In this chapter:
- *Files and the Storage Kit*
- *Using Standard Open and Save Panels*
- *Onward*

10 *Files*

Many utility programs don't involve file handling, but almost all real-world, full-featured applications do. Before your own best-selling Be application can be considered complete, it will no doubt need to have the capability to open files, save files, or both. In this chapter, you'll see how these file-handling techniques are implemented. To open a file, your program will need to find it on disk; and to save a file, your program will need to specify a location on disk. So before getting into the actual manipulation of files, this chapter introduces you to the BeOS file hierarchy.

Files and the Storage Kit

Up to this point, we've managed to avoid the Storage Kit. Now that we're about to work with persistent data, though, it's time to dig into a number of the classes in this useful kit. The classes of the Storage Kit allow you to write programs that recognize the hierarchy of files on disk, read from and write to files, and study or change file attributes.

There are a number of Storage Kit classes that aid in working with files, including, unsurprisingly, the `BFile` class. But Be also tips its hat to Unix programmers by supporting standard POSIX file functions such as `open()`, `close()`, `read()`, and `write()`. If you have a Unix programming background, you'll feel right at home using POSIX functions to implement file-handling tasks such as saving a document's data to a file. If you aren't comfortable with Unix, you probably aren't familiar with POSIX. That's okay, because the Storage Kit also defines classes (such as `BFile`) that let you work with files outside the realm of POSIX. In this chapter I'll cover file manipulation using both techniques.

POSIX, or *Portable Operating System Interface for Unix*, is a standard developed so that buyers (particularly the U.S. government) could be assured of purchasing programs that ran on a variety of systems and configurations. A POSIX-compliant program is written to a strict standard so that it is easily ported. It's also designed to run on any POSIX-compliant operating system, which includes most variants of Unix.

File Organization

The BeOS, like the Mac OS, Windows, and Unix, organizes files hierarchically. Files, and the directories (or folders) that hold files, are organized in a hierarchy or tree. Each directory may hold files, other directories, or both. Each item (file or directory) has a single parent directory—a directory in which the item resides. The parent directory of an item may, of course, have a parent of its own. Thus the creation of a hierarchy. The common ancestor for all the files and directories in the hierarchy is a directory referred to as the *root directory*.

A single file, regardless of its place in the hierarchy, is considered to have both an entry and a node. In short, a file's entry is its pathname, or location in the hierarchy, while the file's node is the actual data that makes up the file. These two parts of a file serve different purposes, and one part can be manipulated without affecting the other part. For instance, a file's entry (its pathname) can be altered without changing the file's node (its contents, or data).

Entries

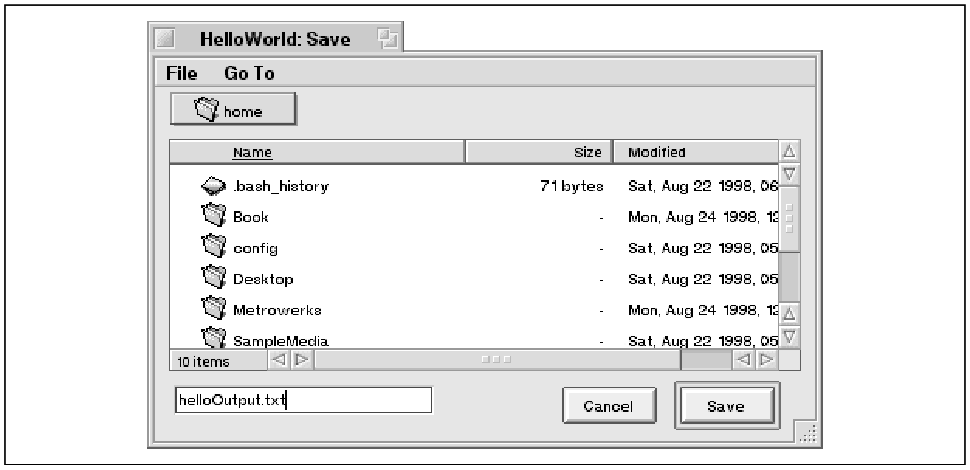
Searching, opening, and saving a file all involve an entry. Your program needs to know, or establish, the location of a file before it can work with it. The `entry_ref` data structure is used to keep track of the entry, or entries, your program is to work with. A Be program relies on an object of the `BEntry` class if it needs to manipulate an entry. In this chapter, you'll see examples that use both the `entry_ref` data structure and the `BEntry` class.

Nodes

To manipulate a file's contents—something done during reading and writing a file—a program works with the file's node. For this purpose, the BeOS defines a `node_ref` data structure and a `BNode` class. The `BFile` class is derived from `BNode`, and it is the `BFile` class that I'll use in this chapter's examples.

Using Standard Open and Save Panels

An operating system with a graphical user interface typically provides standardized means for opening and saving files. That maintains consistency from program to program, and allows the user to work intuitively with files regardless of the program being used. The BeOS is no exception. In Figure 10-1, you see the standard Save file panel. The Open file panel looks similar to the Save file panel, with the primary difference being the Open file panel's omission of the text view used in the Save file panel to provide a name for a file.



Name	Size	Modified
.bash_history	71 bytes	Sat, Aug 22 1998, 06
Book	-	Mon, Aug 24 1998, 1
config	-	Sat, Aug 22 1998, 05
Desktop	-	Sat, Aug 22 1998, 05
Metrowerks	-	Mon, Aug 24 1998, 1
SampleMedia	-	Sat, Aug 22 1998, 05

helloOutput.txt

Figure 10-1. The standard Save file panel

Using BFilePanel to Create a Panel

The Storage Kit defines a single `BFilePanel` class that's used to create both a Save file panel object and an Open file panel object. The `BFilePanel` constructor, shown below, is a bit scary-looking, but as you'll soon see, most of the arguments can be ignored and left at their default values:

```
BFilePanel(file_panel_mode  mode = B_OPEN_PANEL,
           BMessenger       *target = NULL,
           entry_ref        *start_directory = NULL,
           uint32           node_flavors = 0,
           bool             allow_multiple_selection = true,
           BMessage         *message = NULL,
           BRefFilter       *filter = NULL,
           bool             modal = false,
           bool             hide_when_done = true);
```

Of the numerous arguments, by far the one of most importance is the first—`mode`. The type of panel the `BFilePanel` constructor creates is established by the value of `mode`. Once a `BFilePanel` object is created, there's no way to change its type, so you need to know in advance what purpose the panel is to serve. To specify that the new BFilePanel object be a Save file panel, pass the Be-defined constant B_SAVE_PANEL:

```
BFilePanel      *fSavePanel;

savePanel = new BFilePanel(B_SAVE_PANEL);
```

To instead specify that the new object be an Open file panel, pass the Be-defined constant B_OPEN_PANEL. Or, simply omit the parameter completely and rely on the default value for this argument (see the above constructor definition):

```
BFilePanel      *fOpenPanel;

fOpenPanel = new BFilePanel();
```

Creating a new panel doesn't display it. This allows your program to create the panel at any time, then display it only in response to the user's request. For an Open file panel, that's typically when the user chooses the Open item from the File menu. For the Save file panel, the display of the panel comes when the user chooses the Save As item from the **File** menu. In response to the message issued by the system to the appropriate MessageReceived() function, your program will invoke the BFilePanel function Show(), as done here for the fOpenPanel object:

```
fOpenPanel->Show();
```

Assuming you follow normal conventions, the files shown are the contents of the current working directory. When a panel is displayed, control is in the hands of the user. Once the user confirms a choice (whether it's a file selection in the Open file panel, a click on the Save button in the Save file panel, or a click on the Cancel button in either type of panel), a message is automatically sent by the system to the panel's target. By default the panel's target is the application object, but this can be changed (either in the BFilePanel constructor or by invoking the panel object's SetTarget() function). The message holds information about the selected file or files (for an Open file panel) or about the file that's to be created and used to hold a document's data (for a Save file panel). The details of how to handle the message generated in response to a user's dismissing a panel appear in the next sections.

The File-Handling Base Project

In Chapter 8, *Text*, you saw ScrollViewWindow, a program that displays a window with a text area that occupies the entire content area of the window. A simple text editor lends itself well to file opening and saving, so in this chapter I'll modify ScrollViewWindow to make it capable of opening existing text files and saving the current document as a text file. Figure 10-2 shows the window the new FileBase program displays.

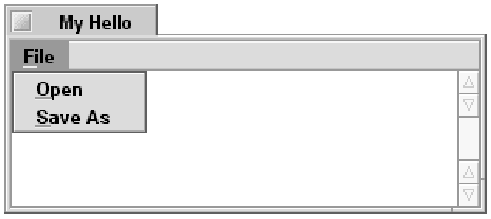


Figure 10-2. The window of the FileBase program

While the FileBase program includes menu items for opening and saving files, you'll soon see that the program isn't up to handling those chores yet. Choosing the Open menu item displays the Open file panel, but selecting a file from the panel's list has no effect—the panel is simply dismissed. The Save As menu item displays the Save file panel, but typing a name and clicking the Save button does nothing more than dismiss the panel. FileBase serves as the basis (hence the name) for a file-handling program. I'll revise FileBase twice in this chapter: once to add file-saving abilities, and one more time to include file-opening powers. With the preliminaries taken care of here in FileBase, those two examples can focus strictly on the tasks of saving and opening a file.

The Application class

FileBase is a spin-off of ScrollViewWindow. A quick look at how that Chapter 8 program has been enhanced makes it easier to follow the upcoming file saving and opening changes. While looking over the old code, I'll insert a few changes here and there to ready the program for the file-handling code. The changes begin in the `MyHelloApplication` class definition. In any Be program, a Save file panel is associated with a particular window—the user will choose Save As to save the contents of the frontmost window to a file on disk. An Open file panel, though, is typically associated with the application itself. In the `MyHelloApplication` class, a `BFilePanel` data member has been added to serve as the Open file panel object, while a `MessageReceived()` function has been added to support the handling of the message generated by the user choosing the Open menu item:

```
class MyHelloApplication : public BApplication {

   public:
                      MyHelloApplication();
      virtual void    MessageReceived(BMessage *message);
   private:
      MyHelloWindow    *fMyWindow;
      BFilePanel       *fOpenPanel;
};
```

The `main()` function remains untouched—it still serves as the vehicle for creating the application object and starting the program running:

```
main()
{
   MyHelloApplication  *myApplication;

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

   delete(myApplication);
   return(0);
}
```

The application constructor now includes the single line of code needed to create a new `BFilePanel` object. No `mode` parameter is passed, so by default the new object is an Open file panel. Recall that the `BFilePanel` constructor creates the panel, but doesn't display it.

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

   fOpenPanel = new BFilePanel();

   aRect.Set(20, 30, 320, 230);
   fMyWindow = new MyHelloWindow(aRect);
}
```

As you'll see ahead, when the user chooses Open from the File menu, the application generates a message that's delivered to the application object. Thus the need for a `MessageReceived()` function for the application class. Here the choosing of the Open menu item does nothing more than display the previously hidden Open file panel:

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

      case MENU_FILE_OPEN_MSG:
         fOpenPanel->Show();
         break;

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

The window class

The window's one menu now holds an Open item and a Save As item, so two message constants are necessary:

```
#define   MENU_FILE_OPEN_MSG           'opEN'
#define   MENU_FILE_SAVEAS_MSG         'svAs'
```

The window class functions are the same, but the data members are a bit different. The Chapter 8 incarnation of the text editing program defined a `BView`-derived class that filled the window and contained the window's text view and scroll view. Here I'm content to place the `BTextView` and `BScrollView` objects directly in the window's top view. Thus the `MyHelloWindow` class doesn't include a `MyDrawView` member (there is no longer a `MyDrawView` class), but it does include the `fTextView` and `fScrollView` members that were formerly a part of `MyDrawView`. The class now also defines a `BFilePanel` object to serve as the window's Save file panel:

```
class MyHelloWindow : public BWindow {

   public:
                       MyHelloWindow(BRect frame);
      virtual bool     QuitRequested();
      virtual void     MessageReceived(BMessage *message);

   private:
      BMenuBar       *fMenuBar;
      BTextView      *fTextView;
      BScrollView    *fScrollView;
      BFilePanel     *fSavePanel;
};
```

Which is the better way to include views in a window—by defining an all-encompassing view to nest the other views in, or by simply relying on the window's top view? It's partially a matter of personal preference. It's also a matter of whether your program will make changes that affect the look of a window's background. The FileBase program won't change the overall look of its window (that is, it won't do something such as change the window's background color), so simply including the views in the window's top view makes sense. It also allows for a good example of an alternate implementation of the Chapter 8 way of doing things!

The `MyHelloWindow` constructor begins in typical fashion with the setup of the menu and its items:

```
MyHelloWindow::MyHelloWindow(BRect frame)
    : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
```

```
{
        BMenu      *menu;
        BMenuItem  *menuItem;
        BRect      menuBarRect;

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

        menu = new BMenu("File");
        fMenuBar->AddItem(menu);
        menu->AddItem(menuItem = new BMenuItem("Open",
                                          new BMessage(MENU_FILE_OPEN_MSG)));
        menuItem->SetTarget(be_app);
        menu->AddItem(new BMenuItem("Save As",
                                     new BMessage(MENU_FILE_SAVEAS_MSG)));
```

If you're referencing the Chapter 8 program from which FileBase is derived, you'll see that the `MyHelloWindow` constructor just lost a few lines of code. The ScrollViewWindow version of the constructor started with code to resize the window size-defining rectangle `frame`. Since the FileBase window no longer includes a `MyDrawView` under the menubar, there's no need to resize the frame such that it fills the window, less the menubar area.

The `MyHelloWindow` constructor next establishes the size of the text view and the text rectangle nested in that view. The constructor creates the `BTextView` object, makes it a part of a `BScrollView` object, and then adds the scroll view to the window:

```
BRect  viewFrame;
BRect  textBounds;

viewFrame = Bounds();
viewFrame.top = MENU_BAR_HEIGHT + 1.0;
viewFrame.right -= B_V_SCROLL_BAR_WIDTH;

textBounds = viewFrame;
textBounds.OffsetTo(B_ORIGIN);
textBounds.InsetBy(TEXT_INSET, TEXT_INSET);

fTextView = new BTextView(viewFrame, "MyTextView", textBounds,
                          B_FOLLOW_ALL, B_WILL_DRAW);

fScrollView = new BScrollView("MyScrollView", fTextView,
                               B_FOLLOW_ALL, 0, false, true);
AddChild(fScrollView);
```

Finally, the Save file panel is created and the window displayed:

```
        fSavePanel = new BFilePanel(B_SAVE_PANEL, BMessenger(this), NULL,
                                    B_FILE_NODE, false);
             Show();
    }
```

Unlike the creation of the Open file panel, the creation of the Save file panel requires that a few parameters be passed to the `BFilePanel` constructor. You already know that the first `BFilePanel` argument, `mode`, establishes the type of panel to be created. The other parameters are worthy of a little explanation.

The second argument, `target`, is used to define the target of the message the system will deliver to the application in response to the user's dismissal of the panel. The default target is the application object, which works well for the Open file panel. That's because the Open file panel affects the application, and is referenced by an application data member. The Save file panel, on the other hand, affects the window, and is referenced by a window data member. So I want the message sent to the window object rather than the application object. Passing `BMessenger` with the window object as the target makes that happen. There's no need to preface `BMessenger()` with `new`, as the `BFilePanel` constructor handles the task of allocating memory for the message.

The other argument that needs to be set is the fifth one—`allow_multiple_selection`. Before passing a value for the fifth argument, I need to supply values for the third and fourth arguments. The third argument, `panel_directory`, specifies the directory to list in the Open file panel when that panel is first displayed. Passing a value of `NULL` here keeps the default behavior of displaying the current working directory. The fourth argument, `node_flavors`, is used to specify the type of items considered to be valid user selections. The Be-defined constant `B_FILE_NODE` is the default flavor—it specifies that a file (as opposed to, say, a directory) is considered an acceptable user choice. The argument I'm really interested in is the fifth one—`allow_multiple_selection`. The default value for this argument is `true`. FileBase doesn't support the simultaneous opening of multiple files, so a value of `false` needs to be passed here.

FileBase terminates when the user closes a window. As you've seen before, that action results in the hook function `QuitRequested()` being invoked:

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

        return(true);
    }
```

An application-defined message is issued in response to the user choosing Save As from the File menu. In response to that message, the program shows the already-created Save file panel by invoking the `BFilePanel` function `Show()`.

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_FILE_SAVEAS_MSG:
         fSavePanel->Show();
         break;

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

Saving a File

Converting FileBase to a program that fully supports file saving is a straightforward process. No changes are needed in the `MyHelloApplication` class. The `MyHelloWindow` class needs one new member function, a routine that implements the saving of a window's data to a file in response to the user's dismissing the Save file panel. The SaveAsFile example program adds that one function—the `Save()` routine holds the code that implements the saving of a document's text to disk. So the class declaration now contains a public declaration of `Save()`:

```
class MyHelloWindow : public BWindow {

   public:
                        MyHelloWindow(BRect frame);
      virtual bool      QuitRequested();
      virtual void      MessageReceived(BMessage *message);
      status_t          Save(BMessage *message);

   private:
      BMenuBar         *fMenuBar;
      BTextView        *fTextView;
      BScrollView      *fScrollView;
      BFilePanel       *fSavePanel;
};
```

When a Save file panel is dismissed, the system sends a `B_SAVE_REQUESTED` message to the affected application. This message is directed to the `MessageReceived()` function of the Save file panel's target. Recall that in the FileBase program the second parameter passed to the `BFilePanel` constructor specified that the window be the target of the Save file panel. So the window's implementation of `MessageReceived()` receives the message. Embedded in this message is the pathname at which the file is to be saved. `MessageReceived()` passes this information on to the application-defined routine `Save()`:

```
void MyHelloWindow::MessageReceived(BMessage* message)
{
   switch(message->what)
   {
      case MENU_FILE_SAVEAS_MSG:
         fSavePanel->Show();
         break;

      case B_SAVE_REQUESTED:
         Save(message);
         break;

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

On the following pages we'll look first at saving a file using POSIX, then at saving a file with the Storage Kit.

Using POSIX to save a file

To work with files, you can use either the `BFile` class or a POSIX file of type `FILE`. Here file-saving will be performed using the `FILE` type—see the sections "Using the Storage Kit to save a file" and "Opening a File," for examples of working with the `BFile` class.

The `Save()` function begins with a number of local variable declarations. Each is described as it's encountered in the `Save()` routine:

```
status_t MyHelloWindow::Save(BMessage *message)
{
   entry_ref   ref;
   const char  *name;
   BPath       path;
   BEntry      entry;
   status_t    err = B_OK;
   FILE        *f;
```

The message received by `MessageReceived()` and passed on to `Save()` has a `what` field of `B_SAVE_REQUESTED`, a `directory` field that holds an `entry_ref`, and a `name` field that holds the user-entered filename string describing a single entry in a directory. The `directory` field's `entry_ref` structure points to the directory to which the user specified the file is to be saved. Invoking the `BMessage` function `FindRef()` strips out this reference and saves it to the `entry_ref` variable `ref`:

```
   if (err = message->FindRef("directory", &ref) != B_OK)
      return err;
```

Next, the filename is retrieved from the message. The `BMessage` function `FindString()` saves the message's `name` field to the string variable `name`:

```
    if (err = message->FindString("name", &name) != B_OK) {
       return err;
```

The next several steps are performed to get the directory and name into a form that can be passed to a file-opening routine. Recall that a file consists of an entry (a location) and a node (data). The entry can be represented by an `entry_ref` or a `BEntry` object. Currently the entry is in the form of an `entry_ref`. Here the entry is stored in a `BEntry` object. The `BEntry` function `SetTo()` handles that task:

```
    if (err = entry.SetTo(&ref) != B_OK)
       return err;
```

A `BPath` object *normalizes* a pathname. That is, it reformats a pathname to clean it up by, say, excluding a trailing slash (such as `/boot/myDir/`). The `BEntry` function `GetPath()` is used to store the `BEntry` information as a `BPath` object. Here the `BPath` object `path` is first set to the directory, then the filename is appended to the directory:

```
    entry.GetPath(&path);
    path.Append(name);
```

The somewhat convoluted journey from the message's `entry_ref` to a `BEntry` object to a `BPath` object is complete. Now the file directory and name appear together in a form that can be used in the POSIX file opening function `fopen()`:

```
    if (!(f = fopen(path.Path(), "w")))
       return B_ERROR;
```

With a new file opened, it's safe to write the window's data. The POSIX file function `fwrite()` can be used for that job. The data to write is the text of the window's text view. That text is retrieved by calling the `BTextView` function `Text()`. The number of bytes the text occupies can be obtained from the `BTextView` function `TextLength()`. After writing the data to the file, call the POSIX file function `fclose()`:

```
    err = fwrite(fTextView->Text(), 1, fTextView->TextLength(), f);
    fclose(f);

    return err;
 }
```

Using the Storage Kit to save a file

Using POSIX is straightforward, but so too is using the Storage Kit. Here I'll modify the previous section's version of the application-defined `Save()` function so that saving the file is done with a reliance on the Storage Kit rather than on POSIX:

```
status_t MyHelloWindow::Save(BMessage *message)
{
   entry_ref   ref;
   const char  *name;
   status_t    err;
```

The `ref` and `name` variables are again declared for use in determining the directory to save the file to and the name to give that file. The `err` variable is again present for use in error checking. Gone are the `BPath` variable `path`, the `BEntry` variable `entry`, and the `FILE` variable `f`.

The above code is unchanged from the previous version of `Save()`. First, `FindRef()` is used to strip the directory in which the file should be saved from the message that was passed to `Save()`. Then `FindString()` is invoked to retrieve the filename from the same message:

```
   if (err = message->FindRef("directory", &ref) != B_OK)
      return err;
   if (err = message->FindString("name", &name) != B_OK) {
      return err;
```

Now comes some new code. The location to which the file is to be saved is contained in the `entry_ref` variable `ref`. This variable is used as the argument in the creation of a `BDirectory` object. To ensure that the initialization of the new directory was successful, call the inherited `BNode` function `InitCheck()`:

```
   BDirectory  dir(&ref);
   if (err = dir.InitCheck() != B_OK)
      return err;
```

A `BFile` object can be used to open an existing file or to create and open a new file. Here we need to create a new file. Passing the directory, filename, and an open mode does the trick. The open mode value is a combination of flags that indicate such factors as whether the file is to have read and/or write permission and whether a new file is to be created if one doesn't already exist in the specified directory. After creating the new file, invoke the `BFile` version of `InitCheck()` to verify that file creation was successful:

```
BFile  file(&dir, name, B_READ_WRITE | B_CREATE_FILE);
   if (err = file.InitCheck() != B_OK)
      return err;
```

With a new file opened, it's time to write the window's data. Instead of the POSIX file function `fwrite()`, here I use the `BFile` function `Write()`. The first argument is the text to write to the file, while the second argument specifies the number of bytes to write. Both of the `BView` functions `Text()` and `TextLength()` were described in the POSIX example of file saving. After the data is written there's no need to explicitly close the file—a file is automatically closed when its `BFile` object is deleted (which occurs when the `Save()` function exits):

```
       err = file.Write(fTextView->Text(), fTextView->TextLength());

       return err;
    }
```

Opening a File

You've just seen how to save a file's data using Be classes to work with the file's path and standard POSIX functions for performing the actual data writing. Here I'll dispense with the POSIX and go with the `BFile` class. The last example, SaveAs-File, was derived from the FileBase program. I'll carry on with the example by now adding to the SaveAsFile code such that the OpenSaveAsFile example becomes capable of both saving a text file (using the already developed POSIX file-saving code) and opening a text file (using the `BFile` class).

When an Open file panel is dismissed, the system responds by sending the application a `B_REFS_RECEIVED` message that specifies which file or files are to be opened. Rather than appearing at the target's `MessageReceived()` routine, though, this message is sent to the target's `RefsReceived()` function. The Open file panel indicates that the application is the panel's target, so a `RefsReceived()` function needs to be added to the application class:

```
    class MyHelloApplication : public BApplication {

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

       private:
          MyHelloWindow     *fMyWindow;
          BFilePanel        *fOpenPanel;
    };
```

The implementation of `RefsReceived()` begins with the declaration of a few local variables:

```
    void MyHelloApplication::RefsReceived(BMessage *message)
    {
       BRect      aRect;
       int32      ref_num;
       entry_ref  ref;
       status_t   err;
```

The rectangle `aRect` defines the boundaries of the window to open:

```
       aRect.Set(20, 30, 320, 230);
```

The integer ref_num serves as a loop counter. While the Open file panel used in the OpenSaveAsFile example allows for only one file selection at a time, the program might be adapted later to allow for multiple file selections in the Open file panel. Creating a loop that opens each file is an easy enough task, so I'll prepare for a program change by implementing the loop now:

```
ref_num = 0;
   do {
      if (err = message->FindRef("refs", ref_num, &ref) != B_OK)
         return;
      new MyHelloWindow(aRect, &ref);
      ref_num++;
   } while (1);
}
```

The message received by RefsReceived() has a what field of B_REFS_RECEIVED and a refs field that holds an entry_ref for each selected file. Invoking the BMessage function FindRef() strips out one reference and saves it to the entry_ref variable ref. The ref_num parameter serves as an index to the refs array of entry_refs. After the last entry is obtained (which will be after the first and only entry in this example), an error occurs, breaking the otherwise infinite do-while loop.

With an entry obtained, a new window is created. Note that the MyHelloWindow constructor now receives two parameters: the boundary-defining rectangle the constructor always has, and a new entry_ref parameter that specifies the location of the file to open. Rather than change the existing constructor, the MyHelloWindow class now defines two constructors: the original and the new two-argument version:

```
class MyHelloWindow : public BWindow {

   public:
                     MyHelloWindow(BRect frame);
                     MyHelloWindow(BRect frame, entry_ref *ref);
      virtual bool   QuitRequested();
      virtual void   MessageReceived(BMessage *message);
      status_t       Save(BMessage *message);

   private:
      void           InitializeWindow(void);
      BMenuBar       *fMenuBar;
      BTextView      *fTextView;
      BScrollView    *fScrollView;
      BFilePanel     *fSavePanel;
};
```

When the program is to create a new, empty window, the original MyHelloWindow constructor is called. When the program needs to instead create a new window that is to hold the contents of an existing file, the new `MyHelloWindow` constructor is invoked.

The two `MyHelloWindow` constructors will create similar windows: each will be the same size, have the same menubar, and so forth. So the two constructors share quite a bit of common code. To avoid writing redundant code, the `MyHelloWindow` class now defines a new routine that holds this common code. Each constructor invokes this new `InitializeWindow()` routine. The file-opening version of the `MyHelloWindow` constructor then goes on to implement file handling.

Note in the above-listed `MyHelloWindow` class that the `InitializeWindow()` routine is declared `private`. It will be invoked only by other `MyHelloWindow` member functions, so there's no need to allow outside access to it. Because all of the code from the original version of the `MyHelloWindow` constructor, with the exception of the last line (the call to `Show()`), was moved wholesale to the `InitializeWindow()` routine, there's no need to show the entire listing for this new function. Instead, a summary is offered below. To see the actual code, refer back to the walk-through of the `MyHelloWindow` constructor in this chapter's "The File-Handling Base Project" section.

```
void MyHelloWindow::InitializeWindow( void )
{
   // menu code

   // text view code

   // Save file panel code
}
```

Almost all of the code found in the original version of the `MyHelloWindow` constructor has been moved to `InitializeWindow()`, so the original, one-argument version of the constructor shrinks to this:

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

   Show();
}
```

The new two-argument version of the constructor begins similarly:

```
MyHelloWindow::MyHelloWindow(BRect frame, entry_ref *ref)
   : BWindow(frame, "My Hello", B_TITLED_WINDOW, B_NOT_ZOOMABLE)
{
   InitializeWindow();
```

With the window set up, it's on to the file-opening code. In the file-saving example, standard POSIX functions were used. Here a `BFile` object and `BFile` functions are instead used—beginning with the declaration of a `BFile` object:

```
      BFile  file;
```

The second parameter passed to this `MyHelloWindow` constructor is the `entry_ref` for the file to open. That `ref` variable is used in a call to the `BFile` function `SetTo()` to assign the `BFile` object a file to open. With the successful assignment of a file to open, it's on to the actual file-opening. That begins with the declaration of a couple of variables:

```
      if (file.SetTo(ref, B_READ_ONLY) == B_OK)
      {
         off_t   length;
         char   *text;
```

The size of the file to open is determined, and returned, by the `BFile` function `GetSize()`:

```
         file.GetSize(&length);
```

Sufficient memory is allocated for the file's contents by a call to `malloc()`:

```
         text = (char *) malloc(length);
```

Now it's time to read the file's data. The `BFile` function `Read()` handles that chore. The data is text, so it's saved to the character pointer `text`. Invoking the `BTextView` function `SetText()` sets the text of the window's text view to the read-in text, while a call to `free()` releases from memory the no-longer-needed file data. With the window set up and the text view holding the data to display, there's nothing left to do but display the window with a call to `Show()`:

```
         if (text && (file.Read(text, length) >= B_OK))
            fTextView->SetText(text, length);
         free(text);
      }
      Show();
   }
```

Onward

This chapter's OpenSaveAsFile example is the most complete program in this book—it actually does something quite useful! Using the techniques presented in the preceding chapters, you should be able to turn OpenSaveAsFile into a real-world application. Begin by polishing up the File menu. Add a New item—that requires just a few lines of code. Also add a Quit item to provide a more graceful means of exiting the program. Chapter 8 covered text editing in detail—use that chapter's techniques to add a complete, functioning Edit menu. Those changes will result in a useful, functional text editor. If you want to develop a program that's more graphics-oriented, go ahead—Chapter 4, *Windows, Views, and Messages*, and Chapter 5, *Drawing*, hold the information to get you started. If you take that route, then you can always include the text-editing capabilities covered here and in Chapter 8 as a part of your program. For example, a graphics-oriented application could include a built-in text editor that allows the user to enter and save notes.

Regardless of the type of application you choose to develop, check out Be's web site at *http://www.be.com/*. In particular, you'll want to investigate their online developer area for tips, techniques, and sample code. For reference documentation, consider the Be Book, Be's own HTML-formatted documentation. For a more complete hardcopy version of that book, look into obtaining one or both of the O'Reilly & Associates books *Be Developer's Guide* and *Be Advanced Topics*.