This chapter provides an overview of the tasks involved in making your application scriptable and recordable. This chapter also introduces some of the ways your application can use the Component Manager and scripting components to manipulate and execute scripts. The three chapters that follow provide detailed information, including sample code, about the topics introduced in this chapter.

The chapter "Introduction to Interapplication Communication" in this book describes the Open Scripting Architecture (OSA) and its relationship to the Apple Event Manager and other parts of the IAC architecture. If your application supports the appropriate core and functional-area events defined in the *Apple Event Registry: Standard Suites,* you can make it scriptable (that is, capable of responding to Apple events sent by scripting components) by providing an Apple event terminology extension (`'aete'`) resource. This chapter describes some of the tasks involved in making your application scriptable and introduces the `'aete'` resource. The next chapter, "Apple Event Terminology Resources," describes in detail how to create an `'aete'` resource.

This chapter also introduces Apple event recording and the use of the standard scripting component routines to manipulate and execute scripts. The chapter "Recording Apple Events" describes in detail how to make your application recordable, and the chapter "Scripting Components" describes how to use the standard scripting component routines.

To use this chapter or any of the chapters that follow, you should be familiar with the chapters "Introduction to Apple Events" and "Responding to Apple Events" in this book. If you plan to make your application recordable, you should also read the chapters "Creating and Sending Apple Events" and "Resolving and Creating Object Specifier Records."

The *AppleScript Software Developers' Kit* (available from APDA) provides development tools, sample applications, and information about the AppleScript language that you will find useful when you begin to apply the information in this chapter to your application.

If you are developing a scripting component, you should provide support for the standard scripting component routines described in the chapter "Scripting Components," and you should read the instructions for creating components in the chapter "Component Manager" in *Inside Macintosh: More Macintosh Toolbox.*

This chapter begins with an overview of scripts and scripting components. The rest of the chapter describes how the OSA makes it possible to

- make your application scriptable
- make your application recordable
- have your application manipulate and execute scripts

About Scripts and Scripting Components

A ***script*** is any collection of data that, when executed by the appropriate program, causes a corresponding action or series of actions. The Open Scripting Architecture (OSA) provides a standard mechanism that allows users to control multiple applications with scripts written in a variety of scripting languages. Each scripting language has a corresponding ***scripting component.*** Each scripting component supports the standard scripting component routines described in the chapter "Scripting Components" in this book.

When a scripting component executes a script, it performs the actions described in the script, including sending Apple events to applications if necessary. Like other components that use the Component Manager, scripting components can provide their own routines in addition to the standard routines that must be supported by all components of the same type.

Scripting components typically implement a text-based scripting language based on Apple events. For example, the ***AppleScript component*** implements ***AppleScript,*** the standard user scripting language defined by Apple Computer, Inc. This book uses AppleScript examples to demonstrate how applications can interact with scripting components.

Other scripting components may support the standard scripting component routines in different ways. Scripting components need not implement a text-based scripting language, or even one that is based on Apple events. For example, specialized scripting components can play sounds, execute XCMDs, or perform almost any other action when they execute scripts.

This chapter describes three ways that you can take advantage of the OSA:

- You can make your application ***scriptable,*** or capable of responding to Apple events sent to it by a scripting component. An application is scriptable if it
  - □ Responds to the appropriate standard Apple events as described in the chapter "Responding to Apple Events" in this book.
  - □ Provides an Apple event terminology extension (`'aete'`) resource describing the Apple events that your application supports and the user terminology that corresponds to those events. The `'aete'` resource allows scripting components to interpret scripts correctly and send the appropriate Apple events to your application during script execution.

- You can make your application ***recordable***— that is, capable of sending Apple events to itself to report user actions to the Apple Event Manager for recording purposes. After a user has turned on recording for a particular scripting component, the scripting component receives a copy of every subsequent Apple event that any application on the local computer sends to itself. The scripting component records such events in the form of a script.
- You can have your application manipulate and execute scripts with the aid of a scripting component. To do so, your application must
    - □ Use the Component Manager to open a connection with the appropriate component.
    - □ Use the standard scripting component routines described in the chapter "Scripting Components" to record, edit, compile, save, load, or execute scripts.

Users of scriptable applications can execute scripts to perform tasks that might otherwise be difficult to accomplish, especially repetitive or conditional tasks that involve multiple applications. For example, a user can execute an AppleScript script to locate database records with specific characteristics, create a series of graphs based on those records, import the graphs into a page-layout document, and send the document to a remote computer on the network via electronic mail. When a user executes such a script, the AppleScript component attempts to perform the actions the script describes, including sending Apple events when necessary.

To respond appropriately to the Apple events sent to it by the AppleScript component, the database application in this example must be able to locate records with specific characteristics so that it can identify and return the requested data. These characteristics are described by an object specifier record that is part of an Apple event supported by the application. Also, the other applications involved must support Apple events that manipulate the data in the ways described in the script. Each application in this example must also provide an `'aete'` resource describing the Apple events that the application supports and the user terminology that corresponds to those events, so that the AppleScript component can interpret the script correctly.

Even with little or no knowledge of a particular scripting language, users of applications that are recordable as well as scriptable can record simple scripts. More knowledgeable users may also wish to record their actions as scripts with recordable applications and then edit or combine scripts as needed.

An application that uses scripting components to manipulate and execute scripts need not be scriptable; however, if it is scriptable, it can execute scripts that control its own behavior. In other words, it can perform tasks by means of scripts and allow users to modify those scripts to suit their own needs.

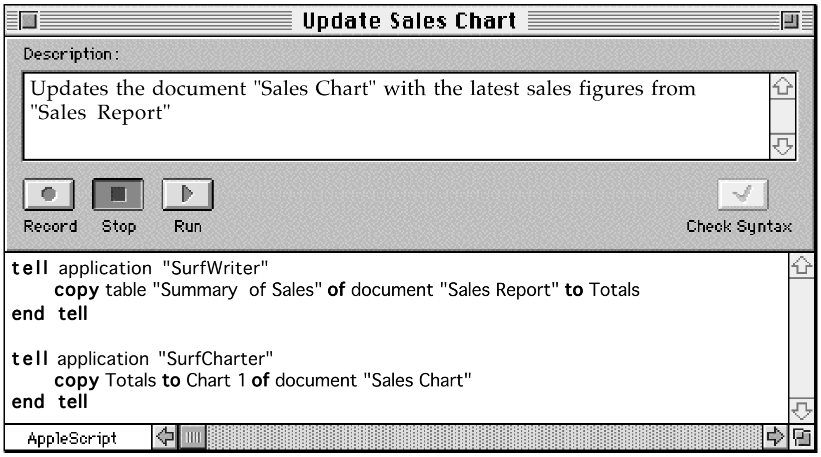
The next three sections provide an overview of the way scripting components can interact with applications.

Script Editors and Script Files

A ***script editor*** is an application that allows users to record, edit, save, and execute scripts. For example, the AppleScript component uses the services of the Script Editor application.

Figure 7-1 shows an AppleScript script displayed in a Script Editor window. The Record, Stop, and Run buttons control a script in much the same way that the equivalent buttons on a cassette recorder control an audio tape. A ***script comment*** at the top of the window describes what the script does. Users with some knowledge of a text-based scripting language such as AppleScript can use Script Editor to modify recorded scripts or write their own scripts.

Figure 7-1 A script window in the Script Editor application



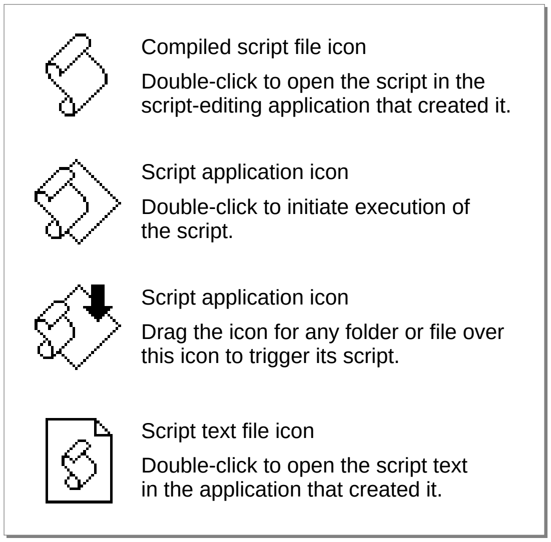
Updates the document "Sales Chart" with the latest sales figures from "Sales Report"

```
tell application "SurfWriter"
      copy table "Summary of Sales" of document "Sales Report" to Totals
end tell

tell application "SurfCharter"
      copy Totals to Chart 1 of document "Sales Chart"
end tell
```

Script Editor provides entry-level scripting capabilities, but it is not intended for intensive script development. Users who wish to write complex scripts may replace Script Editor with more sophisticated editors that provide specialized debugging and development tools.

A script like the one in Figure 7-1 can be stored in a ***script file*** represented by an icon in the Finder, or it can be stored within an application or one of its documents. Figure 7-2 shows the four icons representing the files in which Script Editor stores scripts.

Figure 7-2 Script file icons in the Finder and corresponding user actions



Compiled script file icon

Double-click to open the script in the script-editing application that created it.

Script application icon

Double-click to initiate execution of the script.

Script application icon

Drag the icon for any folder or file over this icon to trigger its script.

Script text file icon

Double-click to open the script text in the application that created it.

Script Editor and similar script-editing applications allow users to store scripts using three file types:

- A ***compiled script file*** has the file type `'osas'` and contains the script data as a resource of type `'scpt'`. Before executing the script in a compiled script file, a user must first open the script from the Finder or from an application such as Script Editor. After opening a compiled script in an application that supports script editing, the user can view the script, modify it if necessary, and execute it.
- A ***script application*** has the file type `'APPL'` and contains the script data as a resource of type `'scpt'`. Its kind is "application." A script application takes one of two forms, each with its own icon:
  - A script application with the creator signature `'aplt'`. A user double-clicks the icon to trigger the script.
  - A script application with the creator signature `'dplt'`. A user can drag the icon for another file or a folder over this script application's icon to trigger a script that acts on that object.

By default, when a user triggers the script in either kind of script application, a splash screen appears that allows the user either to quit or to run the script. Users can also save a script application in a form that bypasses the splash screen, running the script immediately after the user double-clicks its icon.

- A ***script text file*** contains only a plain-text version of uncompiled scripting-language statements. This format is useful primarily as a last resort for saving a script that can't be compiled because of syntax errors or other problems. It is also useful for exchanging unstyled text with other text-based applications. A user must open a script text file in a script editor and successfully compile it before it will execute.

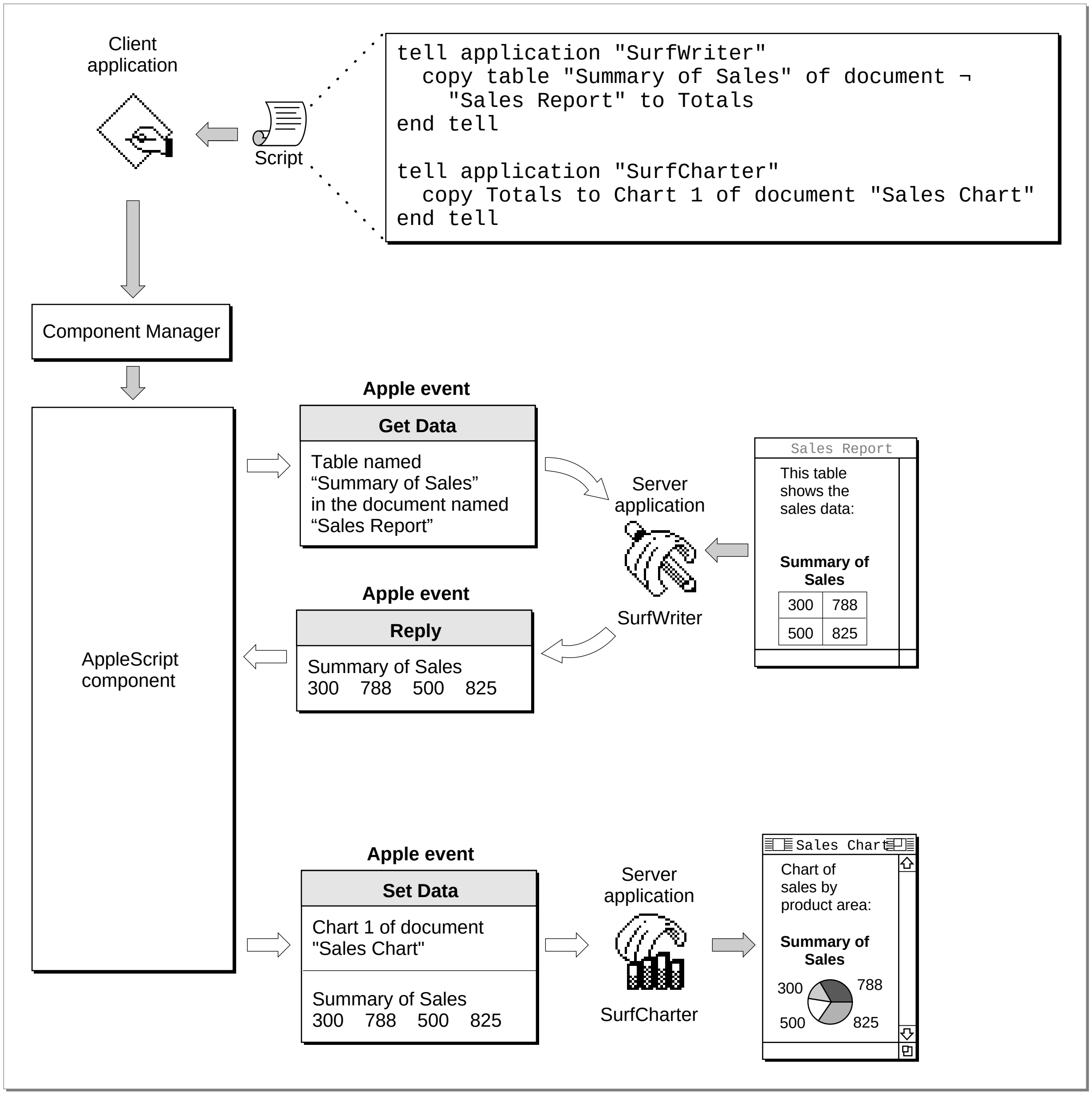
Like sound resources, scripts can be stored within applications and documents as well as in distinct files that can be manipulated from the Finder. Your application can use the standard scripting component routines to manipulate and execute both its own internally stored scripts and scripts stored as separate files whose icons appear in the Finder. For more information about script storage formats, see "Saving Script Data" on page 10-12.

The next two sections describe how scripting components interact with scriptable applications and with applications that execute scripts.

Scripting Components and Scriptable Applications

Scripting components control the behavior of scriptable applications by means of Apple events. For example, when the AppleScript component executes the AppleScript script shown in Figure 7-1, it sends the Apple events shown in Figure 7-3 to trigger the actions described by the script. The client application in this example would most commonly be a script editor but could also be any other application that uses standard scripting component routines to manipulate and execute scripts.

Figure 7-3 How the AppleScript component executes a script



```
tell application "SurfWriter"
  copy table "Summary of Sales" of document ¬
    "Sales Report" to Totals
end tell

tell application "SurfCharter"
  copy Totals to Chart 1 of document "Sales Chart"
end tell
```

As described in the chapter “Introduction to Apple Events” in this book, a client application is any application that uses Apple events to request a service or information. A client application that executes a script does not send the corresponding Apple events itself; instead, it uses scripting component routines to manipulate and execute the script. The scripting component sends Apple events when necessary to trigger the actions described in the script. Similarly, a scriptable application that responds to the Apple events sent by a scripting component can be considered the server application for those Apple events.

When a scripting component evaluates a script, it attempts to perform all the actions described in the script, including sending Apple events when necessary. In the example shown in Figure 7-3, the AppleScript component first performs the action described in the first `tell` statement:

```
tell application "SurfWriter"
   copy table "Summary of Sales" of document¬
      "Sales Report" to Totals
end tell
```

To perform this action, the AppleScript component sends a Get Data event to the SurfWriter application requesting the data from the specified table. The SurfWriter application returns the data to the AppleScript component in a standard reply Apple event, and the AppleScript component sets the value of the variable `Totals` to the data returned by SurfWriter.

Then the AppleScript component performs the action described in the second `tell` statement:

```
tell application "SurfCharter"
   copy Totals to Chart 1 of document "Sales Chart"
end tell
```

In this case, the AppleScript component sends a Set Data event to the SurfCharter application that sets the specified chart to the value of the variable `Totals`.

Both SurfWriter and SurfCharter are server applications for the Apple events sent by the AppleScript component, because they are performing services in response to requests made by the client application via the script.

To send the appropriate Apple events to a scriptable application while executing a script, a scripting component must obtain information about the nature of that application’s support for Apple events and the human-language terminology to associate with those events. A scriptable application provides this information in the form of an Apple event terminology extension (`'aete'`) resource. A scripting component uses both the `'aete'` resource provided by a scriptable application and the Apple event user terminology (`'aeut'`) resource provided by the scripting component itself to obtain the information it needs to execute a script that controls that application.

See “Making Your Application Scriptable,” which begins on page 7-14, for an overview of the tasks you should perform to make your application scriptable and a more detailed description of the `'aete'` and `'aeut'` resources. See “Making Your Application Recordable” on page 7-20 for an overview of the tasks you should perform if you want your application to be recordable as well as scriptable.

Scripting Components and Applications That Execute Scripts

To store and execute scripts as a client application, your application must first establish a connection with a scripting component registered with the Component Manager on the same computer. Each scripting component can manipulate and execute scripts written in the corresponding scripting language when your application calls the standard scripting component routines.

Your application can use scripting component routines to

- obtain a handle to a script in a form that can be saved, and load the script again when necessary
- allow users to modify scripts that have been previously saved
- compile and execute scripts
- redirect Apple events to script contexts
- supply application-defined functions for use by scripting components
- control the recording process directly, turning recording off and on and saving the recorded script for use by your application

Your application can perform these tasks as a client application regardless of whether it is scriptable or recordable. If your application is scriptable, however, it can execute scripts that control its own behavior, thus acting as both the client application and the server application for the corresponding Apple events. For example, your application can allow users to associate a script with a custom menu command that performs a series of routine actions on a selected object, sets preferences, or automates other actions within your application.

You can also use scripting component routines to execute scripts that perform tasks for your application with the aid of other applications. For example, a user of a word-processing application might be able to attach a script to a specific word so that the application executes the script whenever that word is double-clicked. Such a script could trigger Apple events that cause other applications to look up and display related information, run a QuickTime movie, perform a calculation, play a voice annotation, and so on.

Your application can associate a script with either Apple event objects or application-defined objects. Almost any user action can be used to trigger such a script: choosing a menu command, clicking a button, tabbing from one table cell to another, and so on. The script can be executed directly by the application when it detects a triggering action; or, if the script is associated with an Apple event object in the form of a script context, it can be executed automatically when a specified Apple event performs an action on that object.

The rest of this section describes one way that an application could execute such a script. Suppose a forms application allows users to create custom forms that can include scripts associated with specific fields on the form. These scripts are executed when the user presses Enter or Tab in the appropriate field. For the purposes of this example, it doesn't matter whether a field with which a script is associated is an Apple event object (which can be described in an object specifier record) or some other application-defined object (which can't be described in an object specifier record).

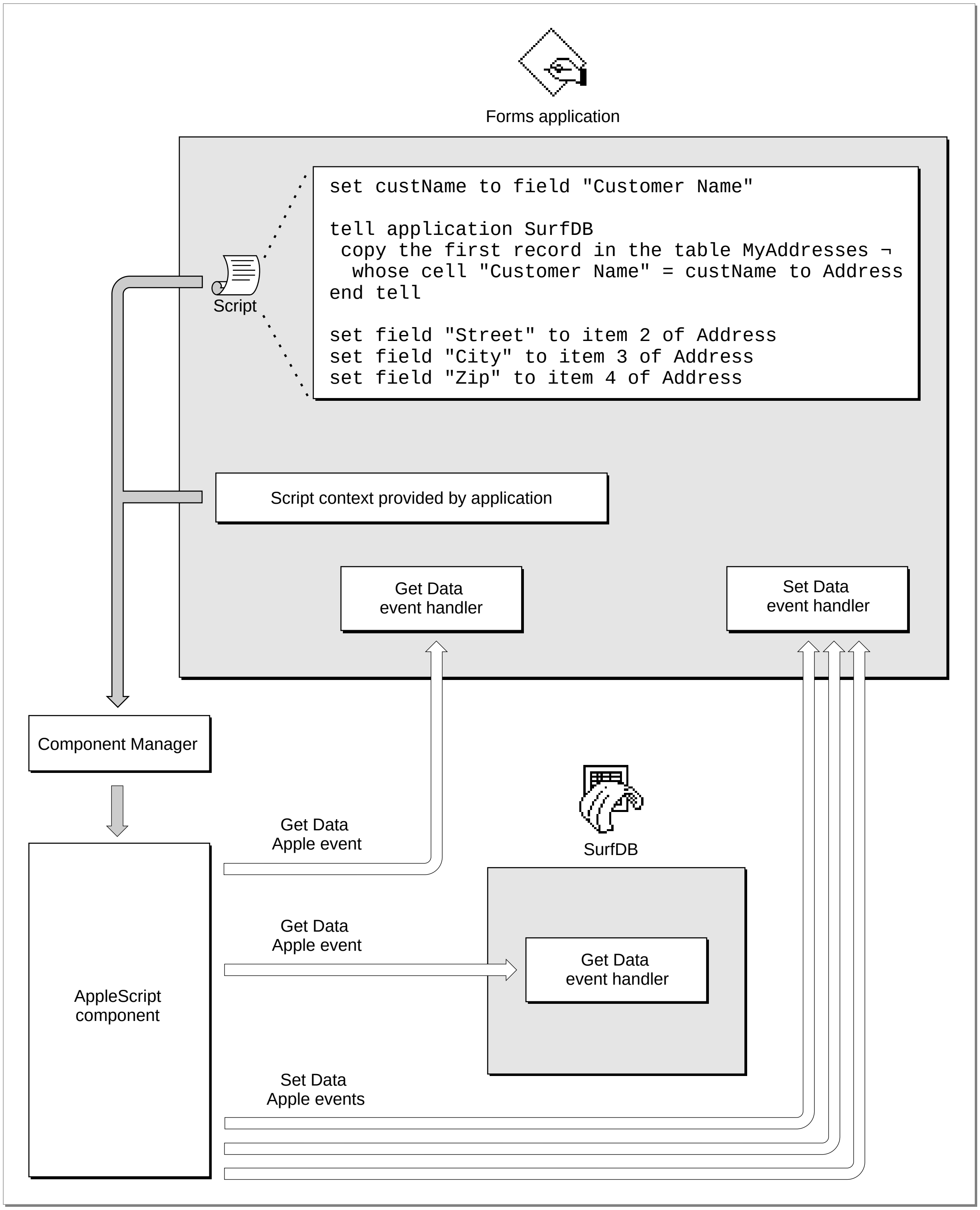
A company could use the forms application to create a custom order form for taking telephone orders. If the customer has ordered from the company before, the user can quickly retrieve the customer's address from the company database by typing the customer's name in a field and pressing the Tab key. In response, the application executes the script associated with the field. The script might look like this in AppleScript:

```
set custName to field "Customer Name"

tell application SurfDB
   copy the first record in the table MyAddresses ¬
      whose cell "Customer Name" = custName to Address
end tell

set field "Street" to item 2 of Address
set field "City" to item 3 of Address
set field "Zip" to item 4 of Address
```

To execute such a script (or to manipulate it any other way, such as when the form is first created), the forms application must previously have established a connection with the appropriate scripting component—in this case, the AppleScript component. When the user enters a customer name and presses Tab, the forms application calls scripting component routines to execute the script. As shown in Figure 7-4, the AppleScript component first sends the forms application a Get Data event that requests the contents of the "Customer Name" field and sets the variable `custName` to that value. It then sends SurfDB a Get Data event that requests the appropriate address information and copies it to the variable `Address`. (The replies to the Get Data events are not shown in Figure 7-4.) Finally, the AppleScript component sends the forms application a Set Data event that copies the address information from the variable `Address` to the appropriate fields.

The AppleScript component needs to maintain the binding of the variables `custName` and `Address` throughout execution of the script. Scripting components bind variables with the aid of a ***script context,*** which is a script that maintains context information for the execution of other scripts. An application specifies a script context when it executes a script. The forms application in Figure 7-4 provides a context for the scripting component to use whenever it executes a script associated with a button.

Figure 7-4 How an application uses the AppleScript component to execute a script



```
set custName to field "Customer Name"

tell application SurfDB
 copy the first record in the table MyAddresses ¬
  whose cell "Customer Name" = custName to Address
end tell

set field "Street" to item 2 of Address
set field "City" to item 3 of Address
set field "Zip" to item 4 of Address
```

In the example shown in Figure 7-4, the application executes the script directly when the cursor is in the appropriate field and the user presses Tab or Enter. Your application can also associate such a script with an object in the form of a script context, so that the script context is executed whenever a specified Apple event acts on the field. The section “Using a Script Context to Handle an Apple Event,” which begins on page 7-25, describes this approach in more detail.

See “Manipulating and Executing Scripts,” which begins on page 7-22, for an overview of methods your application can use to save and load script data, compile source data, and perform other useful tasks with scripting component routines. The chapter “Scripting Components” in this book provides full implementation details, including sample code and human interface guidelines for associating scripts with objects.

Making Your Application Scriptable

To make your application scriptable, you need to

- define a hierarchy of Apple event objects within your application that you want client applications to be able to identify—that is, which objects can be contained by other Apple event objects in your application, which properties each kind of object can have, and so on
- write Apple event handlers, object accessor functions, and other routines required to implement the Apple events and related object classes that you want to support
- create an `'aete'` resource

The chapters “Introduction to Apple Events,” “Responding to Apple Events,” and “Resolving and Creating Object Specifier Records” in this book describe how to perform the first two tasks. The extent to which scripts can control your application depends mainly on the extent of your application’s support for Apple events. For example, if your application does not provide the Apple event handlers and object accessor functions required to locate and manipulate windows, users will not be able to use scripts to control your application’s windows. Although you should use the definitions in the *Apple Event Registry: Standard Suites* whenever possible, you have considerable freedom to extend or limit your implementation of the standard Apple events according to the needs of your application.

The OSA makes it possible to design new kinds of applications that always operate in the background and can be controlled only by means of scripts. For example, it is possible to design a simple telecommunications program that can log on to a network, send and receive text files created by another application, and perform other basic operations in response to scripts without providing any other form of user interface. Such an application would not need to support Apple events that control window movement, the File menu, or the Edit menu; instead, it would need to support only those Apple events that execute its basic telecommunications operations.

At the other extreme, some applications allow users to arrange windows, palettes, and dialog boxes on their screen in many different ways, or to customize menus or other aspects of the presentation of information. If such an application can respond to scripts that control windows, dialog boxes, specialized preferences, and other aspects of the presentation of information, it can allow users who might not otherwise explore those capabilities to take advantage of them. For example, a naive user could execute a script that sets up a powerful word processor with the appropriate menus, window and palette arrangement, and formatting templates for a particular task, such as producing a company newsletter.

Scripting components use `'aeut'` and `'aete'` resources to associate Apple event codes supported by your application with corresponding human-language terms used in scripts that control your application. Each scripting component supplies an `'aeut'` resource, and each scriptable application provides an `'aete'` resource. The next section introduces the `'aeut'` and `'aete'` resources.

About Apple Event Terminology Resources

As explained in the chapter "Introduction to Apple Events" in this book, applications can support different combinations of the standard suites of Apple events. Applications can also extend the definitions of individual Apple events and object classes, or define custom Apple events and object classes. Scripting components use the ***Apple event user terminology resources,*** `'aeut'` and `'aete'`, to associate the IDs, keywords, and other codes used in Apple events with the corresponding human-language terms used in scripts that control your application.

The Apple event user terminology (`'aeut'`) resource contains terminology information for all the standard suites of Apple events defined in the *Apple Event Registry: Standard Suites.* The resource consists of a sequence of concatenated arrays that map human-language names to each of the following:

- the ID defined for each suite
- the Apple events defined for each suite
- the parameters defined for each Apple event
- the Apple event object classes defined for each suite
- the properties defined for each object class
- the elements defined for each object class
- the key forms defined for each element class
- the comparison operators defined for each suite
- the values for enumerators defined for each suite

Each scripting component provides its own 'aeut' resource. A scripting component can also provide different versions of the 'aeut' resource; for example, the user terminology provided by the 'aeut' resource for the AppleScript Japanese dialect component is in Japanese. The IDs, keywords, and other codes listed in the 'aeut' resource are based on the *Apple Event Registry: Standard Suites* and do not vary from one version to another.

An 'aete' resource has the same format as the 'aeut' resource but serves a different purpose. Each scriptable application must include its own 'aete' resource describing which of the standard suites listed in the 'aeut' resource it supports and providing other application-specific information. Since the human-language equivalents for the standard suites are defined in the 'aeut' resource, applications that support standard suites without any modifications do not have to define such equivalents; instead, they can simply list, in the 'aete' resource, the suites they support. The scripting component associates the standard suites listed in the 'aete' resource with the corresponding Apple event descriptions in its 'aeut' resource.

Applications can also use the 'aete' resource to describe extensions to the standard suites, such as additional parameters for standard Apple events, additional properties and element classes for the standard Apple event object classes, and additional key forms for each element class. Information about such extensions must be included in the appropriate arrays of the 'aete' resource, along with the equivalent human-language terms. Similarly, an application can use the 'aete' resource to describe the parts of each standard suite it supports (if it doesn’t support the entire suite) and any custom Apple events or Apple event object classes defined by the application.

The human language in which your Apple event extensions or custom Apple events are displayed in scripts depends on the corresponding user terminology you specify in your application’s 'aete' resource. Therefore, if your application implements such extensions or custom Apple events, you must provide a separate version of this resource for each localized version of your application.

Scripting components can use the information in the 'aete' and 'aeut' resources in a variety of ways. The next section, “How AppleScript Uses Terminology Information,” describes how the AppleScript component uses these resources when it executes or records a script. The next chapter, “Apple Event Terminology Resources,” describes how to create an 'aete' resource for your application.

If you want users to be able to control your application with scripts written in the AppleScript scripting language, you also need to know how the AppleScript component interprets AppleScript commands that trigger Apple events. In this way, you can make sure you support Apple events and specify the user terminology for your 'aete' resource in a way that translates easily into AppleScript statements. The section “Defining Terminology for Use by the AppleScript Component,” which begins on page 8-3, discusses these issues. If you implement Apple events so that they translate into logical and useful AppleScript scripts, your implementation will probably work well with other scripting components that resemble AppleScript.

How AppleScript Uses Terminology Information

The manner in which the AppleScript component uses the information in `'aete'` resources depends on specific characteristics of the AppleScript scripting language. An AppleScript expression consists of an internal compiled form and corresponding expressions in ***dialects,*** or versions of the AppleScript scripting language that resemble different human languages. Users can select the dialect they want to use from within the Script Editor application. If a script is displayed in a window and the user selects a different dialect, the AppleScript component converts the script to the new dialect. Users can install additional dialects as necessary.

This section describes how the AppleScript component uses the information in the `'aeut'` and `'aete'` resources, not how it obtains that information. For a description of the methods available to scripting components for loading information from terminology resources, see "Dynamic Loading of Terminology Information" on page 7-20.

Figure 7-5 shows how the AppleScript component uses information from its `'aeut'` resource and an application's `'aete'` resource to execute a script that consists of AppleScript statements displayed in a script editor window. When a user executes the script from the script editor (for example, by pressing the Run button in the Script Editor application), the AppleScript component first compiles the script into the equivalent compiled expressions, using information from its `'aeut'` resource and the application's `'aete'` resource to map application-specific terms in the script with the equivalent Apple events and Apple event parameters. The AppleScript component then evaluates each expression and performs actions or sends Apple events as appropriate.

For example, the AppleScript component evaluates the expression

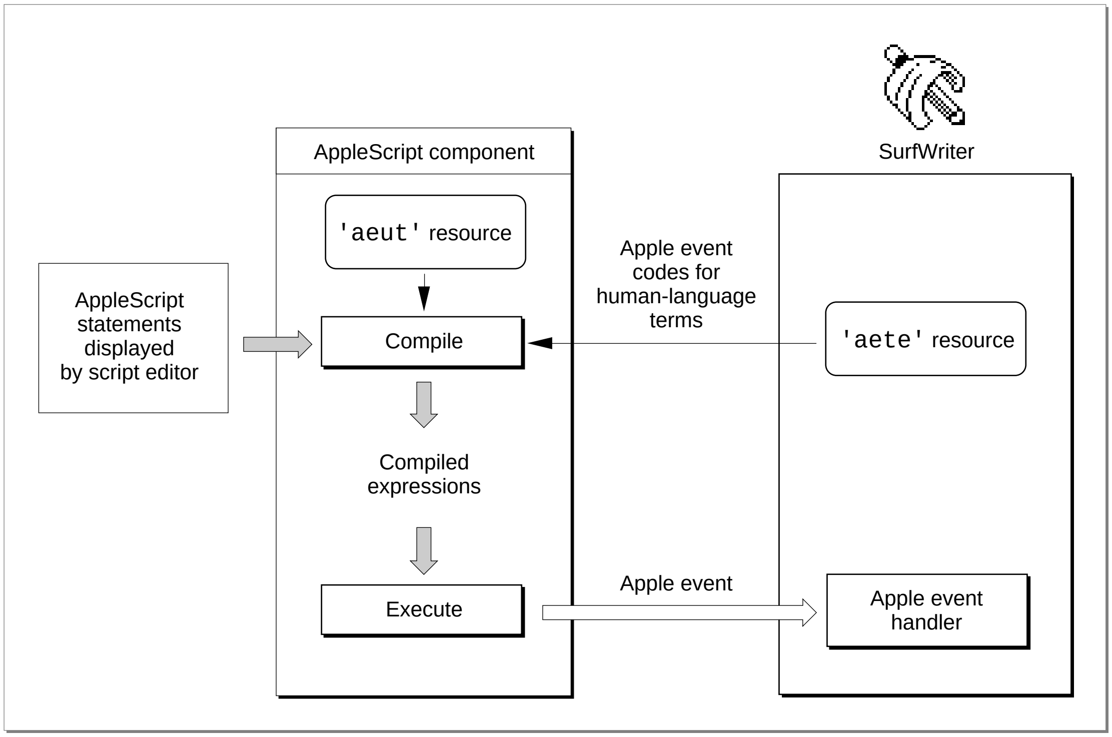
```
2*3
```

as the value 6. The AppleScript component can then decompile and display this value in the script editor window, assign it to a variable, or otherwise manipulate it according to the rest of the script. However, to compile the statement

```
print Chart 1 of document "Sales Report"
```

the AppleScript component uses its `'aeut'` resource and the SurfWriter application's `'aete'` resource to associate the terms used in the script with the Print Apple event, the object class for charts, and the object class for documents, so that it can describe the event accurately in the form of a compiled expression. When the AppleScript component evaluates the compiled expression, it creates and sends a Print event whose direct parameter is an object specifier record that the SurfWriter application can resolve as the specified chart. The SurfWriter application then handles the Apple event by printing the chart as requested.

Figure 7-5 Role of the `'aete'` and `'aeut'` resources when the AppleScript component compiles and executes a script

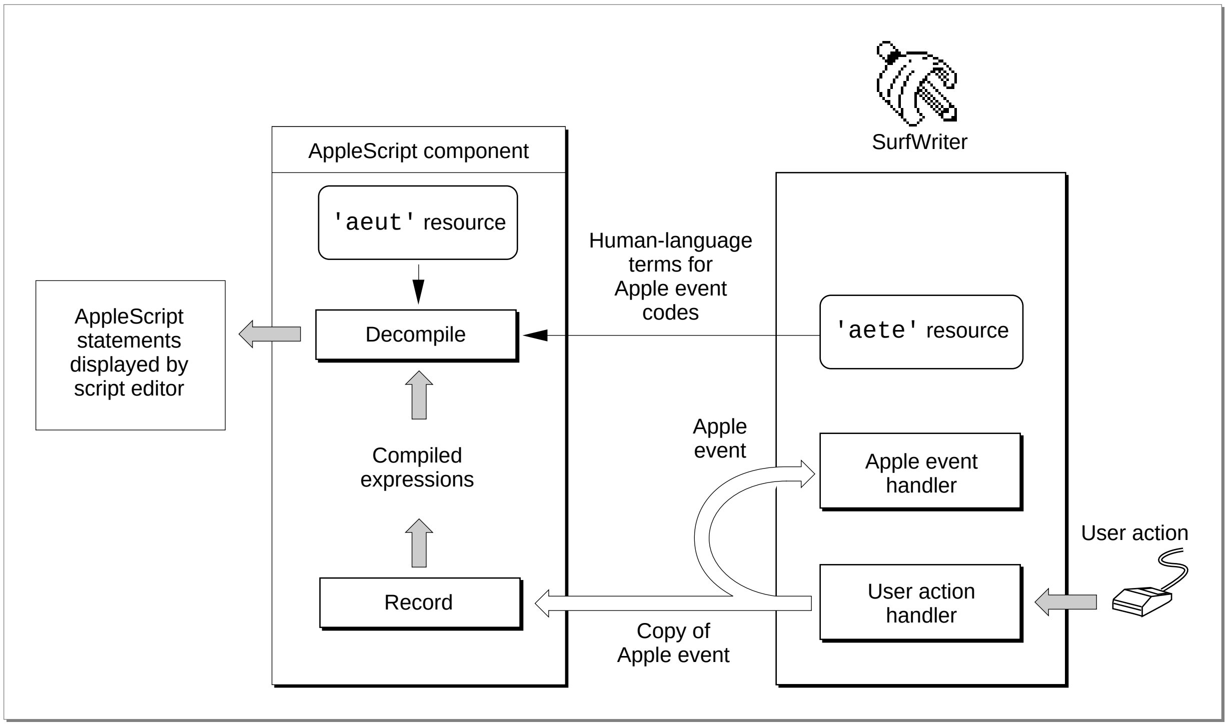


Note that although Figure 7-5 shows only one Apple event generated as a result of executing a script, the AppleScript component could also send a series of Apple events to several different applications, depending on the content of the script.

A recordable application generally needs to be able to send itself a subset of the Apple events that it can handle as a scriptable application. A ***recordable event*** is any Apple event that any recordable application sends to itself while recording is turned on for the local computer (with the exception of events that the application explicitly identifies as not for recording purposes). After a user turns on recording from the Script Editor application, the Apple Event Manager sends copies of all recordable events to Script Editor. A scripting component previously selected by the user handles each copied event for Script Editor by translating the event and recording the translation as part of a Script Editor script. When a scripting component executes a recorded script, it sends the corresponding Apple events to the applications in which they were recorded.

Every scripting component must be able to handle copies of recordable events sent to a recording process (such as Script Editor) by recording them in an appropriate form. For example, as shown in Figure 7-6, the AppleScript component records copies of recordable events in the form of compiled expressions. The AppleScript component can then use information from its `'aeut'` resource and the application's `'aete'` resource to translate the compiled expressions into the appropriate human-language terms and display them as AppleScript statements in the script editor window. When the user opens a recorded script in Script Editor and presses Run, the AppleScript component recompiles the script if necessary and sends the Apple events described by the compiled expressions to the SurfWriter application, just as in Figure 7-5.

Figure 7-6 Role of the `'aete'` and `'aeut'` resources when the AppleScript component records and decompiles a script



If the user copies a chart from one document to another document and the SurfWriter application performs this task by sending itself Apple events, the equivalent statements in the recorded script might look something like this:

```
tell application "SurfWriter"
   select Chart 1 of document "Sales Chart"
   copy
   select paragraph 3 of document "Monthly Report"
   paste
end tell
```

To display these statements in the script editor window, the AppleScript component first translates the Set Data, Copy, and Paste Apple events sent by the recordable application into compiled expressions. It then uses information from its `'aeut'` resource and the application's `'aete'` resource to decompile the compiled expressions and pass the equivalent source data to the script editor for display to the user. After completing a recording session, the user can edit and save the resulting script and execute it again at any time.

As shown in Figure 7-5 and Figure 7-6, the AppleScript component uses information it obtains from the `'aeut'` and `'aete'` resources when it is compiling and decompiling scripts. Other scripting components might use the same information during execution or recording, or in other ways that are specific to each component.

Dynamic Loading of Terminology Information

When a scripting component needs information about the user terminology defined in your application's `'aete'` resource, it sends a Get AETE event to your application. If your application does not handle the Get AETE event, the scripting component reads the terminology information it needs directly from your application's `'aete'` resource.

Your application does not need to handle the Get AETE event unless it provides separate `'aete'` resources for plug-in components. If your application does provide separate plug-in components, the Get AETE event allows it to gather terminology information from the `'aete'` resources for the components that are currently running and add that information to the reply event.

If your application handles the Get AETE event, you must also provide a scripting size resource. A scripting size resource is a resource of type `'scsz'` that provides information about an application's capabilities and preferences for use by scripting components.

To take advantage of dynamic loading, your application must be running. Note that if your application does not provide a handler for the Get AETE event, the scripting component can obtain terminology information directly from your application's `'aete'` resource even if your application is not running.

Making Your Application Recordable

If you decide to make your application scriptable, you can also make it recordable. A ***recordable application*** is an application that uses Apple events to report user actions to the Apple Event Manager for recording purposes. A recordable event is any Apple event that a recordable application sends to itself while recording is turned on for the local computer (with the exception of events sent with the `kAEDontRecord` flag set in the `sendMode` parameter of `AESend`).

When a user turns on recording by clicking the Record button in the Script Editor application, the Apple Event Manager sends copies of all subsequent recordable events to Script Editor. The AppleScript component handles each copied event for Script Editor by translating it into compiled expressions and recording the compiled expressions as part of a script. (Figure 7-6 on page 7-19 shows how the AppleScript component uses the `'aete'` and `'aeut'` resources when it records a script.) The user can view the equivalent decompiled source data in Script Editor while the script is being recorded. When a user executes a recorded script, the AppleScript component sends the corresponding Apple events to the applications in which they were recorded.

Applications generally have two parts: the code that implements the application's user interface and the code that actually performs the work of the application when the user manipulates the interface. One way to make your application recordable is to separate these two parts of your application, using Apple events to connect user actions with the work your application performs. This is called ***factoring*** your application. In a fully factored application, almost all tasks are carried out in response to Apple events. The application translates low-level events that result in significant actions into recordable Apple events and then sends them to itself.

Factoring your application is the recommended method of making your application recordable. However, it is also possible for your application to report user actions by means of Apple events even though it actually performs those actions by some means other than Apple events. You can indicate that you want the Apple Event Manager to record events in this manner, without executing them, by adding the constant `kAEDontExecute` to the `sendMode` parameter of `AESend`.

Before you decide how to map the user's potential actions to recordable Apple events supported by your application, you need to answer these questions:

- What are the significant (that is, undoable) actions a user can perform with your application that you want to record?
- Which actions can you execute by means of Apple events, and which actions should cause Apple events to be sent but not executed?
- How do you want to record actions that can be described in a scripting language in several different ways?

For example, if your application is a word processor, the user's selection of a range of text should probably not generate an Apple event, because users often select various different pieces of text before deciding to do something to the selection. However, if a user changes the font of a selection, a recordable word processor should generate a corresponding Apple event so that the scripting component can record the change.

In general, a recordable application should generate Apple events for any user action that the user could reverse by choosing Undo. A recordable application can usually handle a greater variety of Apple events than it can record, because it must record the same action the same way every time even though Apple events might be able to trigger that action in several different ways.

For more information about recordable applications, factoring, and the Apple Event Manager's recording mechanism, see the chapter "Recording Apple Events" in this book. For a description of the role of the `'aete'` and `'aeut'` resources when the AppleScript component records a script, see "How AppleScript Uses Terminology Information," which begins on page 7-17.

Manipulating and Executing Scripts

Your application can use scripting component routines to manipulate and execute scripts written in any scripting language based on the OSA. This section describes how scripting components use script data and summarizes some of the tasks your application can perform by calling the standard scripting component routines.

Your application can manipulate and execute scripts regardless of whether it is scriptable or recordable. However, if your application is scriptable, you can easily make it capable of manipulating and executing scripts that control its own behavior. For example, the forms application shown in Figure 7-4 on page 7-13 uses standard scripting component routines to execute a script whenever the cursor is in the appropriate field and the user presses Enter or Tab. Applications can also use scripting component routines to allow users to edit, recompile, save, and load such scripts in order to adapt them to their own purposes.

Before using any scripting component routines, your application must open a connection with at least one scripting component. After opening a connection with a component, your application receives a component instance that it can use as the first parameter for any scripting component routine. You can use the Component Manager to establish a connection with the generic scripting component or to establish an explicit connection with any other scripting component. Your application can open connections with different scripting components under different circumstances and, if necessary, simultaneously.

To manipulate or execute scripts written in any scripting language based on the OSA, your application can open a connection with the ***generic scripting component.*** The generic scripting component in turn attempts to open connections dynamically with the appropriate scripting component for a given script. If your application opens a connection with the generic scripting component, it can load and execute scripts created by any scripting component that is registered with the Component Manager on the current computer. The generic scripting component also provides routines that allow you to determine which scripting component created a particular script and to perform other useful tasks when you are using multiple scripting components.

To manipulate and execute scripts written in a single scripting language only, your application can open an explicit connection with the scripting component for that language. In this case your application can load and execute only those scripts that were created by that component; however, your application can also take advantage of additional routines and other special capabilities provided by the component.

After your application has established a connection with the appropriate scripting component for an existing script, it can use the standard scripting component routines to execute scripts. A script that has not yet been compiled consists of ***source data,*** or statements in a scripting language. Before executing source data, your application must use scripting component routines to compile it so that the scripting component can keep track of it in memory and execute it.

Scripting components can refer to at least three kinds of ***script data*** in memory:

- A ***compiled script*** consists of compiled code that an application can decompile into source data or execute using the standard scripting component routines.
- A ***script value*** consists of an integer, a string, a Boolean value, constants, PICT data, or any other fixed data returned or used by a scripting component in the course of executing a script.
- A ***script context*** maintains context information for the execution of other scripts. A script context can also contain executable statements in a scripting language. Like a compiled script, a script context can be decompiled as source data.

  For example, a script context can contain user-defined handlers for specific Apple events. In AppleScript, a script context that contains such handlers or other executable statements is called a ***script object.*** Handlers in a script object resemble HyperTalk message handlers. They consist of AppleScript statements and have no corresponding entry in Apple event dispatch tables.

Scripting components keep track of script data in memory by means of ***script IDs*** of type `OSAID`.

```
TYPE OSAID = LongInt;
```

A scripting component assigns a script ID to a compiled script or script context whenever the component creates or loads the corresponding script data. The scripting component routines that compile, load, and execute scripts all return script IDs, and you must pass valid script IDs to many of the other routines that manipulate scripts.

Applications most commonly use scripting component routines to

- compile source data and execute the resulting compiled script, so that a user can create a new script and execute it immediately from within the application
- get a handle to script data in a form that can be saved, and load and execute the script data again when necessary
- allow users to modify a script, then recompile and save the script
- redirect Apple events to script contexts

The remainder of this section provides an overview of the scripting component routines you can use to perform these tasks.

Your application can also use scripting component routines to

- get information about scripts
- get information about scripting components

- coerce script values to descriptor records and vice versa
- set a resume dispatch function and alternative send, create, and active functions for use by a scripting component
- control the recording process directly, turning recording off and on and saving the recorded script for use by your application

The chapter “Scripting Components” in this book provides detailed information about using all the standard scripting component routines as well as additional routines provided by the AppleScript component and the generic scripting component.

Compiling, Saving, Modifying, and Executing Scripts

This section introduces some of the scripting component functions your application can use to compile, save, modify, and execute scripts.

To create and execute a script using the Script Editor application, a user can type the script, then press the Run button to execute it. Your application can provide similar capabilities by using these functions to compile source data and execute the resulting compiled script:

- The `OSACompile` function takes a descriptor record with a handle to source data (usually text) and a script ID. If you specify `kOSANullScript` instead of an existing script ID, `OSACompile` returns a script ID for the new compiled script, which you can then pass to the `OSAExecute` function.
- The `OSAExecute` function takes a script ID for a compiled script and a script ID for a script context, executes the script, and returns a script ID for the resulting script value.

The binding of any global variables in the compiled script is determined by the script context whose script ID you pass to `OSAExecute`. If you pass `kOSANullScript` instead of the script ID for a script context, the scripting component provides its own default context. If you want to provide your own script context rather than using the scripting component default context, you can use either `OSACompile` or `OSAMakeContext` to create a script context, which you can load and store just like a compiled script.

After creating a script and trying it out, a user may want to save it for future use. Your application should normally save its scripts as script data rather than source data, so that it can reload and execute the data without recompiling it. Before saving script data, you must first call the `OSAStore` function to get a handle to the data in the form of a descriptor record. You can then save the data to disk as a resource or write it to the data fork of a document.

To allow a user to reload and execute a previously compiled and saved script, your application can call these functions:

- The `OSALoad` function takes a descriptor record that contains a handle to the saved script data and returns a script ID for the compiled script.
- The `OSAExecute` function takes a script ID for a compiled script and a script ID for a script context, executes the script, and returns a script ID for the resulting script value.

In most cases you will want to allow users to modify saved scripts and save them again. To allow a user to modify and save a compiled script, your application can call these functions:

- The `OSAGetSource` function takes a script ID and returns a descriptor record with a handle to the equivalent source data.
- The `OSACompile` function takes a descriptor record with a handle to source data and a script ID, and returns the same script ID updated so that it refers to the modified and recompiled script.
- The `OSAStore` function takes a script ID and returns a copy of the corresponding script data in the form of a storage descriptor record.

You can pass the script ID for the compiled script to be modified to the `OSAGetSource` function, which returns a descriptor record with a handle to the equivalent source data. Your application can then present the source data to the user for editing. When the user has finished editing the source data, you can pass the modified source data and the original script ID to the `OSACompile` function to update the script ID so that it refers to the modified and recompiled script. Finally, to obtain a handle to the modified script data so you can save it in a resource or write it to the data fork of a document, you can pass the script ID for the modified compiled script to the `OSAStore` function.

If your application has no further use for a compiled script or a resulting script value after successfully loading, saving, compiling, or executing a script, you can use the `OSADispose` function to release the memory assigned to them. The `OSADispose` function takes a script ID and releases the memory assigned to the corresponding script data. A script ID is no longer valid after the memory associated with it has been released. This means, for example, that a scripting component may assign a different script ID to the same compiled script each time you load it, and that a scripting component may reuse a script ID that is no longer associated with a specific script.

"Using Scripting Component Routines," which begins on page 10-7, provides more information about the standard scripting component routines described in this section.

Using a Script Context to Handle an Apple Event

One way to associate a script with an object is to associate a script context with a specific Apple event object—that is, with any object in your application that can be identified by an object specifier record. When an Apple event acts on an Apple event object with which a script context is associated, your application attempts to use the script context to handle the Apple event. This approach can be useful if you want to associate many different scripts with many different kinds of objects.

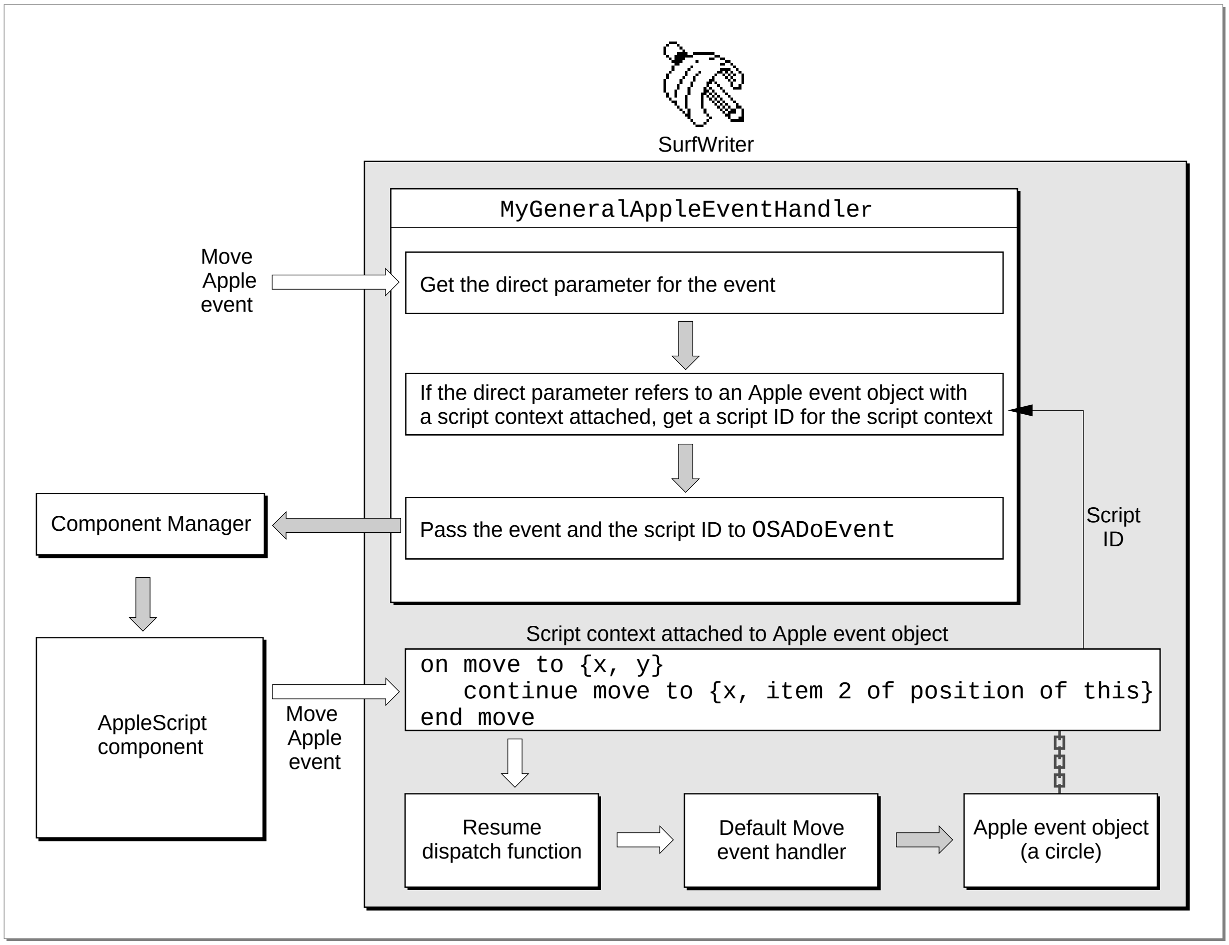
Figure 7-7 illustrates one way that an application can use a script context to handle an Apple event. This example shows how you can use a general Apple event handler to provide initial processing for all Apple events received by your application. If an Apple event acts on an object with which a script context is associated, the general handler attempts to use the script context to handle the event.

The SurfWriter application in Figure 7-7 associates script contexts (called script objects in AppleScript) with geometric shapes such as circles or squares. These script contexts can contain one or more user-defined handlers for specific Apple events. For example, the script context shown in Figure 7-7 is associated with a circle and contains this handler:

```
on move to {x, y}
   continue move to {x, item 2 of position of this}
end move
```

This handler exists only as AppleScript statements in the script context and doesn't have an entry in SurfWriter's Apple event dispatch table. SurfWriter does have its own standard Apple event handlers installed in its Apple event dispatch table. When SurfWriter receives a Move event that acts on the circle with which this script context is associated, SurfWriter uses the handler in the script context to modify its own standard handling of the event. The rest of this section describes how this works.

Figure 7-7 Using a handler in a script context to handle an Apple event


SurfWriter
MyGeneralAppleEventHandler
Move Apple event
Get the direct parameter for the event
If the direct parameter refers to an Apple event object with a script context attached, get a script ID for the script context
Component Manager
Pass the event and the script ID to OSADoEvent
Script ID
Script context attached to Apple event object
on move to {x, y}
   continue move to {x, item 2 of position of this}
end move
AppleScript component
Move Apple event
Resume dispatch function
Default Move event handler
Apple event object (a circle)

The MyGeneralAppleEventHandler function in Figure 7-7 is installed in SurfWriter's special handler dispatch table. Thus, MyGeneralAppleEventHandler provides initial processing for all Apple events received by SurfWriter. When it receives an Apple event, MyGeneralAppleEventHandler checks whether a script context is associated with the object on which the event acts. If so, MyGeneralAppleEventHandler passes the event and a script ID for the script context to the OSADoEvent function. If not, MyGeneralAppleEventHandler returns errAEEventNotHandled, which causes the Apple Event Manager to look for the appropriate handler in SurfWriter's Apple event dispatch table.

The OSADoEvent function looks for a handler in the specified script context that can handle the specified event. If the script context doesn't include an appropriate handler, OSADoEvent returns errAEEventNotHandled. If the script context includes an appropriate handler (in this example, a handler that begins on move), OSADoEvent attempts to use the handler to handle the event.

When it encounters the continue statement during execution of the on move handler shown in Figure 7-7, the AppleScript component calls SurfWriter's resume dispatch function. A ***resume dispatch function*** takes an Apple event and invokes the application's default handler for that event directly, bypassing the application's special handler dispatch table and the MyGeneralAppleEventHandler handler (or its equivalent). In this case, the AppleScript component uses SurfWriter's default Move handler to move the circle to a different location than the one specified in the original Move event. The location specified by {x, item 2 of position of this} has the same horizontal coordinate as the location specified by the original event, but specifies the circle's original vertical coordinate (item 2 of the circle's original position), thus constraining motion to a horizontal direction.

The AppleScript component calls the resume dispatch function as soon as it encounters a continue statement during script execution. For example, if the handler in Figure 7-7 contained additional indented statements after the continue statement, the AppleScript component would proceed with the execution of those statements after calling the resume dispatch function successfully.

A script context can modify the event and use the default Apple event handler to execute the modified event, as in this example; or it can override the default handler completely, performing some completely different action; or it can perform some action and then pass the original event to the application's default handler to be handled in the usual way. Script contexts associated with Apple event objects thus provide a way for users to modify or override the way an application responds to a particular Apple event that manipulates those objects.

A general Apple event handler can use the OSAExecuteEvent function instead of OSADoEvent to execute a script context. The main difference between these functions is is that OSAExecuteEvent returns the script ID for the resulting script value, whereas OSADoEvent returns a reply event.

To create a script context, pass the source data for the scripting-language statements you want the script context to contain to `OSACompile` with the `modeFlags` parameter set to `kOSACompileIntoContext`. The resulting script context is identical to a script context returned by the `OSAMakeContext` function, except that it contains compiled statements.

"Using a Script Context to Handle an Apple Event," which begins on page 10-19, describes this method of executing a script in more detail.