will be appropriately handled without any other application-defined code being present.

You're most familiar with the *system message*: a message that has a corresponding hook function to which the system passes the message. A different type of message the system recognizes and reacts to is the *standard message*. A standard message is known to the system, and may be issued by the system, but it doesn't have a hook function. Among the many standard messages the BeOS provides are four for editing, represented by the Be-defined constants `B_CUT`, `B_COPY`, `B_PASTE`, and `B_SELECT_ALL`. This brief definition of the standard message should tide you over until Chapter 9, *Messages and Threads*, where this message type is described in greater detail. The following snippet demonstrates how an Edit menu that holds a Cut menu item could be created. Assume that this code was lifted from the constructor of a `BWindow`-derived class constructor, and that a menubar referenced by a `BMenuBar` object named `fMenuBar` already exists:

```
BMenu      *menu;
BMenuItem  *menuItem;

menu = new BMenu("Edit");
fMenuBar->AddItem(menu);

menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
```

Recall from Chapter 7 that the first parameter to the `BMenuItem` constructor, `label`, specifies the new menu item's label—the text the user sees in the menu. The second parameter, `message`, associates a message with the menu item. The third parameter, `shortcut`, assigns a shortcut key to the menu item. To let the menu item be responsible for cutting text, you must pass the Be-defined `B_CUT` standard message constant as shown above. The other required step is to set the currently selected text view object as the destination of the message.

The `BInvoker` class exists to allow objects to send a message to a `BHandler` object. The `BMenuItem` class is derived from the `BInvoker` class, so a menu item object can be invoked to send a message to a target. That's exactly what happens when the user selects a menu item. A window object is a type of `BHandler` (the `BWindow` class is derived from `BHandler`), so it can be the target of a menu item message. In fact, by default, the target of a menu item is the window that holds the menu item's menubar. Typically, a menu item message is handled by the target window object's `MessageReceived()` function, as has been demonstrated at length in Chapter 7. While having the window as the message recipient is often desirable, it isn't a requirement. The `BInvoker` function `SetTarget()` can be invoked by a `BInvoker` object (such as a `BMenuItem` object) to set the message target to any other `BHandler` object. The above snippet calls `SetTarget()` to set the active text view object to be the Cut menu item's target.

The first parameter to SetTarget() is a BHandler object, while the second is a BLooper object. Only one of these two parameters is ever used; the other is always passed a value of NULL. I'll examine both possibilities next.

If the target object is known at compile time, you can pass it as the first argument and pass NULL as the second argument. If the window involved in the previous snippet had a single text view object referenced by an fMyText data member, the call to SetTarget() could look like this:

```
menuItem->SetTarget(fMyText, NULL);
```

If the window has more than one text view object, however, setting an editing menu item message to target one specific text view object isn't desirable—selecting the menu item won't have any effect on text that is selected in a text view object other than the one referenced by fMyText. The remedy in such a case is to call SetTarget() as shown here:

```
menuItem->SetTarget(NULL, this);
```

When SetTarget() is called with a first argument of NULL, the second argument is a BLooper object. Passing a looper object doesn't set the looper object itself as the target—it sets the looper object's *preferred handler* to be the target. An object's preferred handler is dependent on the object's type, and can vary as the program runs. If the above line of code appears in a BWindow-derived class constructor, the this argument represents the BWindow-derived object being created. In the case of a BWindow-derived object, the preferred handler is whichever of the window's BHandler objects is the focus object when the window receives a message. For editing, this makes perfect sense—you'll want an editing operation such as the cutting of text to affect the text in the current text view object.

Earlier I mentioned that the default target of a menu item message is the menu's window. In the previous call to SetTarget(), the second argument is this, which is the menu's window. If that makes it seem like the call to SetTarget() is redundant, keep in mind that the object passed as the second argument to SetTarget() doesn't become the new target. Instead, that object's preferred handler becomes the target.

Other editing menu items are implemented in a manner similar to the Cut menu item. This next snippet adds Cut, Copy, Paste, and Select All menu items to an Edit menu and, at the same time, provides a fully functional Edit menu that supports editing operations in any number of text view objects:

```
BMenu      *menu;
BMenuItem  *menuItem;
```

```
menu = new BMenu("Edit");
fMenuBar->AddItem(menu);

menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Copy", new BMessage(B_COPY), 'C'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Paste", new BMessage(B_PASTE), 'V'));
menuItem->SetTarget(NULL, this);
menu->AddItem(menuItem = new BMenuItem("Select All",
                                       new BMessage(B_SELECT_ALL), 'A'));
menuItem->SetTarget(NULL, this);
```

The handling of an edit item comes from the standard message (such as `B_CUT`) that you use for the `message` parameter in the invocation of the `BMenuItem` constructor—not from the string (such as `"Cut"`) that you use for the `label` parameter. While the user will be expecting to see the familiar Cut, Copy, Paste, and Select All menu item names, you could just as well give these items the names Expunge, Mimic, Inject, and Elect Every. From a more practical standpoint, a program designed for non-English speaking people can include native text in the edit menu.

Text editing menu item example project

The TextViewEdit project is a modification of this chapter's TextView project. As shown in Figure 8-9, four menu items have been added to the already present Test item in the Text menu. For simplicity, I've added these four editing items to the existing Text menu, but your application should stick with convention and include these items in a menu titled Edit.

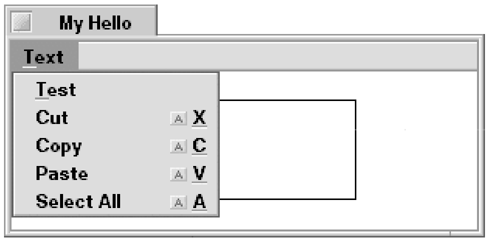


Figure 8-9. The TextViewEdit example program's window

For this TextViewEdit project, the `MyDrawView` class and the implementation of the `MyDrawView` member functions are all unchanged from the TextView project:

- `MyDrawView` class has a `BTextView` data member named `fTextView`.
- The `MyDrawView` constructor creates a `BTextView` object and assigns it to `fTextView`.
- `AttachedToWindow()` sets the focus view and sets up a rectangle to serve as a border for the text view object.
- `Draw()` draws the text view object's border.

The modifications to the project are all found in the `MyHelloWindow` constructor. Here, the four editing menu items are added to the already present Test 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;
   BMenuItem  *menuItem;
   BRect      menuBarRect;

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

   menu = new BMenu("Text");
   fMenuBar->AddItem(menu);

   menu->AddItem(new BMenuItem("Test", new BMessage(TEST_MSG)));
   menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Copy", new BMessage(B_COPY),
                                       'C'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Paste", new BMessage(B_PASTE),
                                       'V'));
   menuItem->SetTarget(NULL, this);
   menu->AddItem(menuItem = new BMenuItem("Select All",
                                        new BMessage(B_SELECT_ALL), 'A'));
   menuItem->SetTarget(NULL, this);

   Show();
}
```

The MyHelloWindow class includes a MessageReceived() member function that handles only the first menu item in the Test menu: the Text item. If MessageReceived() receives a TEST_MSG message, a call to beep() is made. The system routes the other message types (B_CUT, B_COPY, B_PASTE, and B_ SELECT_ALL) to the focus view (which in this example is the one text view object) for automatic handling by that view.

Text Characteristics

While the characteristics of the text displayed by a BStringView object can be altered by invoking BView functions such as SetFont() and SetHighColor(), the characteristics of the text displayed by a BTextView object should be altered by invoking member functions of the BTextView class. For instance, to change either the color or font, or both, of a BTextView object's text, invoke the BTextView function SetFontAndColor().

Getting BTextView text characteristics

A new BTextView object's text is displayed in black and in the system's current plain font. A BTextView object is a BView object, so you might expect that changes to the object's text would be carried out by BView member functions such as SetHighColor() and SetFont(). While this is in fact true, it's important to note that the calls to these BView functions are made indirectly. That is, the BTextView class provides its own set of graphics member functions that a BTextView object should invoke in order to affect the object's text. Each of these BTextView functions in turn invokes whatever BView functions are needed in order to carry out its specific task. The BTextView functions GetFontAndColor() and SetFontAndColor() are of primary importance in changing the characteristics of text displayed in a text view object.

GetFontAndColor() is your means to accessing a text view object's current font so that you can alter its properties. Here's the prototype:

```
    void GetFontAndColor(BFont     *font,
                         uint32    *sameProperties,
                         rgb_color *color = NULL,
                         bool      *sameColor = NULL)
```

GetFontAndColor() returns information about BTextView in its four parameters. After you call GetFontAndColor(), the font parameter holds a copy of the text view object's font. The sameProperties parameter is a mask that specifies which of a number of the text view object's font properties apply to all of the characters in the current selection. GetFontAndColor() combines a number of Be-defined constants to create the mask. For instance, if all of the characters in the selected text are the same size, a test of the returned value of `sameProperties` will reveal that it includes the value of the Be-defined constant `B_FONT_SIZE`:

```
// invoke GetFontAndColor() here

if (sameProperties && B_FONT_SIZE)
   // all characters are the same size
```

If no text is currently selected at the time of the call to `GetFontAndColor()`, the `sameProperties` mask will be set such that all of the properties test `true`. This makes sense because all of the selected characters—all none of them—do indeed share the same properties!

The third parameter, `color`, specifies the RGB color in which the text view object's text is to be displayed. The color returned in this parameter is that of the first character in the selected text (or the character following the insertion point if no text is currently selected). The `sameColor` parameter indicates whether or not all of the selected text (or all characters if no text is currently selected) is of the color returned by the `color` parameter.

If your only interest is in the font of a text view object, the `color` and `sameColor` parameters can be safely ignored—these two parameters have a default value of `NULL`. Here's an example:

```
BFont      font;
uint32     sameProperties;

fTextView->GetFontAndColor(&font, &sameProperties);
```

If, instead, your only interest is in the color of a text view object's text, pass `NULL` as the font parameter:

```
uint32     sameProperties;
rgb_color  color;
bool       sameColor;

fTextView->GetFontAndColor(NULL, &sameProperties, &color, &sameColor);
```

Alternatively, you can pass a variable for each of the four arguments, then simply ignore the returned values of the variables that aren't of interest.

A call to `GetFontAndColor()` doesn't affect a text view object's text in any way—it simply returns to your program information about the text. Once you've obtained a font and color, you'll want to make changes to one or the other, or both.

Setting BTextView text characteristics

After obtaining a copy of a `BTextView` object's font, you can make any desired changes to the font and then pass these changes back to the text view object. The same applies to the text's color. The `BTextView` function `SetFontAndColor()` takes care of both of these tasks:

```
void SetFontAndColor(const BFont  *font,
                     uint32       properties = B_FONT_ALL,
                     rgb_color    *color = NULL)
```

The `font`, `properties`, and `color` parameters are the variables filled in by a previous call to `GetFontAndColor()`. In between the calls to `GetFontAndColor()` and `SetFontAndColor()`, invoke one or more `BFont` functions to change the desired font trait. For instance, to change the size of the font used to display the text of a `BTextView` object named `theTextView`, invoke the `BFont` function `SetSize()` as shown in the following snippet. Note that because this snippet isn't intended to change the color of the text in the `theTextView` text view object, the call to `GetFontAndColor()` omits the `color` and `sameColor` parameters:

```
BFont   font;
uint32  sameProperties;

theTextView->GetFontAndColor(&font, &sameProperties);
font.SetSize(24.0);
theTextView->SetFontAndColor(&font, B_FONT_ALL);
```

In this snippet, the `BFont` object `font` gets its value from `GetFontAndColor()`, is altered by the call to `SetSize()`, and then is passed back to the `theTextView` object by a call to `SetFontAndColor()`. The process is not, however, the same for the `sameProperties` variable.

Recall that the `GetFontAndColor()` `uint32` parameter `sameProperties` returns a mask that specifies which font properties apply to all of the characters in the current selection. The `SetFontAndColor()` `uint32` parameter `properties`, on the other hand, is a mask that specifies which properties of the `BFont` parameter passed to `SetFontAndColor()` should be used in the setting of the text view object's font.

Consider the following example: your program declares a `BFont` variable named `theBigFont` and sets a variety of its properties (such as size, rotation, and shear), but you'd only like `SetFontAndColor()` to apply this font's size property to a view object's font (perhaps your program has set other characteristics of `theBigFont` because it plans to use the font elsewhere as well). To do that, pass the Be-defined constant `B_FONT_SIZE` as the second argument to `SetFontAndColor()`:

```
BFont  theBigFont(be_plain_font);

theBigFont.SetSize(48.0);
theBigFont.SetRotation(-45.0);
theBigFont.SetShear(120.0);
theTextView->SetFontAndColor(&theBigFont, B_FONT_SIZE);
```

In this snippet, you'll note that there's no call to `GetFontAndColor()`. Unlike previous snippets, this code doesn't obtain a copy of the current font used by `theTextView`, alter it, and pass the font back to `theTextView`. Instead, it creates a new font based on the system font `be_plain_font`, changes some of that font's characteristics, then passes this font to `SetFontAndColor()`. The important point to note is that passing `B_FONT_SIZE` as the second properties mask tells `SetFontAndColor()` to leave all of the characteristics of the font currently used by `theTextView` unchanged except for its size. The new size will be that of the first parameter, `theBigFont`. The font control definitions such as `B_FONT_SIZE` are located in the *View.h* header file.

To change the color of a `BTextView` object's text, assign an `rgb_color` variable the desired color and then pass that variable as the third parameter to `SetFontAndColor()`. Here, the `BTextView theTextView` is set to display the characters in the current selection in red:

```
BFont      font;
uint32     sameProperties;
rgb_color  redColor = {255, 0, 0, 255};

theTextView->GetFontAndColor(&font, &sameProperties);
theTextView->SetFontAndColor(&font, B_FONT_ALL, &redColor);
```

Allowing multiple character formats in a BTextView

By default, all of the text that is typed or pasted into a `BTextView` object shares the same characteristics. That is, it all appears in the same font, the same size, the same color, and so forth. If the object is left in this default state, user attempts to change the characteristics of some of the text will fail—all of the text in the object will take on whatever change is made to any selected text. To allow for multiple character formats in a single text view object, pass a value of `true` to the `BTextView` function `SetStylable()`.

```
theTextView->SetStylable(true);
```

To reverse the setting and prevent multiple character formats in a text view object, pass `SetStylable()` a value of `false`. Note that in doing this any styles that were previously applied to characters in the text view will now be lost. To test a text view object's ability to support multiple character formats, call the `BTextView` function `IsStylable()`:

```
bool  supportsMultipleStyles;

supportsMultipleStyles = theTextView->IsStylable();
```

Changing the background color in a BTextView

Changes to the characteristics of text in a `BTextView` object are achieved by using a number of `BFont` functions in conjunction with the `BTextView` functions `GetFontAndColor()` and `SetFontAndColor()`. Changes to the background color of the text view object itself are accomplished by calling an inherited `BView` function rather than a `BTextView` routine. The `BView` function `SetViewColor()`, which was introduced in Chapter 5, is called to change a text view object's background. The call changes the background color of the entire text view object (that is, it changes the background color of the text view object's boundary or framing rectangle—a rectangle that includes the text area rectangle). Here, the background of a `BTextView` object is changed to pink:

```
rgb_color  pinkColor = {200, 150, 200, 255};
theTextView->SetViewColor(pinkColor);
```

Aligning text in a BTextView

A characteristic of the text in a `BTextView` that isn't dependent on the font is the text's placement within the text area of the text view object. By default, the text the user enters into a `BTextView` object is left-aligned. You can change the alignment by invoking the `BTextView` member function `SetAlignment()`. This routine works just like the `BStringView` version described earlier in this chapter: it accepts one of the three Be-defined alignment constants. Pass `B_ALIGN_LEFT`, `B_ALIGN_RIGHT`, or `B_ALIGN_CENTER` and the text that's currently in the text view object's text rectangle will be appropriately aligned: each line in the text view object will start at the alignment indicated by the constant. Here, a `BTextView` object named `theText` has its alignment set to centered:

```
theText->SetAlignment(B_ALIGN_CENTER);
```

Any text subsequently entered in the affected `BTextView` object continues to follow the new alignment.

The `BTextView` member function `Alignment()` is used to obtain the current alignment of a text view object's text. The `BTextView` version of `Alignment()` works just like the `BStringView` version. In this next snippet, the alignment of the text in a `BTextView` object is compared to the `B_ALIGN_RIGHT` constant to see if the text is currently right-aligned:

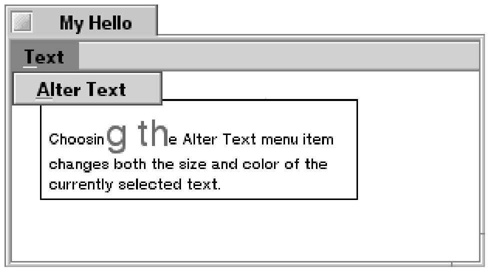
```
alignment  theAlignment;

theAlignment = theText->Alignment();

if (theAlignment == B_ALIGN_RIGHT)
   // the text in this object is right-aligned
```

Another `BTextView` function that affects text placement is `SetWordWrap()`. By default, the text in a text view object word wraps: typed or pasted text fills the width of the object's text rectangle and then continues on the following line. Text can instead be forced to remain on a single line until a newline character (a hard return established by a press of the Return key) designates that a new line be started. Passing `SetWordWrap()` a value of `false` turns word wrapping off for a particular text view object, while passing a value of `true` turns word wrapping on. To test the current word wrapping state of a text view object, call the `BTextView` function `DoesWordWrap()`.

While you can change the font characteristics (such as size and color) of individual characters within a `BTextView` object, you can't change the alignment of individual characters, words, or lines of text within a single `BTextView` object. The setting of a `BTextView` object's alignment or wrapping affects *all* characters in the view.

Text characteristics example project

The TextViewFont project demonstrates how to add support for multiple character formats in a `BTextView` object. Figure 8-10 shows a window with a Text menu and a `BTextView` object similar to the one appearing in this chapter's TextView and TextViewEdit example projects. Making a text selection and then choosing Alter Text from the Text menu increases the size of the font of the selected text. For good measure (and to demonstrate the use of color in a `BTextView`), the Alter Text menu also changes the color of the selected text from black to red. In Figure 8-10, the current selection consists of parts of the first and second words of text, and the Alter Text item has just been selected.



Choosing the Alter Text menu item changes both the size and color of the currently selected text.

Figure 8-10. The TextViewFont example program's window

A menu item named Alter Text is fine for this simple example, but is a bit ambiguous for a real-world application. If your own program includes stylable text, then it should of course include separate menu items for each possible action. For instance, if your program allows the user to change the size of selected text, it should have Increase Size and Decrease Size menu items, or a menu item for each possible font point size. Similarly, if your program allows the color of text to be changed, it should have either a separate menu item for each possible color, or a menu item that brings up a color control (see Chapter 5 for a discussion of the `BColorControl` class).

Earlier in this chapter, you saw that a standard editing message such as `B_CUT` is assigned to the focus view by using menu-creation code like that shown here:

```
menu->AddItem(menuItem = new BMenuItem("Cut", new BMessage(B_CUT), 'X'));
menuItem->SetTarget(NULL, this);
```

You can set application-defined messages to go directly to the focus view in a similar manner. Here, `FONT_TEST_MSG` is an application-defined message type:

```
menu->AddItem(menuItem = new BMenuItem("Alter Text",
                                       new BMessage(FONT_TEST_MSG)));
menuItem->SetTarget(NULL, this);
```

For a standard editing message such as `B_CUT`, the work in handling the message is done; the system knows what to do with a Be-defined standard message. For my own application-defined message, however, a bit more code is needed because `BTextView` won't know how to handle an application-defined `FONT_TEST_MSG`. To create a `BTextView` object that can respond to an application-defined message, I defined a `MyTextView` class derived from `BTextView`. From *MyTextView.h*, here's the new class and the definition of the new message type:

```
#define   FONT_TEST_MSG     'fntt'

class MyTextView : public BTextView {

   public:
      MyTextView(BRect viewFrame, char *name, BRect textBounds);
      virtual void   MessageReceived(BMessage* message);
};
```

The MyTextView class consists of just two functions: a constructor to create a new object and a version of `MessageReceived()` that will be capable of handling the application-defined message. The implementation of these new functions can be found in *MyTextView.cp*. From that file, here's the `MyTextView` constructor:

```
MyTextView::MyTextView(BRect viewFrame, char *name, BRect textBounds)
   : BTextView(viewFrame, name, textBounds, B_FOLLOW_NONE, B_WILL_DRAW)
{
}
```

The MyTextView constructor is empty—it does nothing more than invoke the BTextView constructor. For this example, I'm satisfied with how a BTextView object looks and works, so I've implemented the MyTextView constructor such that it simply passes the arguments it receives on to the BTextView constructor. The real purpose of creating the BTextView-derived class is to create a class that overrides the BTextView version of MessageReceived(). The new version of this routine handles messages of the application-defined type FONT_TEST_MSG:

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

      case FONT_TEST_MSG:
         rgb_color  redColor = {255, 0, 0, 255};
         BFont      font;
         uint32     sameProperties;

         GetFontAndColor(&font, &sameProperties);
         font.SetSize(24.0);
         SetFontAndColor(&font, B_FONT_ALL, &redColor);
         break;

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

When an object of type MyTextView receives a FONT_TEST_MSG message, GetFontAndColor() is called to obtain a copy of the object's font. The size of the font is set to 24 points and SetFontAndColor() is called to pass the font back to the MyTextView object. When calling SetFontAndColor(), the rgb_color variable redColor is included in the parameter list in order to set the MyTextView object to use a shade of red in displaying text.

As is always the case, it's important that MessageReceived() include the default condition that invokes the inherited version of MessageReceived(). The MyTextView version of MessageReceived() handles only the application-defined FONT_TEST_MSG, yet a MyTextView object that is the focus view will also be receiving B_CUT, B_COPY, B_PASTE, and B_SELECT_ALL messages. When the MyTextView version of MessageReceived() encounters a message of one of these Be-defined types, it passes it on to the BTextView version of MessageReceived() for handling.

In this chapter's previous two examples, TextView and TextViewEdit, the MyDrawView class included a data member of type BTextView. Here I also include a data member in the `MyDrawView` class, but I change it to be of type `MyTextView`:

```
class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);

   private:
      MyTextView      *fTextView;
};
```

Like the previous example versions of the `MyDrawView` constructor, the TextView-Font project's version of the `MyDrawView` constructor creates a text view object. Here, however, the object is of the new `MyTextView` class type. Before exiting, the constructor calls the `BTextView` function `SetStylable()` to give the new `MyTextView` object support for multiple character formats. That ensures that changes made to the `MyTextView` object's text apply to only the current selection—not to all of the text in the object.

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new MyTextView(viewFrame, "TextView", textBounds);

   AddChild(fTextView);

   fTextView->SetStylable(true);
}
```

Scrolling

Text or graphics your application uses may not always fit within the confines of a view. Enter the scrollbar. The APIs for other operating systems that employ a graphical user interface include routines to work with scrollbars, but few implement this interface element as elegantly as the BeOS API does. The `BScrollBar` class makes it easy to add one or two scrollbars to any view. Better still, the `BScrollView` class creates a bordered object with one or two `BScrollBar` objects already properly sized to fit any view you specify. Best of all, scrollbar operation and updating is all automatic. Scrollbar objects are unique in that they don't receive drawing or mouse down messages—the Application Server intercepts these messages and responds accordingly.

As part of the automatic handling of a scrollbar, the Application Server is responsible for changing the highlighting of a scrollbar. In Figure 8-11, you see the same scrollbar with two different looks. When the contents of the view a scrollbar is attached to exceed the size of the view, the scrollbar's knob appears and the scrollbar becomes enabled. As the content of the view increases, the scrollbar knob automatically decreases in size to reflect the lessening amount of the total text being displayed within the view.

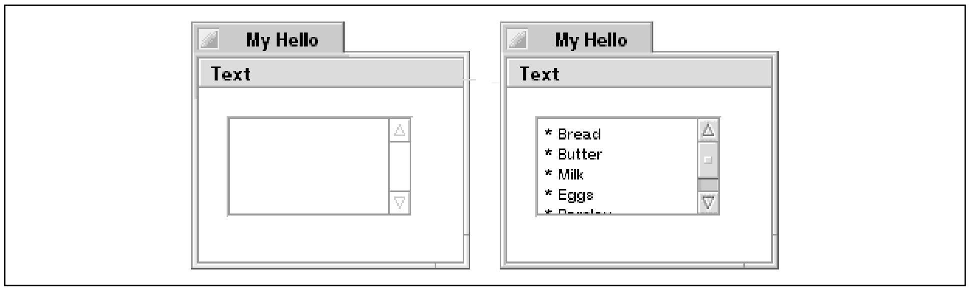


Figure 8-11. A scrollbar's look changes as the scrolled view's content increases

Keep in mind that from the ScrollBar preferences application the user can control the look and behavior of the scrollbars present in *all* applications that run on his or her machine. For instance, the ScrollBar preferences control whether programs display a scrollbar with a pair of scroll arrows on each end, or just one. With that in mind, you'll note that the look of the scrollbars in this chapter's figures differs (compare Figure 8-11 with Figure 8-13).

Scrollbars

To make use of a scrollbar, create a `BScrollBar` object, designate a different view to be the scrollbar's *target*—the thing to be scrolled—and add the `BScrollBar` object to the same parent view as the target view. Or, more likely, create a `BScrollView` object and let it do this work for you. Because the scrolling view is so handy, the "Scrolling" section's emphasis is on the `BScrollView` class. A `BScrollView` object includes one or two `BScrollBar` objects, though, so a study of the `BScrollBar` class won't prove to be time wasted.

The BScrollBar class

A scrollbar object is an instance of the `BScrollBar` class. Pass six arguments to its constructor, the prototype of which is shown here:

```
BScrollBar(BRect       frame,
           const char  *name,
           BView       *target,
           float        min,
           float        max,
           orientation  posture)
```

The first parameter, `frame`, is a rectangle that defines the boundaries of the scrollbar. The coordinates of the rectangle are relative to the scrollbar's parent view, not to the thing that is to be scrolled. User interface guidelines state that scrollbars should be of a uniform thickness. That is, all horizontal scrollbars should be of the same height, and all vertical scrollbars should be of the same width. Use the Be-defined constants `B_H_SCROLL_BAR_HEIGHT` and `B_V_SCROLL_BAR_WIDTH` to ensure that your program's scrollbars have the same look as those used in other Be applications. For instance, to set up the rectangle that defines a horizontal scrollbar to run along the bottom of a text object, you might use this code:

```
BRect  horizScrollFrame(20.0, 50.0, 220.0, 50.0 + B_H_SCROLL_BAR_HEIGHT);
```

The second parameter serves the same purpose as the name parameter in other `BView`-derived classes: it allows the view to be accessed by name using the `BView` function `FindView()`.

A scrollbar acts on a *target*—a view associated with the scrollbar. The target is what gets scrolled, and is typically an object that holds text or graphics. To bind a scrollbar to the object to be scrolled, follow these steps:

1. Create the target object.
2. Add the target object to a view (its parent view).
3. Create a scrollbar object, using the target object as the `target` parameter in the `BScrollBar` constructor.
4. Add the scrollbar object to the view that serves as the target object's parent.

For a target that displays graphics, the `min` and `max` parameters determine how much of the target view is displayable. The values you use for these two parameters will be dependent on the total size of the target. If the target view is a `BTextView` object, the values assigned to `min` and `max` are inconsequential—the scrollbar is always aware of how much text is currently in the target view and adjusts itself accordingly. The exception is if both `min` and `max` are set to 0. In such a case, the scrollbar is disabled and no knob is drawn, regardless of the target view's contents. Refer to the "Scrollbar range" section just ahead for more details on choosing values for these two parameters when the target consists of graphics.

The last parameter, posture, designates the orientation of the scrollbar. To create a horizontal scrollbar, pass the Be-defined constant B_HORIZONTAL. For a vertical scrollbar, pass B_VERTICAL.

The following snippet provides an example of a call to the BScrollBar constructor. Here a horizontal scrollbar is being created for use with a previously created BTextView object named theTextView. Because the target is a text object, the values of min and max are arbitrarily selected:

```
BScrollBar  *horizScrollBar;
BRect       horizScrollFrame(20.0, 50.0, 220.0,
                             50.0 + B_H_SCROLL_BAR_HEIGHT);
float       min = 1.0;
float       max = 1.0;

horizScrollBar = new BScrollBar(scrollFrame, "ScrollBar", theTextView,
                                min, max, B_HORIZONTAL);
```

Scrollbar example project

This chapter's TextView example project demonstrated how a window can support editable text through the use of a BTextView object. Because the topic of scrollbars hadn't been presented when TextView was developed, the program's text view didn't include them. Now you're ready to add a vertical scrollbar, a horizontal scrollbar, or both to a text view. As shown in Figure 8-12, for the TextViewScrollBar project I've elected to add just a vertical scrollbar to the window's BTextView object.

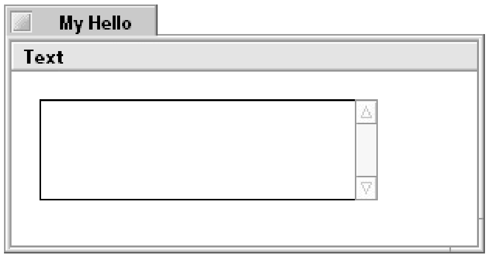


Figure 8-12. The TextViewScrollBar example program's window

The TextView example project shows how to set up viewFrame and textBounds, the rectangles that define the boundary of the window's BTextView object and the area that displays text in that object. Here I'll discuss only the code that's been added to the TextView project to turn it into the TextViewScrollBar project. All those additions appear in the `MyDrawView` constructor. The new code involves the creation of a vertical scrollbar, and appears following the call to `InsetBy()`:

```
BRect  viewFrame(20.0, 20.0, 220.0, 80.0);


MyDrawView::MyDrawView(BRect rect, char *name)
   : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   AddChild(fTextView);

   viewFrame.InsetBy(-2.0, -2.0);

   BScrollBar  *verticalBar;
   BRect       scrollFrame;
   float       min = 1.0;
   float       max = 1.0;

   scrollFrame = viewFrame;
   scrollFrame.left = scrollFrame.right;
   scrollFrame.right = scrollFrame.left + B_V_SCROLL_BAR_WIDTH;

   verticalBar = new BScrollBar(scrollFrame, "VerticalBar", fTextView,
                                min, max, B_VERTICAL);
   AddChild(verticalBar);
   verticalBar->SetResizingMode(B_FOLLOW_NONE);
}
```

The scrollbar will be added to the target's parent view (`MyDrawView`), so the coordinates of the scrollbar's rectangle are relative to the parent view, not to the target view. The `scrollFrame` rectangle coordinates are first set to those of the target view. That gives the scrollbar the same top and bottom coordinates as the target, as desired. Then the scrollbar's left and right side coordinates are adjusted so that the scrollbar lies flush on the right side of the target view. The `scrollFrame` rectangle is then used as the first parameter to the `BScrollBar` constructor. The previously created `BTextView` object `fTextView` is specified as the scrollbar's target. The scrollbar is then added to `MyDrawView`, just as the target was previously added.

Graphical user interface conventions dictate that a vertical scrollbar be located flush against the right side of the area that is scrollable, and that a horizontal scrollbar be located flush against the bottom of the area that's scrollable. But nothing in the BScrollBar class *forces* you to follow that convention. A scrollbar is associated with a view to scroll by naming the view as the target in the BScrollBar constructor. Depending on the coordinates you choose for the frame rectangle parameter of the BScrollBar constructor, the scrollbar can be placed anywhere in the parent view and it will still scroll the contents of the target view.

Notice that in this example the resizing mode of the scrollbar is adjusted. The BScrollBar constructor doesn't include a sizing mode parameter. Instead, the specification of the scrolling bar's orientation (the last parameter to the BScrollBar constructor) defines a default behavior for the scrollbar. A vertically oriented scrollbar (like the one created here) resizes itself vertically. As the parent view changes in size vertically, the vertical scrollbar will grow or shrink vertically. As the parent view changes in size horizontally, the vertical scrollbar will follow the parent view.

In many cases these default characteristics are appropriate. In this project, they aren't. I've given the BTextView object a resizing mode of B_FOLLOW_NONE, indicating that the text view object will remain a fixed size as the parent view changes size. In such a case, I want the target view's scrollbar also to remain fixed in size and location. A call to the BView function SetResizingMode() takes care of that task.

Scrollbar range

If a scrollbar's target is a view that holds a graphical entity, such as a BView object that includes a BPicture object, the BScrollBar constructor min and max parameters take on significance. Together, min and max define a range that determines how much of the target view is displayable. Consider a view that is 250 pixels in width and 200 pixels in height, and is to be displayed in a view that is 100 pixels by 100 pixels in size. If this 100-by-100 pixel view has two scrollbars, and it's desired that the user be able to scroll the entire view, the range of the horizontal scrollbar should be 150 and the range of the vertical scrollbar should be 100. Figure 8-13 illustrates this.

In Figure 8-13, 100 pixels of the 250-pixel width of the view will always be displayed, so the horizontal range needs to be only 150 in order to allow the horizontal scrollbar to bring the remaining horizontal portions of the view through the display area. Similarly, 100 of the 200 vertical pixels will always be displayed in

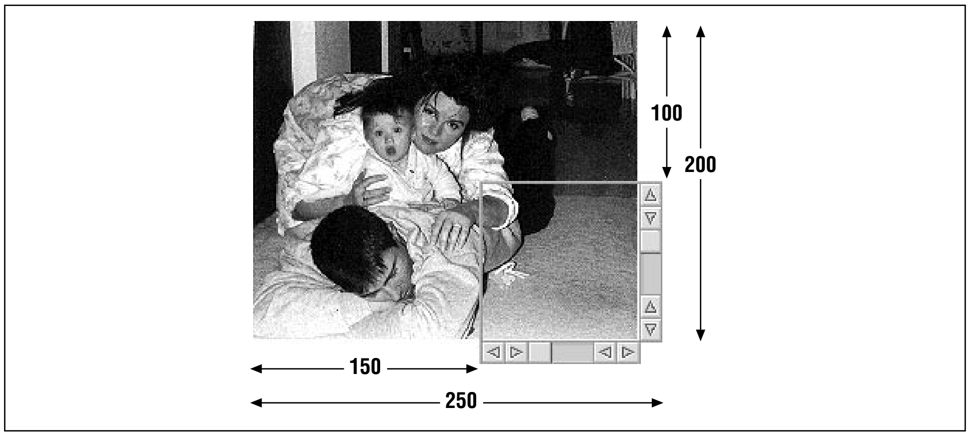


Figure 8-13. An example of determining the range of a pair of scrollbars

the view, so the vertical range needs to be 100 in order to pass the remaining 100 pixels through the display area. Assuming a graphics-holding view named `theGraphicView`, derived from a `BView` object, exists (as shown in Figure 8-13), the two scrollbars could be set up as shown in the following snippet. To see the relationship of the coordinates of the scrollbars to the view, refer to Figure 8-14.

```
BScrollBar  *horizScrollBar;
BRect       horizScrollFrame(170.0, 300.0, 270.0,
                             300.0 + B_H_SCROLL_BAR_HEIGHT);
float       horizMin =   0.0;
float       horizMax = 150.0;
BScrollBar  *vertScrollBar;
BRect       vertScrollFrame(270.0, 200.0, 270.0 + B_V_SCROLL_BAR_WIDTH,
                            300.0);
float       vertMin  =   0.0;
float       vertMax  = 100.0;

horizScrollBar = new BScrollBar(horizScrollFrame, "HScrollBar",
                                theGraphicView,
                                horizMin, horizMax, B_HORIZONTAL);
vertScrollBar  = new BScrollBar(vertScrollFrame, "VScrollBar",
                                theGraphicView,
                                vertMin, vertMax, B_VERTICAL);
```

If the target view changes size during program execution (for instance, your program may allow the user to replace the currently displayed contents of the view with different graphic), the range of any associated scrollbar should change, too. The `BScrollBar` function `SetRange()` exists for this purpose:

```
void SetRange(float  min,
              float  max)
```

The example just discussed has a horizontal scrollbar with a range of 0 to 150 pixels. If for some reason I wanted to allow the user to be able to scroll beyond the

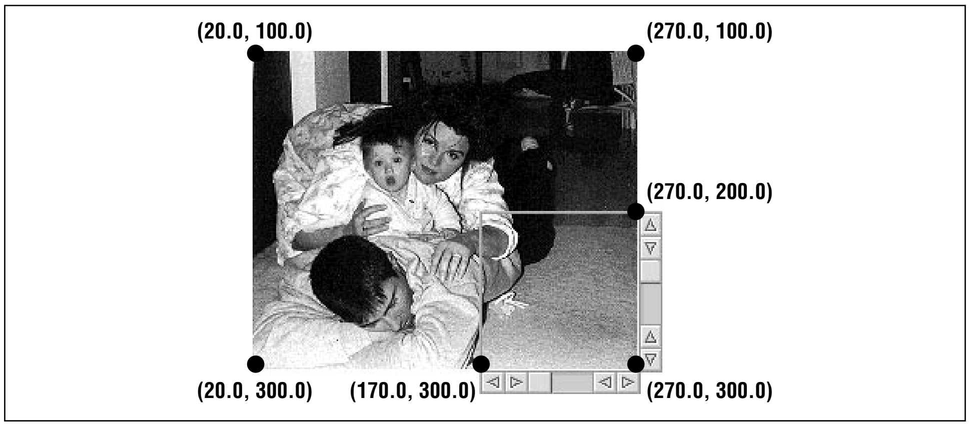


Figure 8-14. The coordinates of a pair of scrollbars and the picture to be scrolled

right edge of the view, I could increase the scrollbar's maximum value. Here I change the scrollbar's range to allow the user to view 50 pixels of white space past the view's right edge:

```
horizScrollBar->SetRange(0.0, 200.0);
```

The companion function to `SetRange()` is `GetRange()`. As expected, this function returns the current minimum and maximum scrolling values of a scrollbar:

```
void GetRange(float  *min,
              float  *max)
```

If this next snippet is executed after the preceding call to `SetRange()`, `min` should have a value of 0.0 and `max` should have a value of 200.0:

```
float  min;
float  max;

horizScrollBar->GetRange(&min, &max);
```

The ScrollViewPicture example near the end of this chapter provides an example of setting the scrollbar range for a `BView` object that's used as the target for a `BScrollView` object—a view that has built-in scrollbars.

Scrolling View

It's a relatively easy assignment to add scrollbars to a view, as just demonstrated in the TextViewScrollBar project. However, the BeOS API makes it easier still. The `BScrollView` class creates a scroll view object that serves as a container for another view. This contained view can hold either text (as a `BTextView` does) or graphics (as a `BPicture` does). Regardless of the content of its contained view, the BScrollView object is responsible for adding scrollbars that allow for scrolling through the entire content and for making itself the parent of the contained view.

The BScrollView class

The seven parameters of the BScrollView constructor make it possible to create a scrolling view object that has one, two, or even no scrollbars:

```
BScrollView(const char    *name,
            BView         *target,
            uint32        resizingMode = B_FOLLOW_LEFT | B_FOLLOW_TOP,
            uint32        flags = 0,
            bool          horizontal = false,
            bool          vertical = false,
            border_style  border = B_FANCY_BORDER)
```

The name, resizingMode, and flags parameters serve the purposes expected of a BView-derived class. The name parameter allows the scroll view object to be accessed by its name. The resizingMode specifies how the object is to be resized as the parent view changes size: the default value of B_FOLLOW_LEFT | B_FOLLOW_TOP indicates that the distance from the scroll view's left side and its parent's left side will be fixed, as will the distance from the scroll view's top and its parent's top. The flags parameter specifies the notification the scroll view object is to receive. The default value of 0 means that the object isn't to receive any notification.

The target parameter specifies the previously created view object to be surrounded by the scroll view object. The contents of the target view are what is to be scrolled. There's no need to specify any size for the new scroll view object. It will automatically be given a framing rectangle that accommodates the target view, any scrollbars that may be a part of the scroll view object, and a border (if present).

A scroll view object can have a horizontal scrollbar, a vertical scrollbar, both, or neither. The horizontal and vertical parameters specify which scrollbar or bars should be a part of the scroll view object. By default, the object includes no scrollbars (meaning the object serves as nothing more than a way to draw a border around a different view, as discussed in the border parameter description next). To include a scrollbar, simply set the appropriate horizontal or vertical parameter to true.

The border parameter specifies the type of border to surround the scroll view object. By default, a scroll view object has a fancy border; the appearance of a groove surrounds the object. To specify a plain line border, pass the Be-defined constant B_PLAIN_BORDER as the last argument to the BScrollView constructor. To omit the border completely, pass B_NO_BORDER instead.

In this next snippet, a scroll view object is created with a plain border and a vertical scrollbar. Here it's assumed that theTextView is a BTextView object that isn't resizable. Because the target is fixed in the window in which it resides, the scroll view too can be fixed. As evidenced by the value of the resizingMode parameter (B_FOLLOW_NONE), theScrollView won't be resizeable:

```
BScrollView  *theScrollView;

theScrollView = new BScrollView("MyScrollView", theTextView, B_FOLLOW_NONE,
                                0, false, true, B_PLAIN_BORDER);
```

Scroll view example project

This chapter's TextViewScrollBar project modified the TextView example project to demonstrate how a BScrollBar object can be used to scroll the contents of a BTextView object. This latest project, ScrollViewText, achieves the same effect. Here, however, a BScrollView object is used to create the BTextView object's scrollbar. The resulting window looks similar to the one generated by the TextViewScrollBar project (refer back to Figure 8-12). Thanks to the BScrollView, however, here the border around the text view object has shading, as shown in Figure 8-15.

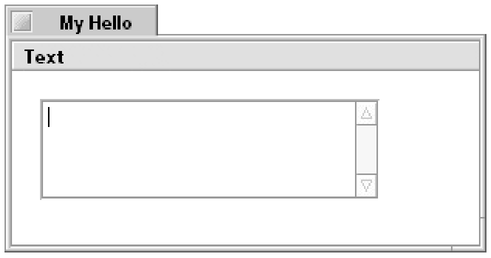


Figure 8-15. The ScrollViewText example program's window

While the results of the TextViewScrollBar and ScrollView Text projects are similar, the effort expended to obtain the results differs. Using a BScrollView object to supply the scrollbar (as done here) rather than using a BScrollBar object means there's no need to supply the scrollbar's coordinates. The BScrollView object takes care of the scrollbar's placement based on the location of the designated target. Additionally, there's no need to draw a border around the text view; the BScrollBar object takes care of that task too.

The MyDrawView class declaration is the same as it was for the original TextView project, with the addition of a BScrollView object:

```
class MyDrawView : public BView {

   public:
                      MyDrawView(BRect frame, char *name);
      virtual void    AttachedToWindow();
      virtual void    Draw(BRect updateRect);

   private:
      BTextView       *fTextView;
      BScrollView     *fScrollView;
};
```

The `MyDrawView` constructor holds the scroll view code:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame(20.0, 20.0, 220.0, 80.0);
   BRect  textBounds;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_NONE, B_WILL_DRAW);
   fScrollView = new BScrollView("MyScrollView", fTextView, B_FOLLOW_NONE,
                                 0, false, true);
   AddChild(fScrollView);
}
```

The `MyDrawView` constructor begins by setting up and creating a `BTextView` object. This code is the same as the code that appears in the TextView version of the `MyDrawView` constructor, with one exception. Here, a call to `AddChild()` doesn't immediately follow the creation of the text view object. The newly created `BTextView` object isn't added to the drawing view because it is instead added to the `BScrollView` object when it is passed to the scroll view's constructor. The `BScrollView` object is then added to the drawing view. When the program creates a window, that window's view hierarchy will look like the one pictured in Figure 8-16. In this figure, you see that the `BScrollView` object `fScrollView` is the parent to two views: the target view `fMyTextView` and a `BScrollBar` object created by the `BScrollView` constructor. Later, in the ScrollViewPicture project, you'll see how your code can easily access this implicitly created scrollbar.

The implementation of this project's `AttachedToWindow()` function is identical to the TextView projects: call `SetFont()` and `SetFontSize()` to specify font information for the drawing view, then call `fTextView->MakeFocus()` to start the cursor blinking in the `BTextView` object. The implementation of `Draw()` is simple—here it's an empty function. In the TextView project, `StrokeRect()` was invoked

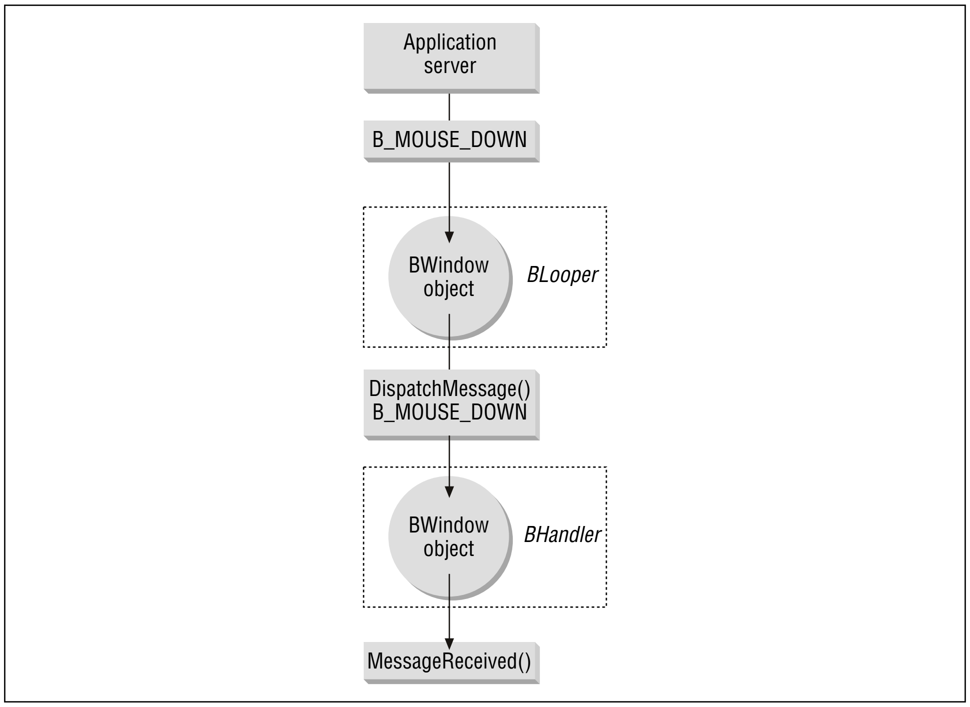


Figure 8-16. View hierarchy of the window of the ScrollViewText program

to draw a border around the `BTextView` object. Here, I rely on the `BScrollView` object's ability to automatically draw its own border.

```
void MyDrawView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);

   fTextView->MakeFocus();
}


void MyDrawView::Draw(BRect)
{
}
```

Scrolling window example project

If your program offers the user text editing capabilities, it may make sense to provide a window that exists for just that purpose. Typically, a simple text editor displays a resizable window bordered by a vertical scrollbar and possibly a horizontal scrollbar. Figure 8-17 shows such a window—the window displayed by the ScrollViewWindow project.

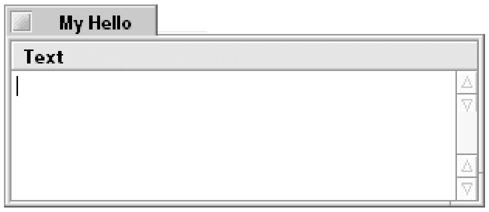


Figure 8-17. The ScrollViewWindow example program's window

The ScrollViewWindow project includes only slight modifications to the Text-ViewScrollBar project. All the changes are found in the `MyDrawView` constructor:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame;
   BRect  textBounds;

   viewFrame = Bounds();
   viewFrame.right -= B_V_SCROLL_BAR_WIDTH;

   textBounds.left = TEXT_INSET;
   textBounds.right = viewFrame.right - viewFrame.left - TEXT_INSET;
   textBounds.top = TEXT_INSET;
   textBounds.bottom = viewFrame.bottom - viewFrame.top - TEXT_INSET;

   fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                             B_FOLLOW_ALL, B_WILL_DRAW);

   fScrollView = new BScrollView("MyScrollView", fTextView,
                                  B_FOLLOW_ALL, 0, false, true);
   AddChild(fScrollView);
}
```

A call to the `BView` function `Bounds()` sets the coordinates of what is to be the `BScrollView` target, the `BTextView` view object, to the coordinates of the drawing view. The drawing view itself is the same size as the content area of the window it resides in, so this brings me close to my goal of making the entire content area of the window capable of holding user-entered text. The exception is that room needs to be allowed for the vertical scrollbar. Subtracting the width of this scrollbar results in a `viewFrame` rectangle with the desired size.

The other changes to the `MyDrawView` constructor involve the `resizingMode` parameter to both the `BTextView` constructor and the `BScrollView` constructor. In the previous example, the scroll view was fixed in size, so neither the scroll view nor its target needed to be concerned with resizing. Here I want the text area of the window to always occupy the entire content area of the window, less the window's scrollbar area. A `resizingMode` of `B_FOLLOW_ALL` (rather than

B_FOLLOW_NONE) tells both the BScrollView object and its BTextView target that they should automatically resize themselves as the user changes the parent window's size.

Accessing a BScrollView scrollbar

The BScrollView constructor lets you specify whether a BScrollView object should include a horizontal scrollbar, a vertical scrollbar, or both. The constructor is responsible for creating the appropriate number of BScrollBar objects and placing them within the BScrollView object. The BScrollView constructor gives each of its BScrollBar objects a default range of 0.0 to 1000.0. If the BScrollView object's target is a BTextView, these default minimum and maximum values always suffice—the BTextView object makes sure that the ranges of the scrollbars that target it are adjusted accordingly. If the BScrollView object's target is instead a view that holds graphics, you'll need to adjust the range of each scrollbar so that the user is guaranteed visual access to the entire target view.

This chapter's "Scrollbar range" section describes how to determine the range for a scrollbar to scroll graphics, as well as how to use the BScrollBar function SetRange() to set a scrollbar's minimum and maximum values. After determining the range a scroll view object's scrollbar should have, invoke the BScrollView function ScrollBar() to gain access to the scrollbar in question. Pass ScrollBar() the type of scrollbar (B_HORIZONTAL or B_VERTICAL) to access, and the routine returns the BScrollBar object. Then invoke that object's SetRange() function to reset its range. Here, access to the vertical scrollbar of a BScrollView object named theScrollView is gained, and the scrollbar's range is then set to a minimum of 0.0 and a maximum of 340.0:

```
BScrollBar  *scrollBar;

scrollBar = theScrollView->ScrollBar(B_VERTICAL);
scrollBar->SetRange(0.0, 340.0);
```

Scrollbar access example project

While topics such as the BStringView class and the BTextView class have made the display of text the focus of this chapter, I'll close with an example that demonstrates how to scroll a picture. The BScrollView class (covered later in this section) places no restrictions on what type of view is to be the designated target, so the differences between scrolling text and scrolling graphics are minimal. As shown in Figure 8-18, the ScrollViewPicture project demonstrates how to use a BPicture object as the target of a BScrollView object.

This example would work just fine with a very simple picture, such as one created by drawing a single large rectangle. I've opted to instead create a slightly more complex picture, and at the same time set up the project such that it's an

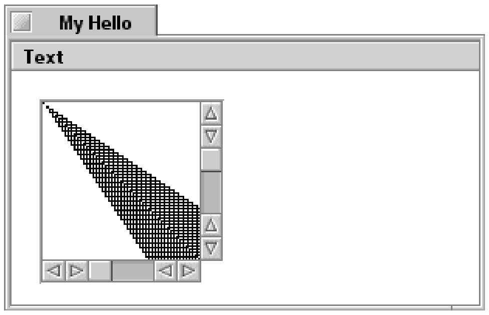


Figure 8-18. The ScrollViewPicture example program's window

easy task to make the picture far more complex. I've done that by creating a `BView`-derived class named `MyPictureView`. The declaration of this new class appears in the *MyPictureView.h* header file, while the implementations of the class member functions can be found in *MyPictureView.cpp*. Here's the `MyPictureView` declaration:

```
class MyPictureView : public BView {

   public:
                     MyPictureView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);

   private:
      BPicture       *fPicture;
};
```

As you'll see in this example, a `MyPictureView` object will become the target of a `BScrollView` object. That means the `MyPictureView` class could include any number of graphics, and the scroll view object will view them all as a single entity. So while I've included a single `BPicture` data member in the `MyPictureView`, your `BView`-derived class could hold two, three, or three hundred pictures, along with other graphic data members such as polygons or bitmaps (see Chapter 5 for shape-drawing information and Chapter 10, *Files*, for bitmap details).

The `AttachedToWindow()` routine sets up the picture. Recall that such code can't appear in the constructor of the picture's parent view. That's because the `BPicture` definition relies on the current state of the picture's parent view, and the parent view's state isn't completely set up until the execution of `AttachedToWindow()`.

```
void MyPictureView::AttachedToWindow()
{
   SetFont(be_bold_font);
   SetFontSize(12);
   BeginPicture(new BPicture);
      BRect  aRect;
      int32  i;

      for (i=0; i<100; i++) {
         aRect.Set(i*2, i*2, i*3, i*3);
         StrokeRect(aRect);
      }
   fPicture = EndPicture();
}
```

I mentioned earlier that your own BView class could contain more than one BPicture data member. Looking at the above version of AttachedToWindow(), you might think that's unnecessary because you can define a single BPicture object to include all the graphics a view needs. That may be the case—or it may not be. Your program might construct a BView-derived class by piecing together multiple BPicture objects. For instance, your application may give the user the opportunity to define a single large graphic by selecting numerous small pictures.

The BScrollView object will exist in a MyDrawView object, and will have a BScrollView object as its target. I've included MyPictureView and BScrollView data members in the MyDrawView class to keep track of these objects.

```
class MyDrawView : public BView {

   public:
                     MyDrawView(BRect frame, char *name);
      virtual void   AttachedToWindow();
      virtual void   Draw(BRect updateRect);

   private:
      MyPictureView  *fPictureView;
      BScrollView    *fScrollView;
};
```

The MyDrawView constructor begins by creating what will be the target view: fPictureView. Next, it creates a BScrollView object with fPictureView as the target. After adding the scroll view to the drawing view, the default ranges of the scroll view's two scrollbars are altered. Looking back at the MyPictureView version of AttachedToWindow(), you'll note that the BPicture object will be 300 pixels wide and 300 pixels high. Yet the width and height of the viewFrame rectangle that is to display this picture are each only 100 pixels. The difference in the actual size of the picture and the size of the rectangle the picture's displayed in is used in resetting the scrollbar ranges:

```
MyDrawView::MyDrawView(BRect rect, char *name)
    : BView(rect, name, B_FOLLOW_ALL, B_WILL_DRAW)
{
   BRect  viewFrame(20.0, 20.0, 120.0, 120.0);

   fPictureView = new MyPictureView(viewFrame, "MyPictureView");

   fScrollView = new BScrollView("MyScrollView", fPictureView,
                                 B_FOLLOW_NONE, 0, true, true);
   AddChild(fScrollView);

   BScrollBar  *scrollBar;

   scrollBar = fScrollView->ScrollBar(B_VERTICAL);
   scrollBar->SetRange(0.0, 200.0);
   scrollBar = fScrollView->ScrollBar(B_HORIZONTAL);
   scrollBar->SetRange(0.0, 200.0);
}
```

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.

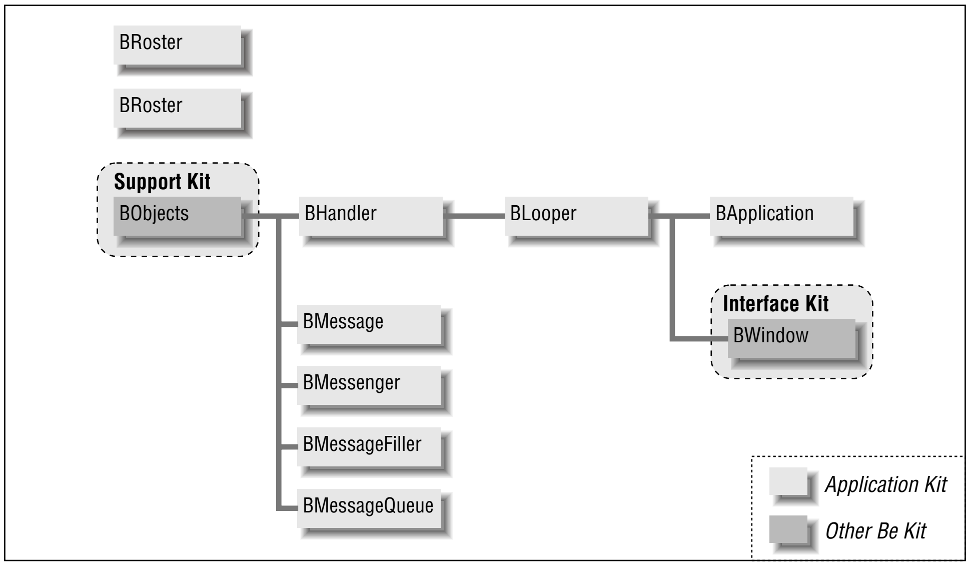


Figure 9-1. The inheritance hierarchy for the Application Kit

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.

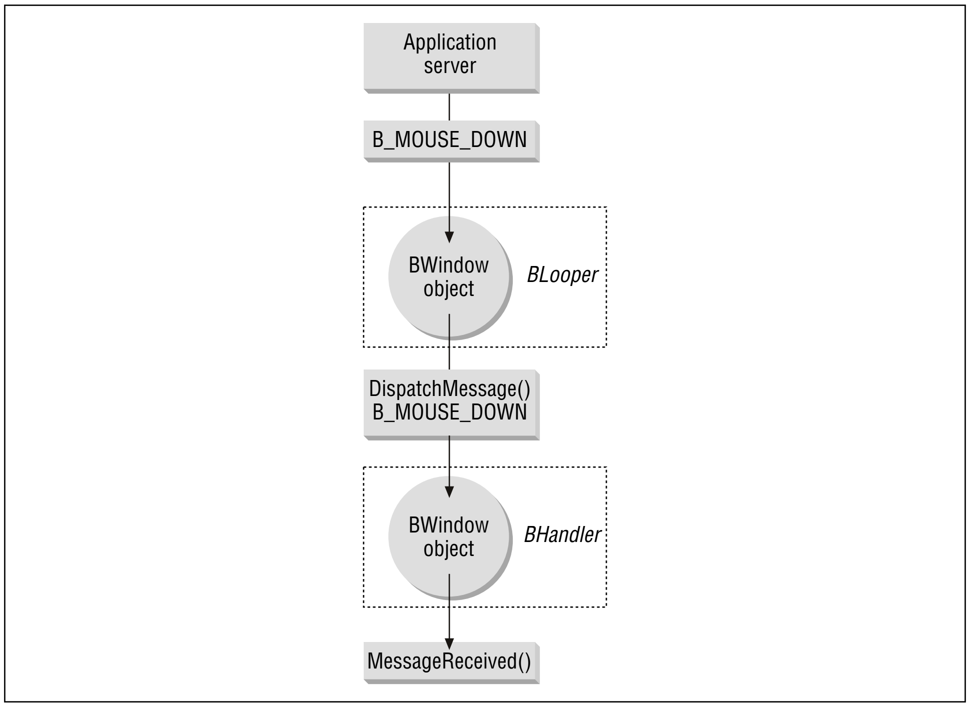


Figure 9-2. A message moves from the Application Server to a view

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()`.

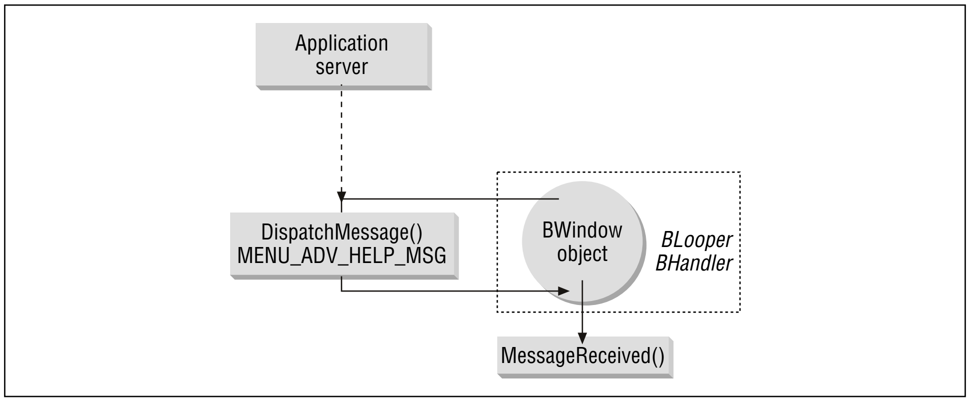


Figure 9-3. A message moves from a window back to that window

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.

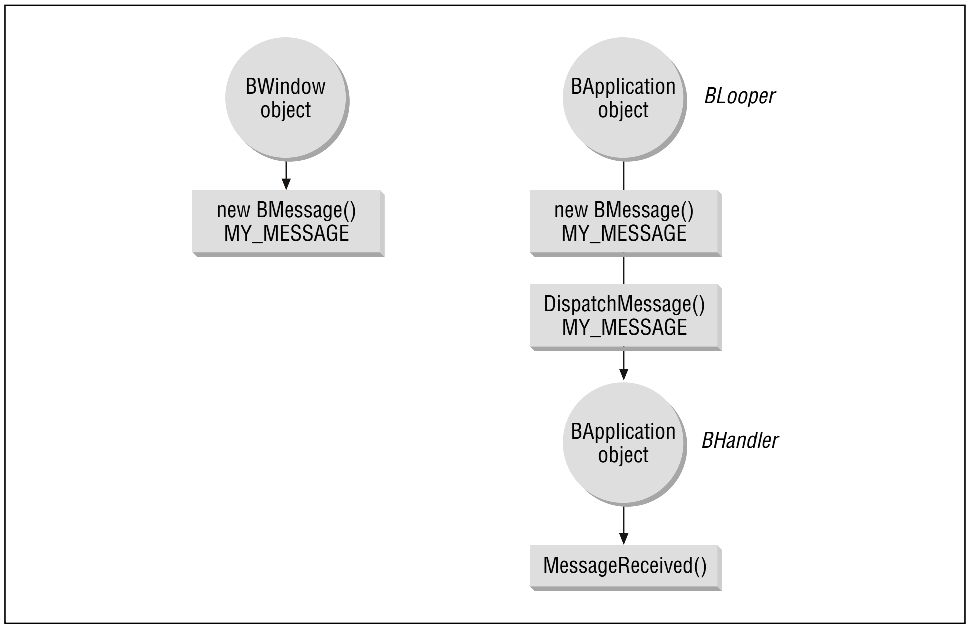


Figure 9-4. A message moves from a window to the application

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.

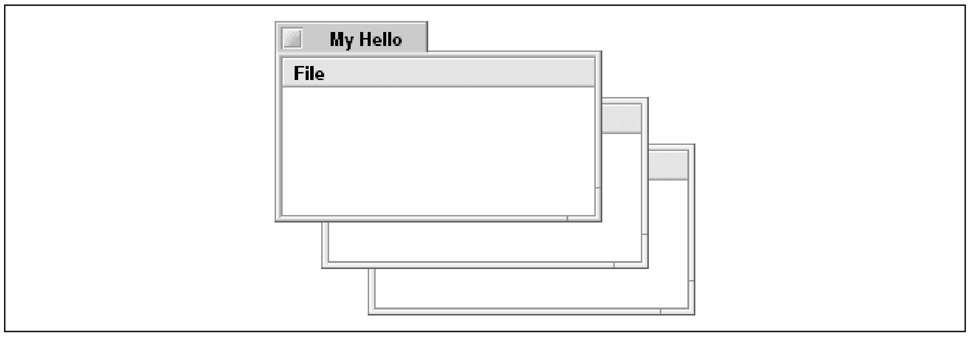


Figure 9-5. The staggered windows of the WindowMessage1 program

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;
    }
}
```

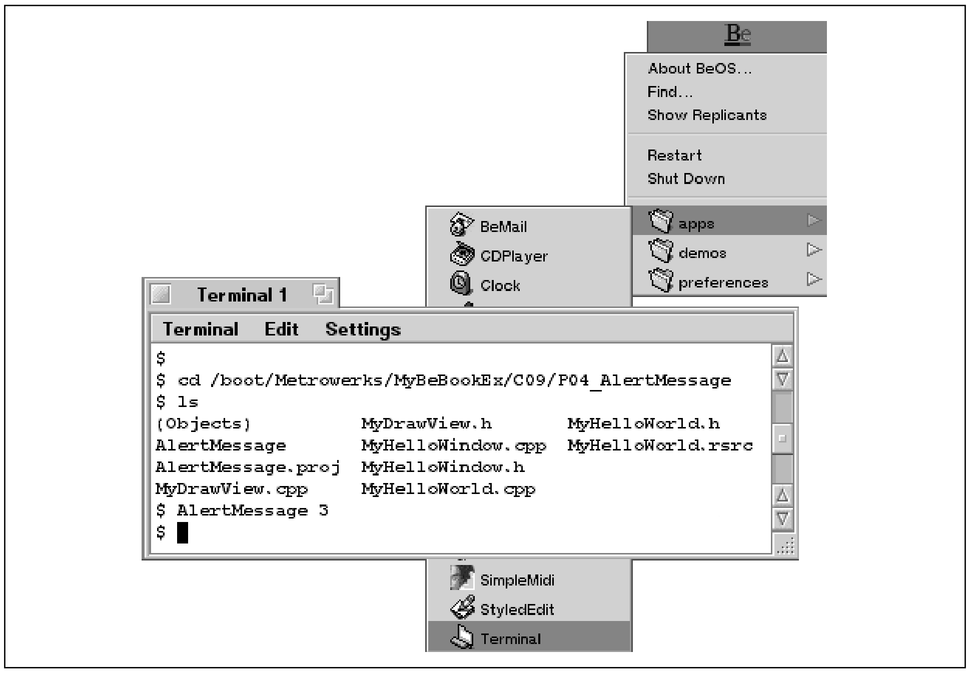
If you enable the debugger and run the program, you might be able to see multithreading in action. If you set a breakpoint in the `MyHelloApplication` version of `MessageReceived()`, you'll note that, as expected, the function gets called once for each already open window. You may be surprised to see 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.

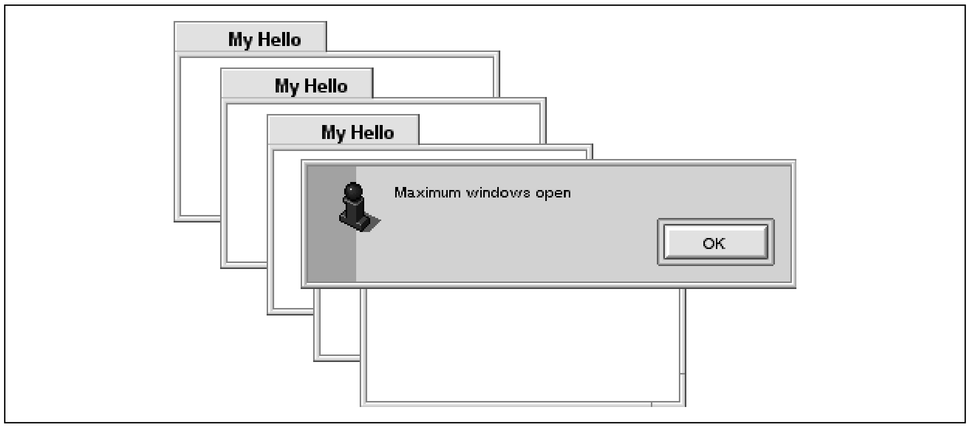
The AlertMessage program allows at most five windows to be opened at application launch. If you launch AlertMessage from the command line and enter a value



```
$
$ cd /boot/Metrowerks/MyBeBookEx/C09/P04_AlertMessage
$ ls
(Objects)          MyDrawView.h        MyHelloWorld.h
AlertMessage       MyHelloWindow.cpp   MyHelloWorld.rsrc
AlertMessage.proj  MyHelloWindow.h
MyDrawView.cpp     MyHelloWorld.cpp
$ AlertMessage 3
$
```

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.



Maximum windows open

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

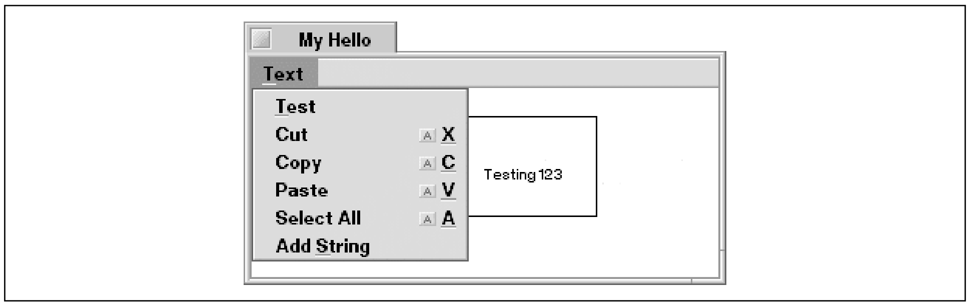


Figure 9-8. The window of the ClipboardMessage program

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);
   }
}
```

In this chapter:
- *Files and the Storage Kit*
- *Using Standard Open and Save Panels*
- *Onward*

10 Files

Many utility programs don't involve file handling, but almost all real-world, full-featured applications do. Before your own best-selling Be application can be considered complete, it will no doubt need to have the capability to open files, save files, or both. In this chapter, you'll see how these file-handling techniques are implemented. To open a file, your program will need to find it on disk; and to save a file, your program will need to specify a location on disk. So before getting into the actual manipulation of files, this chapter introduces you to the BeOS file hierarchy.

Files and the Storage Kit

Up to this point, we've managed to avoid the Storage Kit. Now that we're about to work with persistent data, though, it's time to dig into a number of the classes in this useful kit. The classes of the Storage Kit allow you to write programs that recognize the hierarchy of files on disk, read from and write to files, and study or change file attributes.

There are a number of Storage Kit classes that aid in working with files, including, unsurprisingly, the `BFile` class. But Be also tips its hat to Unix programmers by supporting standard POSIX file functions such as `open()`, `close()`, `read()`, and `write()`. If you have a Unix programming background, you'll feel right at home using POSIX functions to implement file-handling tasks such as saving a document's data to a file. If you aren't comfortable with Unix, you probably aren't familiar with POSIX. That's okay, because the Storage Kit also defines classes (such as `BFile`) that let you work with files outside the realm of POSIX. In this chapter I'll cover file manipulation using both techniques.

POSIX, or *Portable Operating System Interface for Unix*, is a standard developed so that buyers (particularly the U.S. government) could be assured of purchasing programs that ran on a variety of systems and configurations. A POSIX-compliant program is written to a strict standard so that it is easily ported. It's also designed to run on any POSIX-compliant operating system, which includes most variants of Unix.

File Organization

The BeOS, like the Mac OS, Windows, and Unix, organizes files hierarchically. Files, and the directories (or folders) that hold files, are organized in a hierarchy or tree. Each directory may hold files, other directories, or both. Each item (file or directory) has a single parent directory—a directory in which the item resides. The parent directory of an item may, of course, have a parent of its own. Thus the creation of a hierarchy. The common ancestor for all the files and directories in the hierarchy is a directory referred to as the *root directory*.

A single file, regardless of its place in the hierarchy, is considered to have both an entry and a node. In short, a file's entry is its pathname, or location in the hierarchy, while the file's node is the actual data that makes up the file. These two parts of a file serve different purposes, and one part can be manipulated without affecting the other part. For instance, a file's entry (its pathname) can be altered without changing the file's node (its contents, or data).

Entries

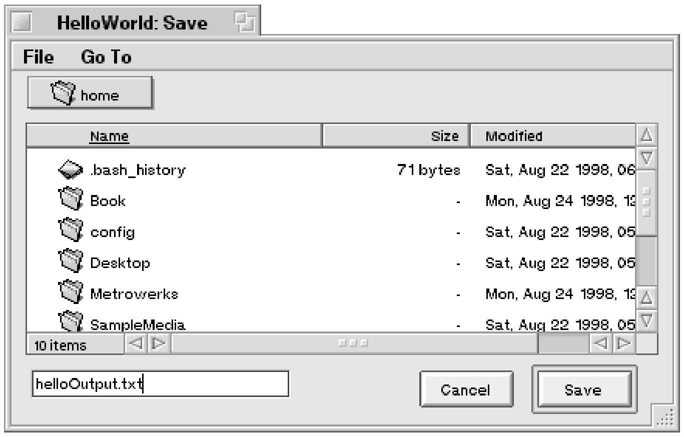
Searching, opening, and saving a file all involve an entry. Your program needs to know, or establish, the location of a file before it can work with it. The `entry_ref` data structure is used to keep track of the entry, or entries, your program is to work with. A Be program relies on an object of the `BEntry` class if it needs to manipulate an entry. In this chapter, you'll see examples that use both the `entry_ref` data structure and the `BEntry` class.

Nodes

To manipulate a file's contents—something done during reading and writing a file—a program works with the file's node. For this purpose, the BeOS defines a `node_ref` data structure and a `BNode` class. The `BFile` class is derived from `BNode`, and it is the `BFile` class that I'll use in this chapter's examples.

Using Standard Open and Save Panels

An operating system with a graphical user interface typically provides standardized means for opening and saving files. That maintains consistency from program to program, and allows the user to work intuitively with files regardless of the program being used. The BeOS is no exception. In Figure 10-1, you see the standard Save file panel. The Open file panel looks similar to the Save file panel, with the primary difference being the Open file panel's omission of the text view used in the Save file panel to provide a name for a file.



Name	Size	Modified
.bash_history	71 bytes	Sat, Aug 22 1998, 06
Book	-	Mon, Aug 24 1998, 1
config	-	Sat, Aug 22 1998, 05
Desktop	-	Sat, Aug 22 1998, 05
Metrowerks	-	Mon, Aug 24 1998, 1
SampleMedia	-	Sat, Aug 22 1998, 05

helloOutput.txt

Figure 10-1. The standard Save file panel

Using BFilePanel to Create a Panel

The Storage Kit defines a single `BFilePanel` class that's used to create both a Save file panel object and an Open file panel object. The `BFilePanel` constructor, shown below, is a bit scary-looking, but as you'll soon see, most of the arguments can be ignored and left at their default values:

```
BFilePanel(file_panel_mode  mode = B_OPEN_PANEL,
           BMessenger       *target = NULL,
           entry_ref        *start_directory = NULL,
           uint32           node_flavors = 0,
           bool             allow_multiple_selection = true,
           BMessage         *message = NULL,
           BRefFilter       *filter = NULL,
           bool             modal = false,
           bool             hide_when_done = true);
```

Of the numerous arguments, by far the one of most importance is the first—`mode`. The type of panel the `BFilePanel` constructor creates is established by the value of `mode`. Once a `BFilePanel` object is created, there's no way to change its type, so you need to know in advance what purpose the panel is to serve. To specify that the new `BFilePanel` object be a Save file panel, pass the Be-defined constant `B_SAVE_PANEL`:

```
BFilePanel        *fSavePanel;

savePanel = new BFilePanel(B_SAVE_PANEL);
```

To instead specify that the new object be an Open file panel, pass the Be-defined constant `B_OPEN_PANEL`. Or, simply omit the parameter completely and rely on the default value for this argument (see the above constructor definition):

```
BFilePanel        *fOpenPanel;

fOpenPanel = new BFilePanel();
```

Creating a new panel doesn't display it. This allows your program to create the panel at any time, then display it only in response to the user's request. For an Open file panel, that's typically when the user chooses the Open item from the File menu. For the Save file panel, the display of the panel comes when the user chooses the Save As item from the **File** menu. In response to the message issued by the system to the appropriate `MessageReceived()` function, your program will invoke the `BFilePanel` function `Show()`, as done here for the `fOpenPanel` object:

```
fOpenPanel->Show();
```

Assuming you follow normal conventions, the files shown are the contents of the current working directory. When a panel is displayed, control is in the hands of the user. Once the user confirms a choice (whether it's a file selection in the Open file panel, a click on the Save button in the Save file panel, or a click on the Cancel button in either type of panel), a message is automatically sent by the system to the panel's target. By default the panel's target is the application object, but this can be changed (either in the `BFilePanel` constructor or by invoking the panel object's `SetTarget()` function). The message holds information about the selected file or files (for an Open file panel) or about the file that's to be created and used to hold a document's data (for a Save file panel). The details of how to handle the message generated in response to a user's dismissing a panel appear in the next sections.

The File-Handling Base Project

In Chapter 8, *Text*, you saw ScrollViewWindow, a program that displays a window with a text area that occupies the entire content area of the window. A simple text editor lends itself well to file opening and saving, so in this chapter I'll modify ScrollViewWindow to make it capable of opening existing text files and saving the current document as a text file. Figure 10-2 shows the window the new FileBase program displays.

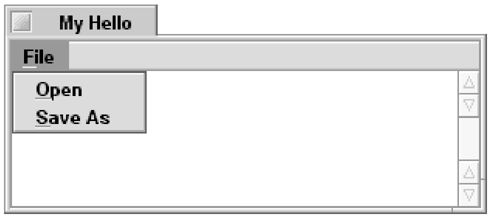


Figure 10-2. The window of the FileBase program

While the FileBase program includes menu items for opening and saving files, you'll soon see that the program isn't up to handling those chores yet. Choosing the Open menu item displays the Open file panel, but selecting a file from the panel's list has no effect—the panel is simply dismissed. The Save As menu item displays the Save file panel, but typing a name and clicking the Save button does nothing more than dismiss the panel. FileBase serves as the basis (hence the name) for a file-handling program. I'll revise FileBase twice in this chapter: once to add file-saving abilities, and one more time to include file-opening powers. With the preliminaries taken care of here in FileBase, those two examples can focus strictly on the tasks of saving and opening a file.

The Application class

FileBase is a spin-off of ScrollViewWindow. A quick look at how that Chapter 8 program has been enhanced makes it easier to follow the upcoming file saving and opening changes. While looking over the old code, I'll insert a few changes here and there to ready the program for the file-handling code. The changes begin in the `MyHelloApplication` class definition. In any Be program, a Save file panel is associated with a particular window—the user will choose Save As to save the contents of the frontmost window to a file on disk. An Open file panel, though, is typically associated with the application itself. In the `MyHelloApplication` class, a `BFilePanel` data member has been added to serve as the Open file panel object, while a `MessageReceived()` function has been added to support the handling of the message generated by the user choosing the Open menu item:

```
class MyHelloApplication : public BApplication {

   public:
                        MyHelloApplication();
      virtual void      MessageReceived(BMessage *message);
   private:
      MyHelloWindow    *fMyWindow;
      BFilePanel       *fOpenPanel;
};
```

The `main()` function remains untouched—it still serves as the vehicle for creating the application object and starting the program running:

```
main()
{
   MyHelloApplication  *myApplication;

   myApplication = new MyHelloApplication();
   myApplication->Run();

   delete(myApplication);
   return(0);
}
```

The application constructor now includes the single line of code needed to create a new `BFilePanel` object. No `mode` parameter is passed, so by default the new object is an Open file panel. Recall that the `BFilePanel` constructor creates the panel, but doesn't display it.

```
MyHelloApplication::MyHelloApplication()
     : BApplication("application/x-dps-mywd")
{
   BRect  aRect;

   fOpenPanel = new BFilePanel();

   aRect.Set(20, 30, 320, 230);
   fMyWindow = new MyHelloWindow(aRect);
}
```

As you'll see ahead, when the user chooses Open from the File menu, the application generates a message that's delivered to the application object. Thus the need for a `MessageReceived()` function for the application class. Here the choosing of the Open menu item does nothing more than display the previously hidden Open file panel:

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

      case MENU_FILE_OPEN_MSG:
         fOpenPanel->Show();
         break;

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

The window class

The window's one menu now holds an Open item and a Save As item, so two message constants are necessary:

```
#define   MENU_FILE_OPEN_MSG          'opEN'
#define   MENU_FILE_SAVEAS_MSG        'svAs'
```

The window class functions are the same, but the data members are a bit different. The Chapter 8 incarnation of the text editing program defined a `BView`-derived class that filled the window and contained the window's text view and scroll view. Here I'm content to place the `BTextView` and `BScrollView` objects directly in the window's top view. Thus the `MyHelloWindow` class doesn't include a `MyDrawView` member (there is no longer a `MyDrawView` class), but it does include the `fTextView` and `fScrollView` members that were formerly a part of `MyDrawView`. The class now also defines a `BFilePanel` object to serve as the window's Save file panel:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage *message);

   private:
      BMenuBar      *fMenuBar;
      BTextView     *fTextView;
      BScrollView   *fScrollView;
      BFilePanel    *fSavePanel;
};
```

Which is the better way to include views in a window—by defining an all-encompassing view to nest the other views in, or by simply relying on the window's top view? It's partially a matter of personal preference. It's also a matter of whether your program will make changes that affect the look of a window's background. The FileBase program won't change the overall look of its window (that is, it won't do something such as change the window's background color), so simply including the views in the window's top view makes sense. It also allows for a good example of an alternate implementation of the Chapter 8 way of doing things!

The `MyHelloWindow` constructor begins in typical fashion with the setup of the menu and its items:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
```

```
{
        BMenu      *menu;
        BMenuItem  *menuItem;
        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(menuItem = new BMenuItem("Open",
                                  new BMessage(MENU_FILE_OPEN_MSG)));
        menuItem->SetTarget(be_app);
        menu->AddItem(new BMenuItem("Save As",
                             new BMessage(MENU_FILE_SAVEAS_MSG)));
```

If you're referencing the Chapter 8 program from which FileBase is derived, you'll see that the `MyHelloWindow` constructor just lost a few lines of code. The ScrollViewWindow version of the constructor started with code to resize the window size-defining rectangle `frame`. Since the FileBase window no longer includes a `MyDrawView` under the menubar, there's no need to resize the frame such that it fills the window, less the menubar area.

The `MyHelloWindow` constructor next establishes the size of the text view and the text rectangle nested in that view. The constructor creates the `BTextView` object, makes it a part of a `BScrollView` object, and then adds the scroll view to the window:

```
    BRect  viewFrame;
    BRect  textBounds;

    viewFrame = Bounds();
    viewFrame.top = MENU_BAR_HEIGHT + 1.0;
    viewFrame.right -= B_V_SCROLL_BAR_WIDTH;

    textBounds = viewFrame;
    textBounds.OffsetTo(B_ORIGIN);
    textBounds.InsetBy(TEXT_INSET, TEXT_INSET);

    fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                              B_FOLLOW_ALL, B_WILL_DRAW);

    fScrollView = new BScrollView("MyScrollView", fTextView,
                                  B_FOLLOW_ALL, 0, false, true);
    AddChild(fScrollView);
```

Finally, the Save file panel is created and the window displayed:

```
        fSavePanel = new BFilePanel(B_SAVE_PANEL, BMessenger(this), NULL,
                                    B_FILE_NODE, false);
            Show();
    }
```

Unlike the creation of the Open file panel, the creation of the Save file panel requires that a few parameters be passed to the `BFilePanel` constructor. You already know that the first `BFilePanel` argument, `mode`, establishes the type of panel to be created. The other parameters are worthy of a little explanation.

The second argument, `target`, is used to define the target of the message the system will deliver to the application in response to the user's dismissal of the panel. The default target is the application object, which works well for the Open file panel. That's because the Open file panel affects the application, and is referenced by an application data member. The Save file panel, on the other hand, affects the window, and is referenced by a window data member. So I want the message sent to the window object rather than the application object. Passing `BMessenger` with the window object as the target makes that happen. There's no need to preface `BMessenger()` with `new`, as the `BFilePanel` constructor handles the task of allocating memory for the message.

The other argument that needs to be set is the fifth one—`allow_multiple_selection`. Before passing a value for the fifth argument, I need to supply values for the third and fourth arguments. The third argument, `panel_directory`, specifies the directory to list in the Open file panel when that panel is first displayed. Passing a value of `NULL` here keeps the default behavior of displaying the current working directory. The fourth argument, `node_flavors`, is used to specify the type of items considered to be valid user selections. The Be-defined constant `B_FILE_NODE` is the default flavor—it specifies that a file (as opposed to, say, a directory) is considered an acceptable user choice. The argument I'm really interested in is the fifth one—`allow_multiple_selection`. The default value for this argument is `true`. FileBase doesn't support the simultaneous opening of multiple files, so a value of `false` needs to be passed here.

FileBase terminates when the user closes a window. As you've seen before, that action results in the hook function `QuitRequested()` being invoked:

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

    return(true);
}
```

An application-defined message is issued in response to the user choosing Save As from the File menu. In response to that message, the program shows the already-created Save file panel by invoking the `BFilePanel` function `Show()`.

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_FILE_SAVEAS_MSG:
         fSavePanel->Show();
         break;

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

Saving a File

Converting FileBase to a program that fully supports file saving is a straightforward process. No changes are needed in the `MyHelloApplication` class. The `MyHelloWindow` class needs one new member function, a routine that implements the saving of a window's data to a file in response to the user's dismissing the Save file panel. The SaveAsFile example program adds that one function—the `Save()` routine holds the code that implements the saving of a document's text to disk. So the class declaration now contains a public declaration of `Save()`:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();
      virtual void      MessageReceived(BMessage *message);
      status_t          Save(BMessage *message);

   private:
      BMenuBar         *fMenuBar;
      BTextView        *fTextView;
      BScrollView      *fScrollView;
      BFilePanel       *fSavePanel;
};
```

When a Save file panel is dismissed, the system sends a `B_SAVE_REQUESTED` message to the affected application. This message is directed to the `MessageReceived()` function of the Save file panel's target. Recall that in the FileBase program the second parameter passed to the `BFilePanel` constructor specified that the window be the target of the Save file panel. So the window's implementation of `MessageReceived()` receives the message. Embedded in this message is the pathname at which the file is to be saved. `MessageReceived()` passes this information on to the application-defined routine `Save()`:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_FILE_SAVEAS_MSG:
         fSavePanel->Show();
         break;

      case B_SAVE_REQUESTED:
         Save(message);
         break;

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

On the following pages we'll look first at saving a file using POSIX, then at saving a file with the Storage Kit.

Using POSIX to save a file

To work with files, you can use either the `BFile` class or a POSIX file of type `FILE`. Here file-saving will be performed using the `FILE` type—see the sections "Using the Storage Kit to save a file" and "Opening a File," for examples of working with the `BFile` class.

The `Save()` function begins with a number of local variable declarations. Each is described as it's encountered in the `Save()` routine:

```
status_t MyHelloWindow::Save(BMessage *message)
{
   entry_ref   ref;
   const char  *name;
   BPath       path;
   BEntry      entry;
   status_t    err = B_OK;
   FILE        *f;
```

The message received by `MessageReceived()` and passed on to `Save()` has a `what` field of `B_SAVE_REQUESTED`, a `directory` field that holds an `entry_ref`, and a `name` field that holds the user-entered filename string describing a single entry in a directory. The `directory` field's `entry_ref` structure points to the directory to which the user specified the file is to be saved. Invoking the `BMessage` function `FindRef()` strips out this reference and saves it to the `entry_ref` variable `ref`:

```
   if (err = message->FindRef("directory", &ref) != B_OK)
      return err;
```

Next, the filename is retrieved from the message. The `BMessage` function `FindString()` saves the message's `name` field to the string variable `name`:

```
    if (err = message->FindString("name", &name) != B_OK) {
       return err;
```

The next several steps are performed to get the directory and name into a form that can be passed to a file-opening routine. Recall that a file consists of an entry (a location) and a node (data). The entry can be represented by an `entry_ref` or a `BEntry` object. Currently the entry is in the form of an `entry_ref`. Here the entry is stored in a `BEntry` object. The `BEntry` function `SetTo()` handles that task:

```
    if (err = entry.SetTo(&ref) != B_OK)
       return err;
```

A `BPath` object *normalizes* a pathname. That is, it reformats a pathname to clean it up by, say, excluding a trailing slash (such as `/boot/myDir/`). The `BEntry` function `GetPath()` is used to store the `BEntry` information as a `BPath` object. Here the `BPath` object `path` is first set to the directory, then the filename is appended to the directory:

```
    entry.GetPath(&path);
    path.Append(name);
```

The somewhat convoluted journey from the message's `entry_ref` to a `BEntry` object to a `BPath` object is complete. Now the file directory and name appear together in a form that can be used in the POSIX file opening function `fopen()`:

```
    if (!(f = fopen(path.Path(), "w")))
       return B_ERROR;
```

With a new file opened, it's safe to write the window's data. The POSIX file function `fwrite()` can be used for that job. The data to write is the text of the window's text view. That text is retrieved by calling the `BTextView` function `Text()`. The number of bytes the text occupies can be obtained from the `BTextView` function `TextLength()`. After writing the data to the file, call the POSIX file function `fclose()`:

```
    err = fwrite(fTextView->Text(), 1, fTextView->TextLength(), f);
    fclose(f);

    return err;
 }
```

Using the Storage Kit to save a file

Using POSIX is straightforward, but so too is using the Storage Kit. Here I'll modify the previous section's version of the application-defined `Save()` function so that saving the file is done with a reliance on the Storage Kit rather than on POSIX:

```
    status_t MyHelloWindow::Save(BMessage *message)
    {
       entry_ref   ref;
       const char  *name;
       status_t    err;
```

The `ref` and `name` variables are again declared for use in determining the directory to save the file to and the name to give that file. The `err` variable is again present for use in error checking. Gone are the `BPath` variable `path`, the `BEntry` variable `entry`, and the `FILE` variable `f`.

The above code is unchanged from the previous version of `Save()`. First, `FindRef()` is used to strip the directory in which the file should be saved from the message that was passed to `Save()`. Then `FindString()` is invoked to retrieve the filename from the same message:

```
       if (err = message->FindRef("directory", &ref) != B_OK)
          return err;
       if (err = message->FindString("name", &name) != B_OK) {
          return err;
```

Now comes some new code. The location to which the file is to be saved is contained in the `entry_ref` variable `ref`. This variable is used as the argument in the creation of a `BDirectory` object. To ensure that the initialization of the new directory was successful, call the inherited `BNode` function `InitCheck()`:

```
       BDirectory  dir(&ref);
       if (err = dir.InitCheck() != B_OK)
          return err;
```

A `BFile` object can be used to open an existing file or to create and open a new file. Here we need to create a new file. Passing the directory, filename, and an open mode does the trick. The open mode value is a combination of flags that indicate such factors as whether the file is to have read and/or write permission and whether a new file is to be created if one doesn't already exist in the specified directory. After creating the new file, invoke the `BFile` version of `InitCheck()` to verify that file creation was successful:

```
    BFile  file(&dir, name, B_READ_WRITE | B_CREATE_FILE);
       if (err = file.InitCheck() != B_OK)
          return err;
```

With a new file opened, it's time to write the window's data. Instead of the POSIX file function `fwrite()`, here I use the `BFile` function `Write()`. The first argument is the text to write to the file, while the second argument specifies the number of bytes to write. Both of the `BView` functions `Text()` and `TextLength()` were described in the POSIX example of file saving. After the data is written there's no need to explicitly close the file—a file is automatically closed when its `BFile` object is deleted (which occurs when the `Save()` function exits):

```
        err = file.Write(fTextView->Text(), fTextView->TextLength());

        return err;
    }
```

Opening a File

You've just seen how to save a file's data using Be classes to work with the file's path and standard POSIX functions for performing the actual data writing. Here I'll dispense with the POSIX and go with the `BFile` class. The last example, SaveAsFile, was derived from the FileBase program. I'll carry on with the example by now adding to the SaveAsFile code such that the OpenSaveAsFile example becomes capable of both saving a text file (using the already developed POSIX file-saving code) and opening a text file (using the `BFile` class).

When an Open file panel is dismissed, the system responds by sending the application a `B_REFS_RECEIVED` message that specifies which file or files are to be opened. Rather than appearing at the target's `MessageReceived()` routine, though, this message is sent to the target's `RefsReceived()` function. The Open file panel indicates that the application is the panel's target, so a `RefsReceived()` function needs to be added to the application class:

```
    class MyHelloApplication : public BApplication {

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

       private:
          MyHelloWindow     *fMyWindow;
          BFilePanel        *fOpenPanel;
    };
```

The implementation of `RefsReceived()` begins with the declaration of a few local variables:

```
    void MyHelloApplication::RefsReceived(BMessage *message)
    {
       BRect      aRect;
       int32      ref_num;
       entry_ref  ref;
       status_t   err;
```

The rectangle `aRect` defines the boundaries of the window to open:

```
       aRect.Set(20, 30, 320, 230);
```

The integer `ref_num` serves as a loop counter. While the Open file panel used in the OpenSaveAsFile example allows for only one file selection at a time, the program might be adapted later to allow for multiple file selections in the Open file panel. Creating a loop that opens each file is an easy enough task, so I'll prepare for a program change by implementing the loop now:

```
ref_num = 0;
   do {
      if (err = message->FindRef("refs", ref_num, &ref) != B_OK)
         return;
      new MyHelloWindow(aRect, &ref);
      ref_num++;
   } while (1);
}
```

The message received by `RefsReceived()` has a `what` field of `B_REFS_RECEIVED` and a `refs` field that holds an `entry_ref` for each selected file. Invoking the `BMessage` function `FindRef()` strips out one reference and saves it to the `entry_ref` variable `ref`. The `ref_num` parameter serves as an index to the `refs` array of `entry_refs`. After the last entry is obtained (which will be after the first and only entry in this example), an error occurs, breaking the otherwise infinite `do-while` loop.

With an entry obtained, a new window is created. Note that the `MyHelloWindow` constructor now receives two parameters: the boundary-defining rectangle the constructor always has, and a new `entry_ref` parameter that specifies the location of the file to open. Rather than change the existing constructor, the `MyHelloWindow` class now defines two constructors: the original and the new two-argument version:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
                     MyHelloWindow(BRect frame, entry_ref *ref);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage *message);
      status_t       Save(BMessage *message);

   private:
      void           InitializeWindow(void);
      BMenuBar       *fMenuBar;
      BTextView      *fTextView;
      BScrollView    *fScrollView;
      BFilePanel     *fSavePanel;
};
```

When the program is to create a new, empty window, the original `MyHelloWindow` constructor is called. When the program needs to instead create a new window that is to hold the contents of an existing file, the new `MyHelloWindow` constructor is invoked.

The two `MyHelloWindow` constructors will create similar windows: each will be the same size, have the same menubar, and so forth. So the two constructors share quite a bit of common code. To avoid writing redundant code, the `MyHelloWindow` class now defines a new routine that holds this common code. Each constructor invokes this new `InitializeWindow()` routine. The file-opening version of the `MyHelloWindow` constructor then goes on to implement file handling.

Note in the above-listed `MyHelloWindow` class that the `InitializeWindow()` routine is declared `private`. It will be invoked only by other `MyHelloWindow` member functions, so there's no need to allow outside access to it. Because all of the code from the original version of the `MyHelloWindow` constructor, with the exception of the last line (the call to `Show()`), was moved wholesale to the `InitializeWindow()` routine, there's no need to show the entire listing for this new function. Instead, a summary is offered below. To see the actual code, refer back to the walk-through of the `MyHelloWindow` constructor in this chapter's "The File-Handling Base Project" section.

```
void MyHelloWindow::InitializeWindow( void )
{
   // menu code

   // text view code

   // Save file panel code
}
```

Almost all of the code found in the original version of the `MyHelloWindow` constructor has been moved to `InitializeWindow()`, so the original, one-argument version of the constructor shrinks to this:

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

   Show();
}
```

The new two-argument version of the constructor begins similarly:

```
MyHelloWindow::MyHelloWindow(BRect frame, entry_ref *ref)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   InitializeWindow();
```

With the window set up, it's on to the file-opening code. In the file-saving example, standard POSIX functions were used. Here a `BFile` object and `BFile` functions are instead used—beginning with the declaration of a `BFile` object:

```
BFile  file;
```

The second parameter passed to this `MyHelloWindow` constructor is the `entry_ref` for the file to open. That `ref` variable is used in a call to the `BFile` function `SetTo()` to assign the `BFile` object a file to open. With the successful assignment of a file to open, it's on to the actual file-opening. That begins with the declaration of a couple of variables:

```
if (file.SetTo(ref, B_READ_ONLY) == B_OK)
{
   off_t   length;
   char   *text;
```

The size of the file to open is determined, and returned, by the `BFile` function `GetSize()`:

```
   file.GetSize(&length);
```

Sufficient memory is allocated for the file's contents by a call to `malloc()`:

```
   text = (char *) malloc(length);
```

Now it's time to read the file's data. The `BFile` function `Read()` handles that chore. The data is text, so it's saved to the character pointer `text`. Invoking the `BTextView` function `SetText()` sets the text of the window's text view to the read-in text, while a call to `free()` releases from memory the no-longer-needed file data. With the window set up and the text view holding the data to display, there's nothing left to do but display the window with a call to `Show()`:

```
      if (text && (file.Read(text, length) >= B_OK))
         fTextView->SetText(text, length);
      free(text);
   }
   Show();
}
```

Onward

This chapter's OpenSaveAsFile example is the most complete program in this book—it actually does something quite useful! Using the techniques presented in the preceding chapters, you should be able to turn OpenSaveAsFile into a real-world application. Begin by polishing up the File menu. Add a New item—that requires just a few lines of code. Also add a Quit item to provide a more graceful means of exiting the program. Chapter 8 covered text editing in detail—use that chapter's techniques to add a complete, functioning Edit menu. Those changes will result in a useful, functional text editor. If you want to develop a program that's more graphics-oriented, go ahead—Chapter 4, *Windows, Views, and Messages*, and Chapter 5, *Drawing*, hold the information to get you started. If you take that route, then you can always include the text-editing capabilities covered here and in Chapter 8 as a part of your program. For example, a graphics-oriented application could include a built-in text editor that allows the user to enter and save notes.

Regardless of the type of application you choose to develop, check out Be's web site at *http://www.be.com/*. In particular, you'll want to investigate their online developer area for tips, techniques, and sample code. For reference documentation, consider the Be Book, Be's own HTML-formatted documentation. For a more complete hardcopy version of that book, look into obtaining one or both of the O'Reilly & Associates books *Be Developer's Guide* and *Be Advanced Topics*.

Index