In this chapter:
- *Introduction to Controls*
- *Buttons*
- *Picture Buttons*
- *Checkboxes*
- *Radio Buttons*
- *Text Fields*
- *Multiple Control Example Project*

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

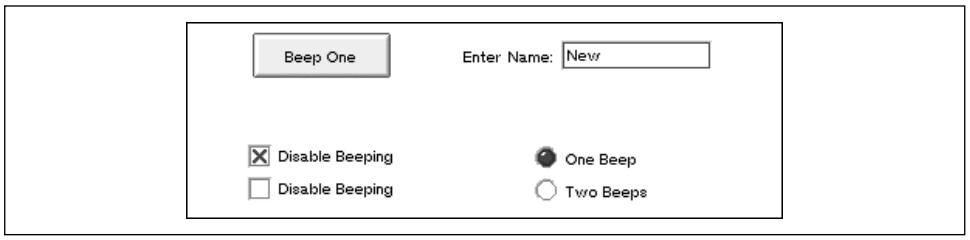

Figure 6-1. Examples of button, text field, checkbox, and radio button controls

underlined when it's current. Once current, other key presses (typically the Return and Enter key) affect that control.

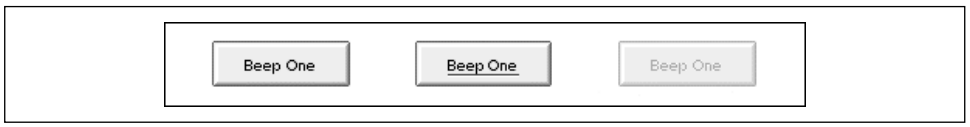

Figure 6-2. A button control that's (from left to right) enabled, current, and disabled

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.

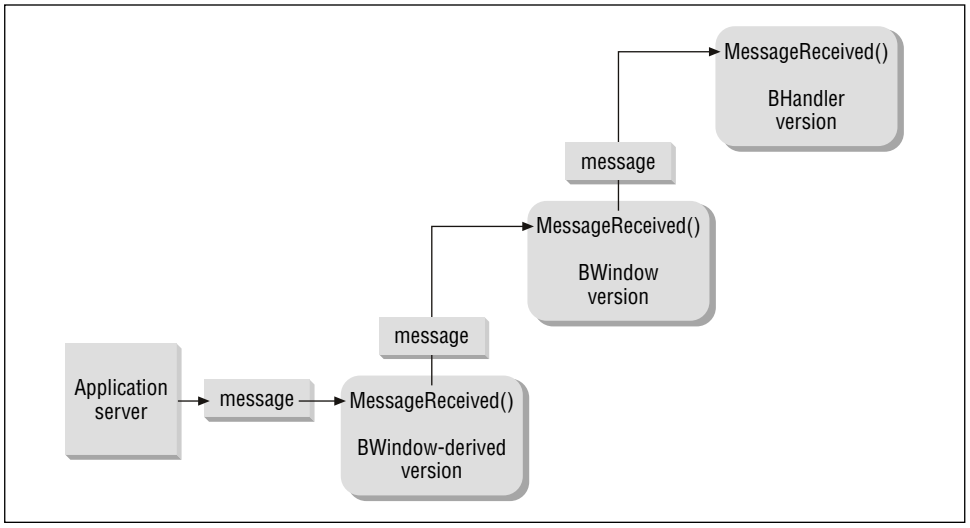

Figure 6-3. Message passed to parent class's version of MessageReceived()

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.

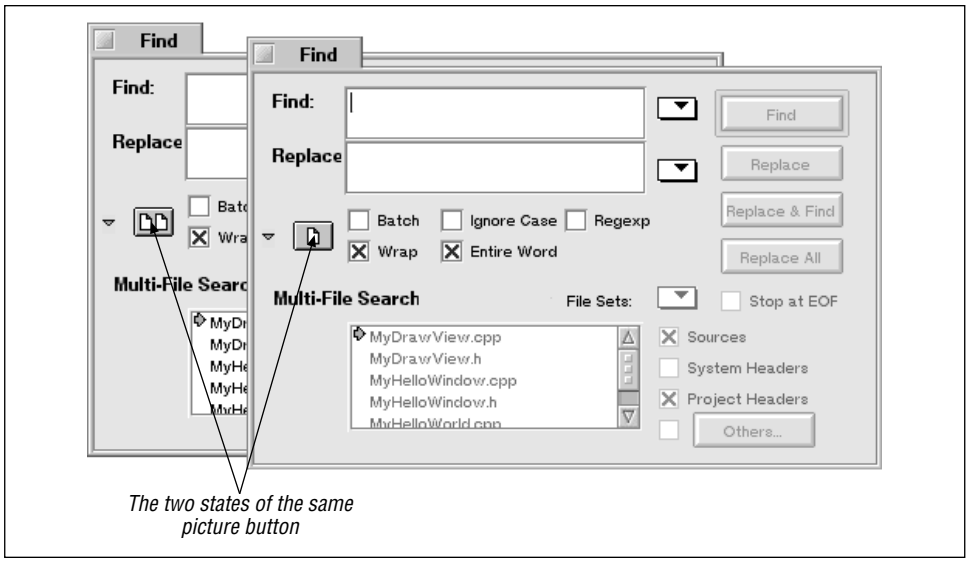

Figure 6-5. The Find window of the BeIDE provides an example of a picture button

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.

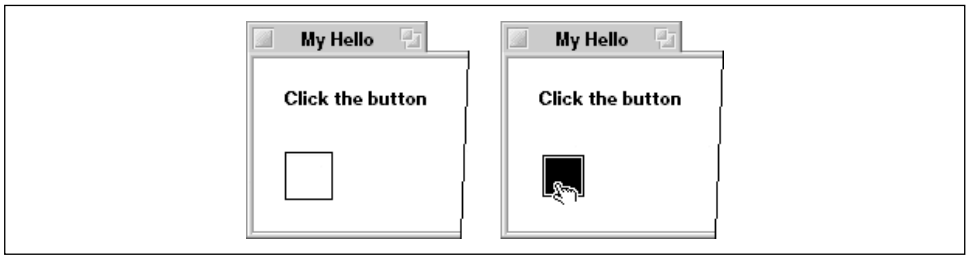

Figure 6-6. The window that results from running the PictureButton program

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 check-

box 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.

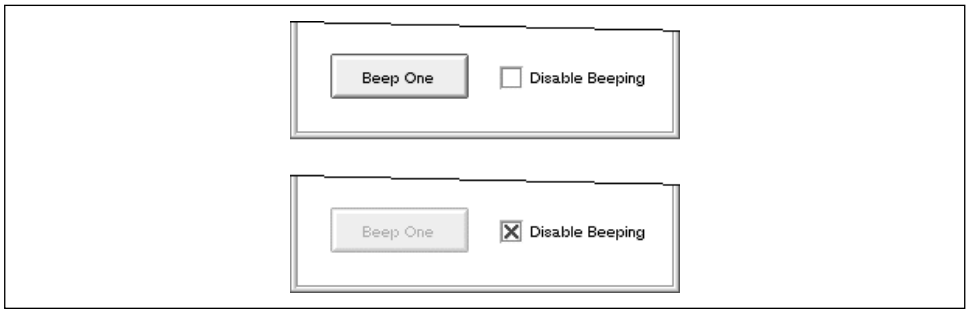

Figure 6-7. The windows that result from running the CheckBoxNow program

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 CheckBoxLater project demonstrates this approach. Figure 6-7 shows that the CheckBoxLater 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:

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.

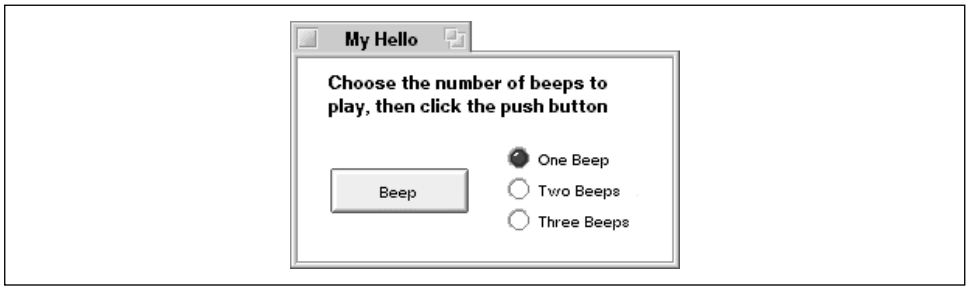

Figure 6-9. The window that results from running the RadioButtonGroup program

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";
```

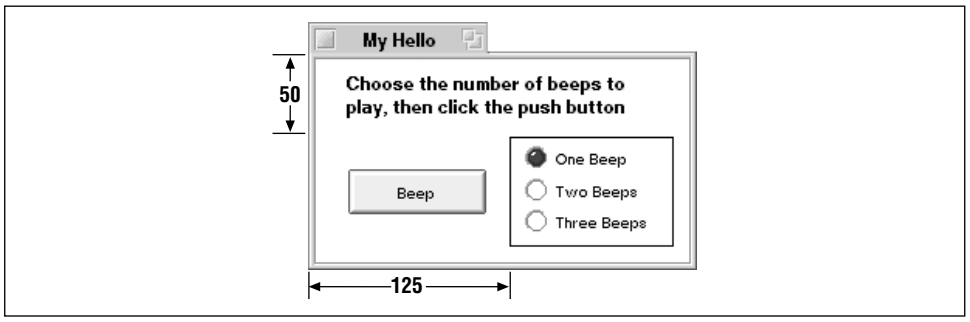

Figure 6-10. A group of radio buttons can reside in their own view

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

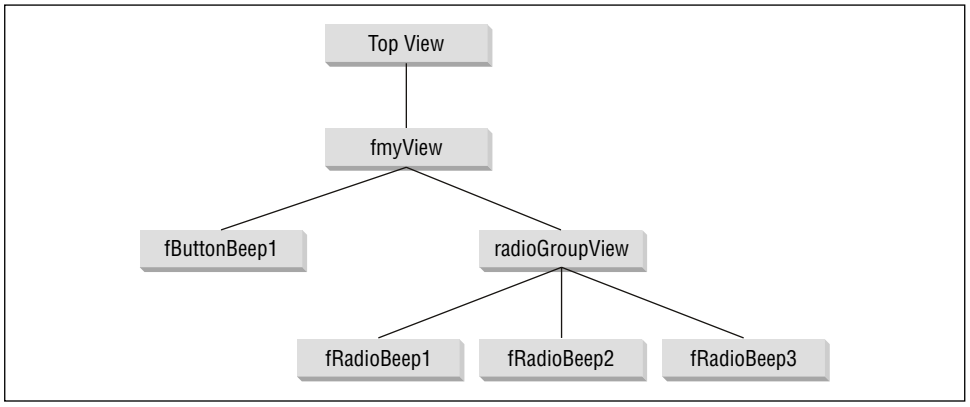

Figure 6-11. The view hierarchy for the RadioButtonGroup window

`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()`.

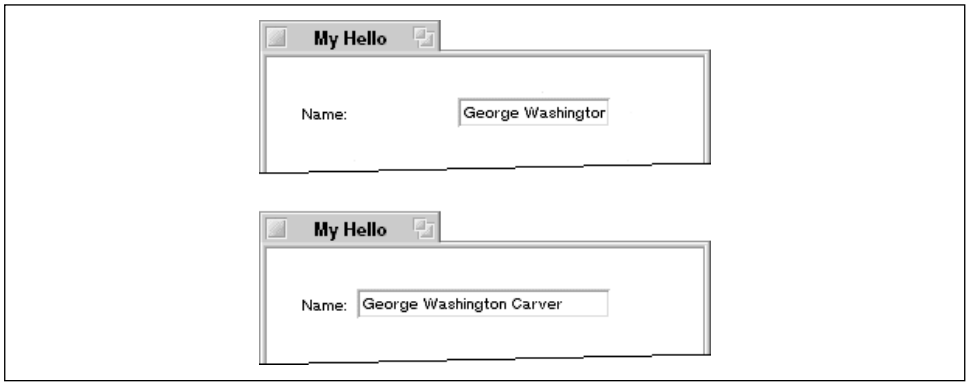

Figure 6-12. The two areas of a text field control can be proportioned in different ways

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

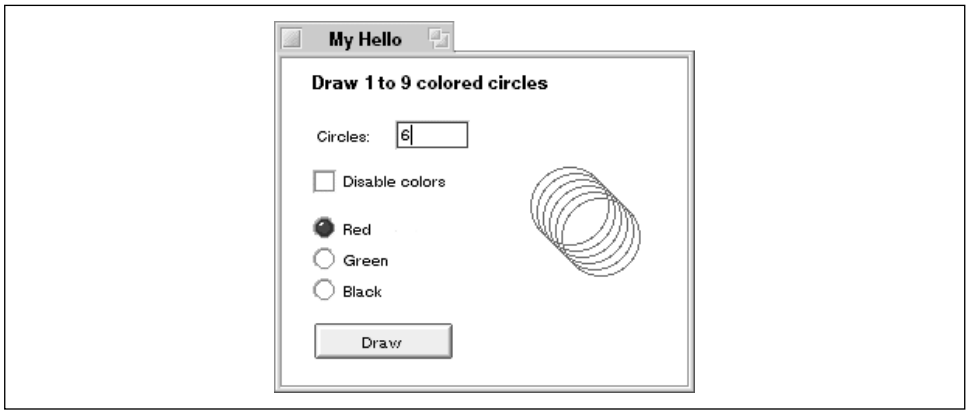

Figure 6-14. The window that results from running the ControlDemo program

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()`!