

Resources

This chapter describes how your application can use the Resource Manager to create and manage resources, collections of data stored in a file's resource fork that have a defined structure or type. The Macintosh Operating System and the Macintosh Toolbox define a large number of resource types. You'll need to include resources of some of these types in your application's resource file to meet various requirements of the system software. In addition, the system software provides a number of resources (such as fonts, patterns, and icons) that you can use to help create the standard Macintosh user interface for your application.

This chapter begins with a general description of resources. Then it shows how to

- use predefined system resources
- create resources of a standard type
- define your own custom resources and resource types

For a complete description of the capabilities of the Resource Manager and for code samples illustrating more advanced resource-handling techniques, see the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox*.

About Resources

An experienced Macintosh programmer might cringe at several features of the GreetMe source code shown in Listing 1-1 on page 3. One of the main sins it commits is this line:

```
gString := 'Hello, world!';
```

The problem with this line is that it includes, as part of the source code of the application, the message string that is to be displayed in the output window. While such an intermixing of code and data might be standard in some programming environments, it's definitely nonstandard in the Macintosh environment. To change the message, or to produce a version of the message in a different language, you'd need to change the source code and recompile the application. It would be better to isolate the changing data (the message string) from the application's code.

When you're programming on the Macintosh, you can do this by creating a resource that contains the message string. A **resource** is any collection of data having a defined structure that is stored in a file designed to hold resources, known as a **resource file**. Then you can read the message string from the resource file using a call like this:

```
GetIndString(gString, kMessages, kGreetingString);
```

Resources

The `GetIndString` procedure reads the resource of type 'STR#' that has the resource ID `kMessages` in an open resource fork. This type of resource contains a string list, which is a sequential list of Pascal strings. Then `GetIndString` selects the string having the index `kGreetingString`. If there are at least that many strings in the string list, it puts the appropriate string into the first parameter (in this case, `gString`).

Note

The `GetIndString` procedure is not part of the Resource Manager, but it does call the Resource Manager. Many Toolbox and Operating System routines internally call the Resource Manager to retrieve information from resources. ♦

The resources used by an application can be created and changed separately from the application's code. This separation is the main advantage to having resource files. A change in a simple greeting or in the title of a menu, for example, won't require any recompilation of code, nor will translation to another language.

IMPORTANT

Properly written Macintosh applications should store *all* language- or location-sensitive data as resources, so that localization is largely a matter of editing the application's resources. ▲

Resource Paths

At any given time during your application's execution, there are usually two or more open resource files from which you can read information. The system resource file is opened by the Operating System at startup time. It contains standard resources, called *system resources*, shared by all applications. Among these are icons, fonts, sounds, and other collections of data. The system resource file also contains a number of code resources that you call indirectly to help create the standard Macintosh user interface. For example, the standard appearance and behavior of pull-down menus is governed by a menu-definition procedure, stored as a resource of type 'MDEF' in the system resource file. The system resource file also contains code resources that help you create standard windows and controls.

Your application's resource file is opened when your application is launched. You can call the `CurResFile` function early in your application's execution to get the reference number of your application's resource file.

```
gAppsResourceFile := CurResFile;
```

Resources

You need to keep track of your application’s resource file because the Resource Manager always looks for resources in the current resource file, which can change. Each time you open a resource file, it becomes the current resource file. You’re likely to open a number of different resource files at various points in your application’s execution. For instance, many applications store the user’s general preferences in a resource file in the *Preferences folder* in the System Folder. In addition, if your application supports document files, you’ll probably store some of the document’s settings in the document’s resource file. Table 3-1 summarizes the typical locations of resources used by an application.

Table 3-1 Typical locations of resources

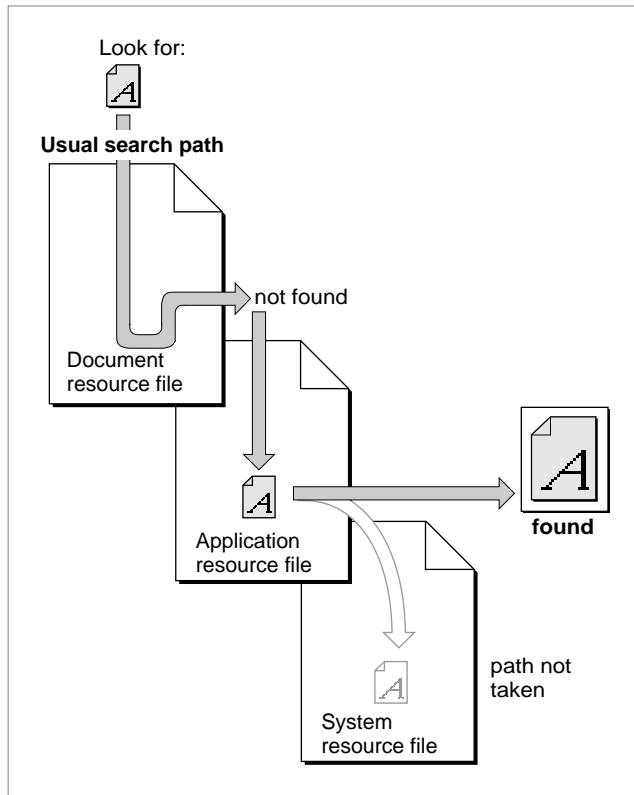
Resource file	Resources contained in file
System resource file	Standard elements of the Macintosh user interface (such as fonts, sounds, and icons) shared by all applications, and code resources that manage user interface elements (such as menus, controls, and windows)
Application resource file	Resources containing static data (such as menu titles, menu items, and text strings) used by the application
Application preferences file	Resources encoding the user’s global preferences for the application
Document resource file	Resources used only in this document, or resources that govern the appearance of the document’s window (such as its location on the screen)

When searching resource files, the Resource Manager generally begins with the most recently opened one. When you ask it to open a resource of a particular type and ID, it first looks in the current resource file. If the Resource Manager doesn’t find the specified resource there, it then looks in the resource file opened just before the current resource file. As long as the resource remains unfound, the Resource Manager continues until it reaches the last resource file in the chain, which is probably the system resource file. If the specified resource isn’t there either, the Resource Manager gives up and notifies your application that the resource can’t be found.

Resources

Figure 3-1 illustrates a typical search path followed by the Resource Manager as it looks for a particular font.

Figure 3-1 Searching for a resource

**Note**

Unlike the system resource file and your application's resource file, a document's resource file is not automatically opened when you open the document's data fork. If you want to include a document's resource fork in the chain of open resource files, you need to open it explicitly (for instance, using the `HOpenResFile` routine). ♦

In general it's best not to rely too much on the Resource Manager's ability to search through open resource files; instead, you should explicitly set the appropriate resource file as the current resource file (by calling `SetResFile`) before you read or write any resource data. In addition, you can restrict the Resource Manager's search for a resource to the current resource file by using special Resource Manager routines. For example, instead of calling `GetResource`, you can call `Get1Resource`. This instructs the Resource Manager to look only in the first resource file in the chain of open resource files.

Resource Types

As indicated above, resources are grouped logically by function into *resource types*. You refer to a resource by passing the Resource Manager a *resource specification*, which consists of the resource type and an ID number or a name. Any resource type is valid, whether one of those recognized by the Toolbox as referring to a standard Macintosh resource (such as a pattern), or a custom type created for use by your application.

Note

The Resource Manager knows nothing about the formats of the individual types of resources. Only the routines in the other parts of the Toolbox and Operating System that call the Resource Manager have this knowledge. ♦

A resource type can be any sequence of four alphanumeric characters, including the space character. You can create resource types for your application, provided that they consist of all uppercase letters and do not conflict with the standard resource types already created. A resource type is defined by the ResType data type:

```
TYPE ResType = PACKED ARRAY[1..4] OF CHAR;
```

IMPORTANT

Uppercase letters are distinguished from their lowercase counterparts in resource types. In addition, Apple reserves for its own use all resource types that include any lowercase letters. If you create custom resource types for use by your application, make sure that the type includes all uppercase letters. ▲

Table 3-2 lists the names and uses of some of the standard resource types used by the Macintosh system software. Uppercase resources are listed first.

Table 3-2 Some standard resource types

Resource type	Meaning
'ALRT'	Alert box template
'CODE'	Application code segment
'CURS'	Cursor
'DITL'	Item list in a dialog or alert box
'DLOG'	Dialog box template
'FONT'	Bitmapped font
'ICON'	Icon
'MBAR'	Menu bar
'MENU'	Menu
'PAT '	Pattern (The space in the resource type is required.)

continued

Resources

Table 3-2 Some standard resource types (continued)

Resource type	Meaning
'PICT'	QuickDraw picture
'SIZE'	Size of an application's partition and other information
'STR '	String (The space in the resource type is required.)
'STR#'	String list
'WIND'	Window template
'hdlg'	Help for dialog box or alert box items
'sfnt'	Outline font
'snd '	Sound (The space in the resource type is required.)

You pick out a particular resource by specifying its type together with a *resource name* or a *resource ID* number. In general, it's best to use resource IDs because they're guaranteed to be unique within any given resource file. By contrast, it's possible to have two different resources of the same type with the same name.

Resource Structure

A resource file consists of a number of individual resources together with a *resource map*, an indication of where in the resource file the data for a given resource is to be found. You usually don't need to know about the structure—or even the existence—of the resource map. The Resource Manager uses it to keep track of a resource file's resources. If you lengthen or shorten a resource, or remove one from the resource file entirely, the Resource Manager takes care of modifying the resource map accordingly.

Often, you don't even need to know about the structure of the individual resources you access in a resource fork. Sometimes you just need to open a resource and pass the handle you receive from the Resource Manager to some Toolbox routine. Here's an example:

```
FOR count := 1 TO 4 DO
    gEmptyPats[count] := GetPattern(kEmptyID + (count - 1));

FillRgn(myRegion, gEmptyPats[gEmptyIndex]^);
```

Resources

At application startup time, the Venn Diagrammer application reads the four available emptiness patterns from the application's resource file. Later, when it is drawing the current contents of the Venn diagram, it might fill a specified region with the current pattern. The application itself knows nothing about the actual structure of a pattern.

Sometimes, however, you do need to know about the structure of the individual resources you want to use in your application. This is certainly true for any resources your application defines itself. Occasionally, you also need to know how the data in a system resource is structured. *Inside Macintosh* uses two general methods for displaying the structure of a resource's data: resource descriptions and resource diagrams.

The first method used in *Inside Macintosh* to describe the structure of a resource involves specifying a description in the Rez resource description language. Listing 3-1 shows the Rez input for a sample dialog box.

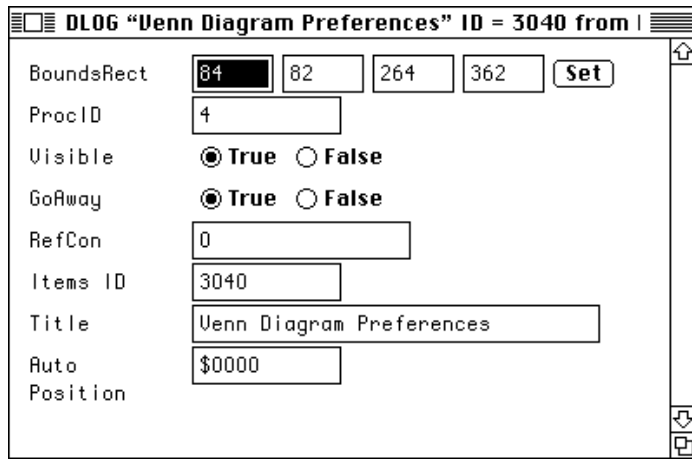
Listing 3-1 Rez input for the Preferences dialog box

```
resource 'DLOG' (rVennDPrefsDial, purgeable) {           /*dialog resource*/
    {84, 82, 264, 362},           /*rectangle for dialog box*/
    noGrowDocProc,               /*window definition ID for modeless dialog*/
    visible,                     /*display this dialog box immediately*/
    goAway,                      /*draw a close box*/
    0x0,                         /*initial refCon value of zero*/
    rVennDPrefsDial,             /*use item list with res ID rVennDPrefsDial*/
    "Venn Diagram Preferences", /*window title*/
    noAutoCenter                 /*don't automatically center the window*/
};
```

Rez is a resource compiler: it takes a resource description like the one shown in Listing 3-1 and produces a compiled resource. As you can see, the Rez description includes information about the desired dialog box, including the box's rectangle, window definition ID, and initial window title.

Rez is provided as part of the Macintosh Programmer's Workshop (MPW) and as part of some third-party development environments. If you prefer, you can create and edit resources using tools like ResEdit, a graphic resource editor provided by Apple Computer, Inc. Using ResEdit, you'll create and modify resources in a slightly more friendly atmosphere, by manipulating windows like the one shown in Figure 3-2.

Resources

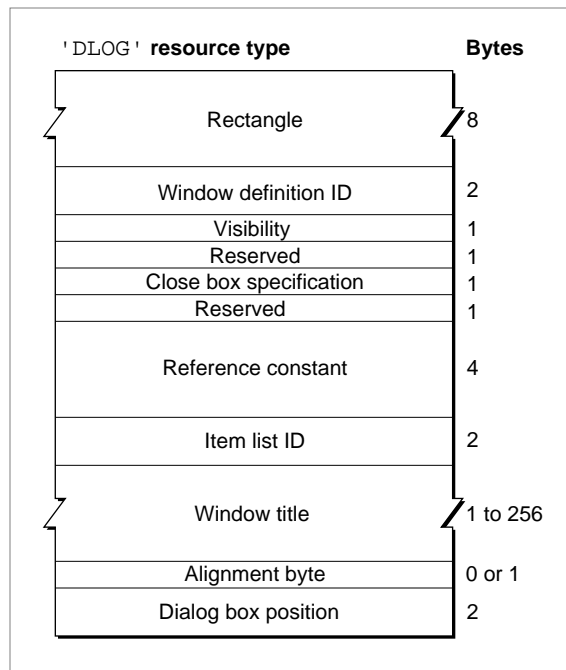
Figure 3-2 The ResEdit version of the Preferences dialog box

ResEdit uses an internal resource compiler to turn this graphic representation of a resource into a compiled resource.

Note

For most purposes, and especially for programmers new to the Macintosh environment, ResEdit is a perfectly adequate tool for creating and editing resources. For information about using ResEdit to create resources, see *ResEdit Reference*. For complete information about using Rez to compile resource descriptions into resources, see *Macintosh Programmer's Workshop Reference*. ♦

Whether you use Rez or ResEdit's internal resource compiler to create resources, the compiled resource will have the same structure. This structure is sometimes depicted in *Inside Macintosh* using a resource diagram, as illustrated in Figure 3-3.

Figure 3-3 A resource diagram

Using Standard Resources

In general, you'll need to create resources describing the standard user interface elements used by your application, including

- dialog boxes
- dialog box item lists
- menus
- windows
- controls

Resources

For standard user interface elements, the Macintosh Toolbox provides special routines you can use to open the appropriate resources. For instance, you can call the Dialog Manager function `GetNewDialog` to read a dialog box resource (of type 'DLOG') and the corresponding item list (of type 'DITL') from your application's resource fork.

```
myDialog := GetNewDialog(myKind, myPointer, WindowPtr(-1));
```

Similarly, you can call the Window Manager routine `GetNewWindow` to open a window description resource (of type 'WIND'). Internally, these routines call Resource Manager routines such as `GetResource` to read the resource data from the resource file.

Some Toolbox routines are simply loosely disguised Resource Manager calls. For example, the code shown on page 56 which uses `GetPattern` to open four available emptiness patterns could be replaced by this functionally equivalent code:

```
FOR count := 1 TO 4 DO
    gEmptyPats[count] := GetResource('PAT ', kEmptyID + (count - 1));
```

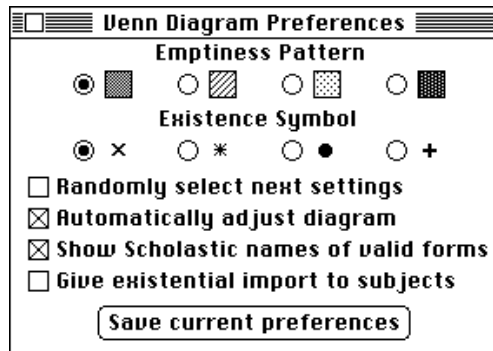
Most Resource Manager routines that open resources return a handle to the specified resource data. You can pass that handle to other Resource Manager routines, or doubly dereference it to get at the resource data.

Using Custom Resources

In addition to using system resources to help create the standard Macintosh user interface for your application and standard resource types to help isolate its localizable data, you'll probably also want to create custom resources. This section illustrates how to define a custom resource type and how to create and manage resources of that type. The source code provided here shows how to handle a preferences file. This file stores the user's global preferences, and your application can retrieve them each time it is launched. When it starts up, the Venn Diagrammer application tries to open a preferences file, which contains a single resource with the following type and ID:

```
CONST
    kPrefResType      = 'PRFN';    {type of preferences resource}
    kPrefResID        = 259;       {ID of preferences resource}
```

As you've seen earlier in this book, the preferences file needs to contain information about the user's Venn diagram preferences, as displayed in the Preferences dialog box shown in Figure 3-4.

Figure 3-4 The Preferences dialog box

Here, there are six pieces of information that need to be tracked. To maintain this information, the Venn Diagrammer application defines a data structure of type `MyPrefsRec` (defined in Listing 3-2).

Listing 3-2 The structure of a resource containing Venn diagram preferences

TYPE

```
MyPrefsRec = RECORD
    autoDiag:   Boolean;    {do we automatically fix the diagram?}
    showName:   Boolean;    {do we show names of valid arguments?}
    isImport:   Boolean;    {do subjects have existential import?}
    isRandom:   Boolean;    {do we select next setting randomly?}
    emptyInd:   Integer;    {index of the desired emptiness pattern}
    existInd:   Integer;    {index of the desired existence symbol}
END;
MyPrefsPtr = ^MyPrefsRec;
MyPrefsHnd = ^MyPrefsPtr;
```

When it is first launched, the Venn Diagrammer application calls the application-defined routine `DoReadPrefs` (defined in Listing 3-3) to read the user's existing preferences settings. First, `DoReadPrefs` determines the name of the preferences file by reading a resource in the application's resource file that contains that name. By convention, the name of the preferences file consists of the name of the application followed by the string " Preferences", for instance, Venn Diagrammer Preferences.

Resources

Listing 3-3 Reading a user's preferences

```

PROCEDURE DoReadPrefs;
  VAR
    myVRefNum: Integer;
    myDirID: LongInt;
    myName: Str255;      {name of this application}
    myPrefs: Handle;     {handle to actual preferences data}
    myResNum: Integer;   {reference number of opened resource file}
    myResult: OSErr;
  CONST
    kNameID = 4000;      {resource ID of 'STR#' with filename}
BEGIN
  {Determine the name of the preferences file.}
  GetIndString(myName, kNameID, 1);

  {Figure out where the preferences file is.}
  IF IsFindFolder THEN
    myResult := FindFolder(kOnSystemDisk, kPreferencesFolderType,
                          kDontCreateFolder, myVRefNum, myDirID)
  ELSE
    myResult := -1;

  IF myResult <> noErr THEN
    BEGIN
      myVRefNum := 0;      {use default volume}
      myDirID := 0;       {use default directory}
    END;

    {Open the preferences resource file.}
    myResNum := HOpenResFile(myVRefNum, myDirID, myName, fsCurPerm);

    {If no preferences file successfully opened, create one }
    { by copying default preferences in app's resource file.}
    IF myResNum = -1 THEN
      myResNum := DoCreatePrefsFile(myVRefNum, myDirID, myName);

  IF myResNum <> -1 THEN      {if we successfully opened the file...}
    BEGIN
      UseResFile(myResNum); {make the new resource file current one}
      myPrefs := Get1Resource(kPrefResType, kPrefResID);
      IF myPrefs = NIL THEN
        exit(DoReadPrefs);
      WITH MyPrefsHnd(myPrefs)^^ DO

```

Resources

```

BEGIN                                {read the preferences settings}
    gAutoAdjust := autoDiag;
    gShowNames := showName;
    gGiveImport := isImport;
    gStepRandom := isRandom;
    gEmptyIndex := emptyInd;
    gExistIndex := existInd;
END;

{Make sure some preferences globals make sense.}
IF NOT (gExistIndex IN [1..4]) THEN
    gExistIndex := 1;
IF NOT (gEmptyIndex IN [1..4]) THEN
    gEmptyIndex := 1;

{Reinstate the application's resource file.}
UseResFile(gAppsResourceFile);
END;

gPreferencesFile := myResNum;          {remember its resource ID}
END;

```

After determining the name of the preferences file, `DoReadPrefs` calls the application-defined utility `IsFindFolder` to see whether the operating environment supports the `FindFolder` function. (See Listing 9-6 on page 179 for a definition of `IsFindFolder`.) If it does, `DoReadPrefs` calls `FindFolder` to find the location of the Preferences folder. The `FindFolder` function returns the volume reference number and the directory ID of that folder, if it can be found. If `FindFolder` isn't available or if it cannot find the Preferences folder, `DoReadPrefs` looks in the default directory on the default volume.

IMPORTANT

Just looking in the default directory when you cannot find the Preferences folder isn't really the best thing to do. Your application would probably want to look in the System Folder to see if your preferences file is there. ▲

Once the target folder is successfully located, `DoReadPrefs` calls the `HOpenResFile` function to try to open a file having the required name in that folder. If no such file can be opened (as indicated by a returned reference number of -1), `DoReadPrefs` calls the application-defined function `DoCreatePrefsFile` to attempt to create a new preferences file. (See Listing 3-4 for a definition of `DoCreatePrefsFile`.)

If the existing or newly created preferences file is successfully opened, then `DoReadPrefs` calls `UseResFile` to make that file the current resource file. Then it reads the resource of type `kPrefResType` and ID `kPrefResID` from that file. If all goes

Resources

well, `DoReadPrefs` reads the current settings from that resource and assigns them to the appropriate global variables:

```
WITH MyPrefsHnd(myPrefs)^^ DO
    BEGIN                {read the preferences settings}
        gAutoAdjust := autoDiag;
        gShowNames := showName;
        gGiveImport := isImport;
        gStepRandom := isRandom;
        gEmptyIndex := emptyInd;
        gExistIndex := existInd;
    END;
```

Finally, `DoReadPrefs` ensures that the values of the two index variables are within acceptable limits and then restores the application's resource file as the current resource file by calling `UseResFile` once again. Notice that the preferences resource file is left open; this way, the Venn Diagrammer application need not reopen the file if the user wants to change the stored preferences settings.

The `DoCreatePrefsFile` function that is called by `DoReadPrefs` is defined in Listing 3-4. Essentially, `DoCreatePrefsFile` creates a resource file in the appropriate location and with the appropriate name; then it copies into that new resource file an existing set of preferences (stored in the application's resource fork).

Listing 3-4 Creating a preferences file

```
FUNCTION DoCreatePrefsFile (myVRefNum: Integer; myDirID: LongInt;
                           myName: Str255): Integer;

VAR
    myResNum:   Integer;
    myResult:   OSErr;
    myID:       Integer;    {resource ID of resource in app's res fork}
    myHandle:   Handle;     {handle to resource in app's res fork}
    myType:     ResType;    {ignored; used for GetResInfo}
BEGIN
    myResult := noErr;
    HCreateResFile(myVRefNum, myDirID, myName);
    IF ResError = noErr THEN
        BEGIN
            myResNum := HOpenResFile(myVRefNum, myDirID, myName, fsCurPerm);
            IF myResNum <> -1 THEN
                BEGIN
                    UseResFile(gAppsResourceFile);
                    myHandle := Get1Resource(kPrefResType, kPrefResID);
                    IF ResError = noErr THEN
```

Resources

```

        BEGIN
            GetResInfo(myHandle, myID, myType, myName);
            myResult := DoCopyResource(kPrefResType, myID,
                                      gAppsResourceFile, myResNum);
        END
    ELSE
        BEGIN
            CloseResFile(myResNum);
            myResult := HDelete(myVRefNum, myDirID, myName);
            myResNum := -1;
        END;
    END;

    DoCreatePrefsFile := myResNum;
END;
END;

```

To copy the existing resource from the application's resource file to the new preferences resource file, `DoCreatePrefsFile` calls the application-defined routine `DoCopyResource`. A version of `DoCopyResource` is shown in Listing 3-5.

Listing 3-5 Copying a resource from one resource file to another

```

FUNCTION DoCopyResource (rType: ResType; rID: Integer; source: Integer;
                        dest: Integer): OSErr;

VAR
    myHandle:   Handle;           {handle to resource to copy}
    myName:     Str255;           {name of resource to copy}
    myAttr:     Integer;          {resource attributes}
    myType:     ResType;          {ignored; used for GetResInfo}
    myID:       Integer;          {ignored; used for GetResInfo}
    myResult:   OSErr;
    myCurrent:  Integer;          {current resource file on entry}
BEGIN
    myCurrent := CurResFile;      {remember current resource file}
    UseResFile(source);          {set the source resource file}
    myHandle := Get1Resource(rType, rID); {open the source resource}
    IF myHandle <> NIL THEN
        BEGIN
            GetResInfo(myHandle, myID, myType, myName); {get res name}
            myAttr := GetResAttrs(myHandle);             {get res attributes}
            DetachResource(myHandle);                    {so we can copy the resource}
            UseResFile(dest);                             {set destination resource file}
        END
    END
END;

```

Resources

```

IF ResError = noErr THEN
    AddResource(myHandle, rType, rID, myName);
IF ResError = noErr THEN
    SetResAttrs(myHandle, myAttr); {set res attributes of copy}
IF ResError = noErr THEN
    ChangedResource(myHandle);    {mark resource as changed}
IF ResError = noErr THEN
    WriteResource(myHandle);      {write resource data}
END;

DoCopyResource := ResError;      {return result code}
ReleaseResource(myHandle);      {get rid of resource data}
UseResFile(myCurrent);          {restore original resource file}
END;

```

As you can see, `DoCopyResource` opens the resource to be copied. It copies that resource into the destination resource file by making the destination file the current resource file and then calling the Resource Manager routine `AddResource`. However, before calling `AddResource`, you need to disassociate the source resource from its resource file. Because `AddResource` requires a handle to some data in memory that is not a handle to an existing resource, you need to call the `DetachResource` procedure to cut the link between the resource data and its original resource file.

You can determine whether a Resource Manager call succeeded by calling the function `ResError`, which returns the result code from the most recently executed Resource Manager routine. The `DoCopyResource` function calls `ResError` repeatedly to make sure that the resource data was successfully added, that the resource attributes were successfully copied, that the destination resource was successfully marked as changed, and that the data was successfully written out to disk.

It's easy to see how to save a set of preferences to the user's preferences file. In essence, you simply need to reverse the strategy employed in reading the preferences. Listing 3-6 defines the `DoSavePrefs` procedure, which the Venn Diagrammer application calls whenever the user wants to save the current preferences settings. The `DoSavePrefs` procedure assumes that the application's preferences file is already open.

Listing 3-6 Saving current preferences settings

```

PROCEDURE DoSavePrefs;
VAR
    myPrefData: Handle;    {handle to new resource data}
    myHandle:   Handle;    {handle to resource to replace}
    myName:     Str255;    {name of resource to copy}
    myAttr:     Integer;   {resource attributes}
    myType:     ResType;   {ignored; used for GetResInfo}
    myID:       Integer;   {ignored; used for GetResInfo}

```

Resources

```

BEGIN
    {Make sure we have an open preferences file.}
    IF gPreferencesFile = -1 THEN
        exit(DoSavePrefs);

    myPrefData := NewHandleClear(sizeof(MyPrefsRec));
    HLock(myPrefData);
    WITH MyPrefsHnd(myPrefData)^ DO
        BEGIN
            autoDiag := gAutoAdjust;
            showName := gShowNames;
            isImport := gGiveImport;
            isRandom := gStepRandom;
            emptyInd := gEmptyIndex;
            existInd := gExistIndex;
        END;

    UseResFile(gPreferencesFile);                {use preferences file}
    myHandle := Get1Resource(kPrefResType, kPrefResID);
    IF myHandle <> NIL THEN
        BEGIN
            GetResInfo(myHandle, myID, myType, myName); {get res name}
            myAttr := GetResAttrs(myHandle);           {get res attributes}
            RmveResource(myHandle);
            IF ResError = noErr THEN
                AddResource(myPrefData, kPrefResType, kPrefResID, myName);
            IF ResError = noErr THEN
                WriteResource(myPrefData);
        END;

    HUnlock(myPrefData);
    ReleaseResource(myPrefData);
    UseResFile(gAppsResourceFile);                {restore app's resource file}
END;

```

The `DoSavePrefs` procedure creates a new preferences record and fills in the fields as appropriate. Then it removes the existing preferences resource from the preferences file and adds a new resource. To make sure that the new resource data is written out to disk, `DoSavePrefs` calls the `WriteResource` procedure. Finally, `DoSavePrefs` restores the application's resource file as the current resource file.

