8

Version 1.2 (Frozen)

REQUIRED APPLE EVENTS

Includes Demonstration Program AppleEventsPascal

Introduction

System 7 introduced a new type of event, called the **high-level** event, along with a number of new Event Manager routines which allow applications to communicate with each other by exchanging high-level events.

Using high-level events, an application can instruct another application to perform a specific action, such as adding a row to a spreadsheet or changing the font size of a paragraph. An application can also request information from another application; for example, it might request a dictionary application to return the definition of a particular word.

Fig 1 shows the general event-handling mechanism which has existed since the introduction of System 7. In Fig 1, three different applications are communicating with each other by sending and receiving high-level events. Note that high-level events are placed in a separate queue maintained by the operating system and that a high-level event queue is maintained for each application that has announced itself as capable of receiving high-level events.

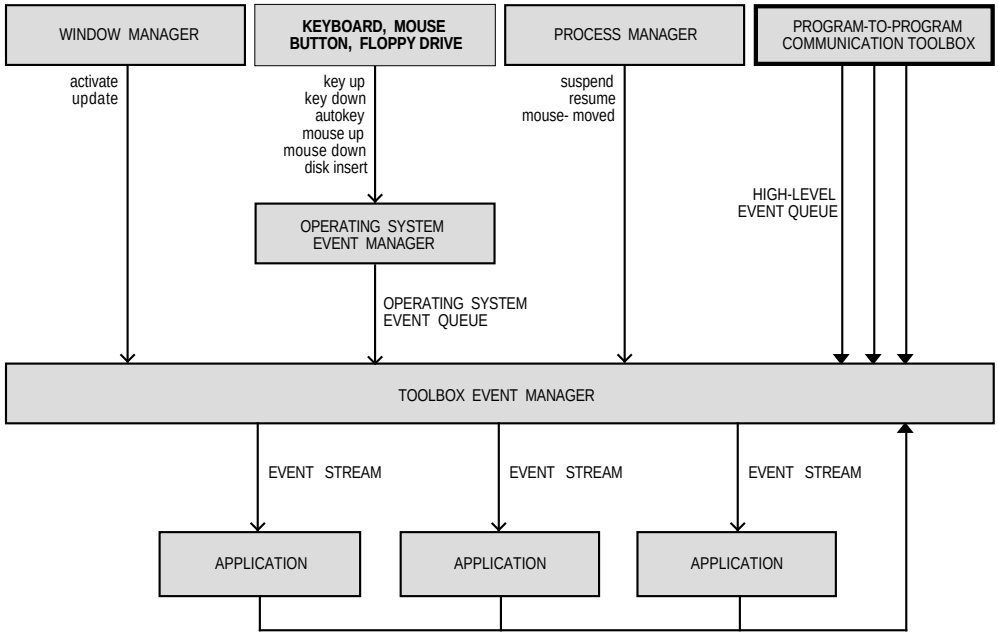

FIG 1 - EVENTS IN SYTEM 7

For effective communication between applications, an application must define the set of high-level events it responds to and let other applications know the events it accepts. For a high-level event sent by one application to be understood by another application, the sender and receiver must agree on a **protocol**, that is, on the way the event is to be interpreted.

Apple Events

Apple events are high-level events whose structure and interpretation are determined by the **Apple Event InterProcess Messaging Protocol** (AEIMP). Applications typically use Apple events to request services and information from other applications and to provide services and information in response to such requests.

Communication between two applications which support Apple events is initiated by a **client application**, which sends an Apple event to request a service or information. The application providing the service or information is called a **server application**.[1] Fig 2 shows a common Apple event, called the Open Documents event. The Finder (which is, itself, an application) is the client; it requests that the application My Application open the documents named Document A and Document B. My Application responds by opening windows for the specified documents.

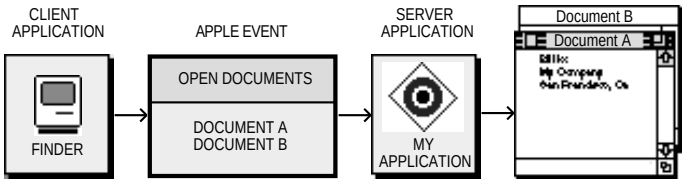

FIG 2 - CLIENT AND SERVER

To identify Apple events and respond appropriately, every application can rely on a vocabulary of standard Apple events which developers and Apple have established for all applications to use. These events are defined in the Apple Event Registry: Standard Suites. The standard **suites** (groups of Apple events that are usually implemented together) include:

- The **required** suite, which consists of four Apple events that the Finder sends to applications. The **required Apple events** are:
    - Open Application.
    - Open Documents.
    - Print Documents.
    - Quit Application.
- The **core** suite, which consists of the basic Apple events, including Get Data, Set Data, Move, Delete and Save, that nearly all applications use to communicate.
- The **functional-area** suite, which consists of a group of Apple events which support a related functional area, and which include the Text suite and the Database suite.

Required Apple Events

In System 7 and later versions of the system software, the Finder uses the required Apple events as part of the mechanism for launching and terminating applications. To be System 7-friendly, therefore, your application must support the required Apple events.

[1] An application can also send Apple events to itself, thus acting as both client and server.

This chapter is concerned with the required Apple events only, exploring the subject of Apple events only to the extent necessary to gain an understanding of the measures involved in supporting the required suite.

Apple Event Attributes and Parameters

When an application creates and sends an Apple event, the Apple Event Manager uses arguments passed to Apple Event Manager routines to construct the data structures that make up the Apple event. An Apple event comprises **attributes** (which identify the Apple event and denote its task) and, often, **parameters** (which contain information to be used by the target application).

Apple Event Attributes

An Apple event attribute is a record which identifies the **event class**, **event ID**, target application, and other characteristics of an Apple event. Taken together, the attributes denote the task to be performed on any data specified in the event's parameters. After receiving an Apple event, a server application can use Apple Event Manager routines to extract and examine its attributes. Apple events are identified by their event class and event ID attributes.

Event Class

The event class is the attribute that identifies a group of related Apple events. It appears in the `message` field of the event record for an Apple event (see Fig 3). For example, the required Apple events have the value `'aevt'` in the `message` field of their event records. `'aevt'` is represented by the constant `kCoreEventClass`.

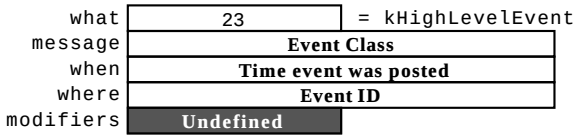

FIG 3 - CONTENTS OF AN EVENT RECORD - HIGH LEVEL (APPLE) EVENT

Event ID

The event ID is the attribute which identifies the particular event within the event class. In conjunction with the event class, the event ID uniquely identifies the Apple event and communicates what action the Apple event should perform. It appears in the `where` field of the event record for an Apple event (see Fig 3). For example, the event ID of an Open Documents event has the value `'odoc'`, which is represented by the constant `kAEOpenDocuments`. The `kCoreEventClass` constant in combination with the `kAEOpenDocuments` constant identifies the Open Documents event to the Apple Event Manager.

The following are the event IDs for the four required Apple events:

Event ID	Value	Description
`kAEOpenApplication`	`'oapp'`	Perform tasks required when a user opens your application.
`kAEOpenDocuments`	`'odoc'`	Open documents
`kAEPrintDocuemnts`	`'pdoc'`	Print Documents
`kAEQuitApplication`	`'quit'`	Quit your application.

Target Application

In addition to the event class and event ID, every Apple event must include an attribute which specifies the target application's address.

Apple Event Parameters

An Apple event parameter is a record containing data that the target application uses. Apple events can use standard data types, such as strings of text, long integers, boolean values, and alias records, for the data in their parameters. As with attributes, a client application can use Apple Event Manager routines to extract and examine the parameters of an Apple event it has received.

There are various kinds of Apple event parameters, including **direct parameters** and **additional parameters**.

Direct Parameters

A direct parameter usually specifies the data to be acted upon by the target application. For example, a list of documents is contained in the direct parameter of the Print Documents event.

Additional Parameters

Some Apple events also take additional parameters, which the target application uses in addition to the data specified in the direct parameter. For example, an Apple event for arithmetic operations may include additional parameters which specify operands in an equation.

Required and Optional Parameters

All parameters are either **required parameters** or **optional parameters.** A required parameter is one which must be present for the target application to carry out the task denoted by the Apple event. An optional parameter is a supplemental Apple event parameter that can also be used to specify data to a target application. Direct parameters are usually defined as **required parameters** in the Apple Event Registry - Standard Suites.

Interpreting Apple Event Attributes and Parameters

Fig 4 shows the major Apple event attributes and direct parameter for the Open Documents event.

To process this event, your application would use the `AEProcessAppleEvent` function, which uses the event class and event ID attributes to dispatch the event to My Application's Open Documents handler. In response, the Open Documents handler opens the documents specified in the direct parameter.

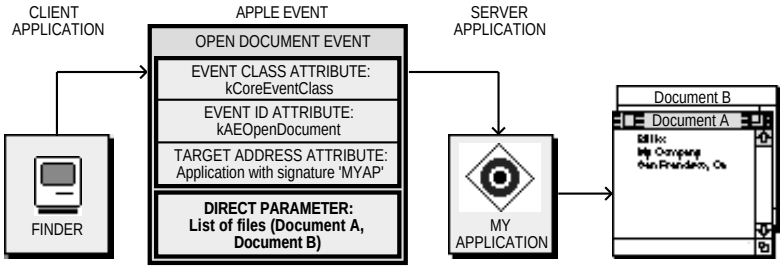

FIG 4 - MAJOR ATTRIBUTES AND DIRECT PARAMETERS IN AN OPEN DOCUMENTS EVENT

Data Structures Within Apple Events

The Apple Event Manager constructs its own internal data structures to contain the information in an Apple event.

Descriptor Records

Descriptor records are the building blocks used by the Apple Event Manager to construct Apple event attributes and parameters. A **descriptor record** is a data structure of type `AEDesc`. It consists of a handle to data and a descriptor type which identifies the type of data to which the handle refers:

```
type
AEDesc = record
  descriptorType : DescType; {Type of data.}
  dataHandle : Handle;         {Handle to data.}
  end;
```

The **descriptor type** is a structure of type `DescType` which, in turn, is of data type `ResType`, that is, a four-character code. Constants are used in place of these codes when referring to descriptor types. The following are some of the major descriptor type constants, their values, and the kind of data they identify:

Descriptor Type	Value	Description of Data
`typeChar`	`'TEXT'`	Unterminated string.
`typeType`	`'type'`	Four-character code.
`typeBoolean`	`'bool'`	One-byte Boolean value.
`typeLongInteger`	`'long'`	32-bit integer.
`typeAEList`	`'list'`	List of descriptor records.
`typeAERecord`	`'reco'`	List of keyword-specified descriptor records.
`typeAppleEvent`	`'aevt'`	Apple event record.
`typeFSS`	`'fss '`	File system specification.
`typeKeyword`	`'keyw'`	Apple event keyword.
`typeNull`	`'null'`	Nonexistent data (handle whose value is nil).

The following illustrates the logical arrangement of a descriptor record with a descriptor of type `typeChar`, which specifies that the data handle refers to an unterminated string:

Data Type `AEDesc`

Descriptor type:	`typeChar`
Data:	`"Summary of Sales"`

The following illustrates the logical arrangement of a descriptor record with a descriptor type of `typeType`, which specifies that the data handle refers to a four-character code (in this case the constant `kCoreEventClass`, whose value is `'aevt'`):

Data Type `AEDesc`

Descriptor type:	`typeType`
Data:	`(kCoreEventClass)`

Address Descriptor Record

Every Apple event includes an attribute specifying the address of the target application. A descriptor record which contains an application's address is called an **address descriptor record**:

```
AEAddressDesc = AEDesc;    {An AEDesc which contains addressing data.}
```

The address in an address descriptor record can be specified as one of the four basic types (or as any other descriptor types you define that can be coerced to one of these types):

Descriptor Type	Value	Description
`typeApplSignature`	`'sign'`	Application signature.
`typeSessionID`	`'ssid'`	Session reference number.
`typeTargetID`	`'targ'`	Target ID record.
`typeProcessSerialNumber`	`'psn '`	Process serial number.

Like several other data structures defined by the Apple Event Manager for use in Apple event attributes and Apple event parameters, an address descriptor record is identical to a descriptor record of data type `AEDesc`; the only difference is that the data for an address descriptor record must always consist of an application's address.

Keyword-Specified Descriptor Records

After the Apple Event Manager has assembled the necessary descriptor records as the attributes and parameters of an Apple event, your application must use Event Manager routines to request each attribute and parameter by **keyword**. Keywords are arbitrary names used by the Apple Event Manager to keep track of various descriptor records. The `AEKeyword` data type is defined as a four-character code:

```
type
FourCharCode = UNSIGNEDLONG;
AEKeyword = FourCharCode;
```

Constants are typically used to represent keywords.

Keywords for Attributes. Here is a partial list of keyword constants for Apple event attributes:

Attribute Keyword	Value	Description
`keyEventClassAttr`	`'evcl'`	Event class of Apple event.
`keyMissedKeywordAttr`	`'miss'`	Keyword for first required parameter remaining in an Apple event.
`keyAddressAttr`	`'addr'`	Address of target or client application.
`keyEventIDAttr`	`'evid'`	Event ID of Apple event.
`keyEventSourceAttr`	`'esrc'`	Nature of the source application.
`keyReturnIDAttr`	`'rtid'`	Return ID for reply Apple event.

Keywords for Parameters. Here is a list of keyword constants for commonly used Apple event parameters:

Parameter Keyword	Value	Description
`keyDirectObject`	`'----'`	Direct parameter.
`keyErrorNumber`	`'errn'`	Error number parameter.
`keyErrorString`	`'errs'`	Error string parameter.

The Apple Event Manager associates keywords with specific descriptor records by means of a **keyword-specified descriptor record**, a data structure of type `AEKeyDesc` that consists of a keyword and a descriptor record:

```
type
AEKeyDesc = record
  descKey : AEKeyword;    {Keyword}
  descContent : AEDesc;   {Descriptor record.}
  end;
```

The following illustrates a keyword-specified descriptor record with the keyword `keyEventClassAttr`, the keyword that identifies an event class attribute. It shows the logical arrangement of the event class attribute for the Open Documents event shown at Fig 4.

Data Type `AEKeyDesc`

Keyword:	`keyEventClassAttr`
Descriptor Record:	Descriptor Type: `typeType` Data: Event Class `(coreEventClass)`

Descriptor Lists, AE Records, and AppleEvent Records

Descriptor Lists

When extracting data from an Apple event, you use Apple Event Manager functions to copy data to a buffer specified by a pointer, or to return a descriptor record whose data handle refers to a copy of the data, or to return lists of descriptor records (called **descriptor lists**).

A descriptor list is a data structure of type `AEDescList` defined by the data type `AEDesc`. That is, a descriptor list is a descriptor record whose handle refers to a list of other descriptor records (unless it is an empty list):

```
type
AEDescList = AEDesc;   {List of descriptor records.}
```

The following illustrates the logical arrangement of the descriptor list that specifies the direct parameter of the Open Documents event shown at Fig 4. This descriptor list consists of a list of descriptor records which contain alias records to filenames.

Data Type `AEDescList`

Descriptor type:	`typeAEList`
Data:	**List of descriptor records:**

Descriptor type:	`typeAlias`
Data:	Alias record for filename (Document A)
Descriptor type:	`typeAlias`
Data:	Alias record for filename (Document B)

This descriptor list provides the data for a keyword-specified descriptor record.

AE Record

Keyword-specified descriptor records for Apple event parameters can in turn be combined into an **AE record**, which is a descriptor list of type `AERecord`:

```
type
AERecord = AEDescList;   {List of keyword-specified descriptor records.}
```

The handle for a descriptor list of data type `AERecord` refers to a list of keyword-specified descriptor records that can be used to construct Apple event parameters. An AE record has the descriptor type `typeAERecord` and can be coerced to several other descriptor types.

Apple Event Record

An **Apple event record**, which is different from an AE record, is another special descriptor list of data type `AppleEvent` and descriptor type `typeAppleEvent`:

```
type
AppleEvent = AERecord;   {List of attributes and parameters for Apple event.}
```

An Apple event record describes a full-fledged Apple event. Like the data for an AE record, the data for an Apple event record consists of a list of keyword-specified descriptor records. Unlike an AE record, the data for an Apple event record is divided into two parts, one for attributes and one for parameters. This division allows the Apple event to distinguish between an Apple event's attributes and its parameters.

Passing Descriptor Lists, AE Records and Apple Event Records to Apple Event Manager Functions

Descriptor lists, AE records and Apple event records are all descriptor records whose handles refer to a nested list of other descriptor records. The data associated with each data type may be organised differently and used by the Apple Event Manager for different purposes. In each case, however, the data is identified by a handle in a descriptor record. This means that you can pass an Apple event record to any Apple Event Manager function that expects an AE record. Similarly, you can pass Apple event records and AE records, as well as descriptor lists and descriptor records, to any Apple Event Manager functions that expect records of data type `AEDesc`.

Example Complete Apple Event

Fig 5 shows an example of a complete Apple event — a data structure of type `AppleEvent` containing a list of keyword-specified descriptor records that name the attributes and parameters of an Open Documents event.

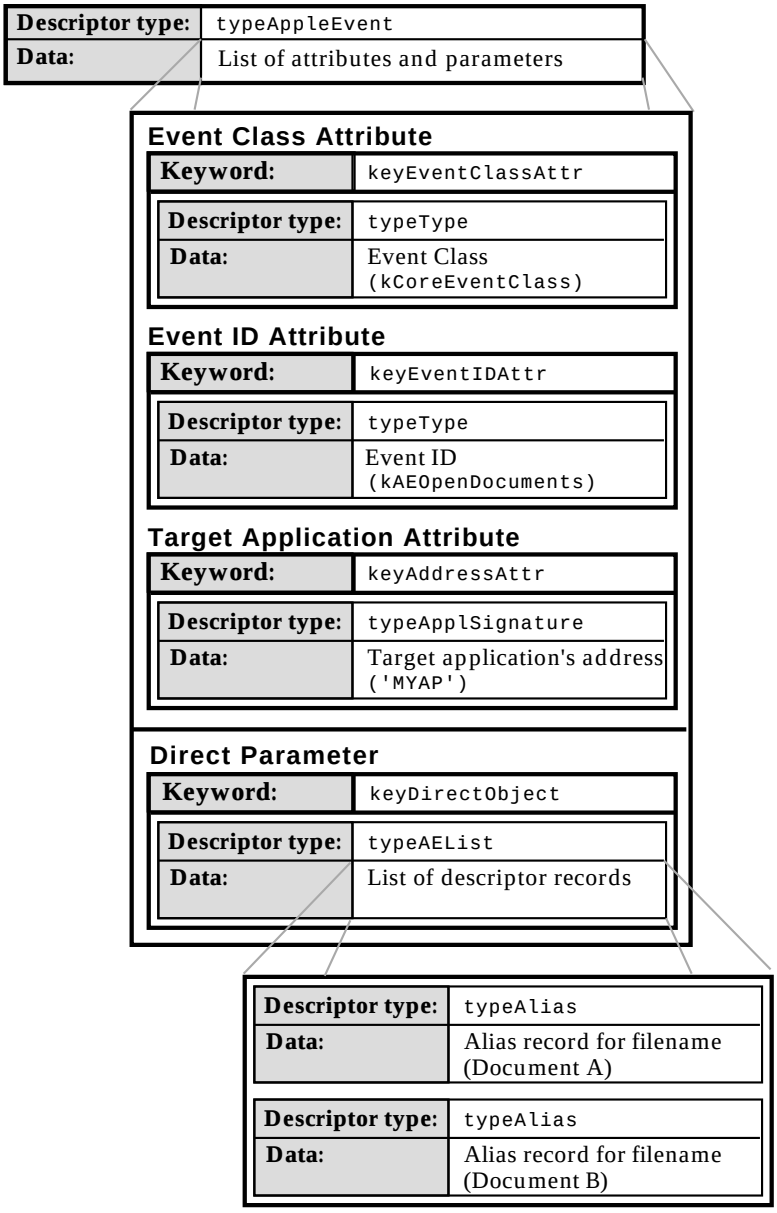

FIG 5 - DATA STRUCTURES WITHIN AN OPEN DOCUMENTS EVENT

Handling Apple Events

A client application uses the Apple Event Manager to create and send an Apple event requesting a service or information. A server application responds by using the Apple Event Manager to process the Apple event, extract data from the attributes and parameters of the Apple event and, if necessary, add requested data to the reply event returned by the Apple Event Manager to the client application.

As a first step in supporting Apple events, and as previously stated, your application should support the required Apple events sent by the Finder. To support the required Apple events, you must:

- Set the `isHighLevelEventAware` flag in the `'SIZE'` resource of your application.

- Test for high-level events in your application's event loop. An Apple event (like all high-level events) is identified by a message class of `kHighLevelEvent` in the `what` field of the event record. Your application should therefore test the `what` field of the event record to determine whether it contains the value represented by `kHighLevelEvent`.

- Use `AEProcessAppleEvent` to process the Apple events. `AEProcessAppleEvent` first identifies the Apple event by examining the data in the event class and event ID attributes. It then uses that data to call the appropriate Apple event handler provided by your application.

- Provide handlers for the required Apple events in your application. Your Apple event handlers must extract the pertinent data from the Apple event, perform the requested action, and return a result.

- Use `AEInstallEventHandler` to install your Apple event handlers. This function installs handlers in an **Apple event dispatch table** for your application. The Apple Event Manager uses this table to map Apple events to handlers in your application When your application calls `AEProcessAppleEvent`, the Apple Event Manager checks the dispatch table and, if your application has installed a handler for the event, calls the handler. Each entry in the Apple event dispatch table should specify:

  - The event class.
  - The event ID.
  - The address of the Apple event handler.
  - A reference constant.[2]

  Accordingly, the parameters for the call to `AEInstallEventHandler` are the event class, the event ID, a pointer to the event handler, a reference constant, and `false`.[3]

Apple Event Handlers

Each Apple event handler must be a function which uses this syntax:

```
function theEventHandler(var theAppleEvent: AppleEvent; var reply: AppleEvent;
                handlerRefcon: longint): OSErr;
```

`appleEvent`	The Apple event to handle. Your handler uses Apple Event Manager functions to extract any parameters and attributes from the Apple event and then perform the necessary processing.
`reply`	The default reply provided by the Apple Event Manager.
`handlerRefcon`	Reference constant stored in the Apple event dispatch table entry for the Apple event. Your handler can ignore this parameter if your application does not use the reference constant.

Apple event handlers must generally perform the following tasks:

- Extract the parameters and attributes from the Apple event.

- Check that all required parameters have been extracted.

---

[2]The reference constant is passed to your handler by the Apple Event Manager each time your handler is called. Your application can use this reference constant for any purpose. If your application does not use the reference constant, specify 0.

[3]`false` causes the handler to be installed in the application's Apple event dispatch table. `true` causes the handler to be installed in the system's Apple event dispatch table. The system Apple event dispatch table is a table in the system heap containing handlers that are available to all applications and processes running on the same computer. The handlers in your application's Apple events dispatch table are available only to your application. If `AEProcessAppleEvent` cannot find a handler for the Apple event in your application's Apple event dispatch table, it looks in the system Apple event dispatch table for a handler. If it does not find a handler in the system table, it returns the `errAEEventNotHandled` result code.

- Perform the action requested by the Apple event.
- Dispose of any copies of the descriptor records that have been created.
- Add information to the reply Apple event if requested.

Extracting and Checking Data

You must use Apple Event Manager functions to extract the data from Apple events. The following are the main functions involved:

Function	Description
`AEGetAttributePtr`	Uses a buffer to return a copy of the data contained in an Apple event attribute. Used to extract data of fixed length or known maximum length.
`AEGetParamDesc`	Returns a copy of the descriptor record or descriptor list for an Apple event parameter. Usually used to extract data of variable length, for example, to extract the descriptor list for a list of alias records specified in the direct parameter of an Open Documents event.
`AECountItems`	Returns the number of descriptor records in a descriptor list. Used, for example, to determine the number of alias records for documents specified in the direct parameter of an Open Documents event.
`AEGetNthPtr`	Uses a buffer to return a copy of the data for a descriptor record contained in a descriptor list. Used to extract data of fixed length or known maximum length, for example, to extract the name and location of a document from the descriptor list specified in the direct parameter of the Open Documents event.

Data Type Coercion. You can specify the descriptor type in the resulting data from these functions. If this type is different from the descriptor type of the attribute or parameter, the Apple Event Manager attempts to coerce it to the specified type. In the direct parameter of the Open Documents event, for example, each descriptor record in the descriptor list is an **alias record** and each alias record specifies a document to be opened. All your application usually needs to open a document is a **file system specification record** (`FSSpec`) of the document. When you extract the descriptor record from the descriptor list, you can request that the Apple Event Manager return the data to your application as a file system specification record instead of an alias record.

Checking That All Required Parameters Have Been Retrieved. After extracting all known Apple event parameters, your handler should check that it has retrieved all the parameters that the source application considered to be required. To do this, determine whether the `keyMissedKeywordAttr` attribute exists. If this attribute does exist, your handler has not retrieved all the required parameters, and it should return an error.

Interacting With the User

In some cases, the server application may need to interact with the user when it handles an Apple event. For example, your handler for the Print Documents event may need to display a print options dialog box and get settings from the user before printing .

The Apple Event Manager does not allow the server application to interact with the user in response to a client application's Apple event unless at least two conditions are met:

- First, the client application must set flags in the `sendMode` parameter of the `AESend` function to indicate that user interaction is allowed.
- Second, the server application must either:
    - Set flags to the `AESetInterActionAllowed` function indicating that user interaction is allowed. (These flags relate to permitting interaction where the client and server are the same application, the client application is on the same computer as the server, or the client is on any computer.)
    - Set no user interaction preferences (that is, make no call to `AESetInterActionAllowed`), in which case `AEInteractWithUser` (the function used to initiate interaction with the user

when your application is a server responding to an Apple event) assumes that only interaction with a client on the local computer is allowed.

If these two conditions are met, and if `AEInteractWithUser` determines that both the client and server applications allow user interaction under the current circumstances, `AEInteractWithUser` brings your application to the foreground if it is not already in the foreground. Your application can then display its dialog box or alert box or otherwise interact with the user.

Performing the Requested Action and Returning a Result

When your application responds to an Apple event, it should perform the standard action requested by the event.

Your Apple event handler should always set its function result to either `noErr`, if it successfully handles the Apple event, or to a non-zero result code if an error occurs. If your handler returns a non-zero result code, the Apple Event Manager adds a `keyErrorNumber` parameter to the reply Apple event. This parameter contains the result code that your handler returns.

Disposing of Copies of Descriptor Records

When your handler is finished with a copy of a descriptor record created by `AEGetParamDesc` and related functions, it should dispose of it by calling `AEDisposeDesc`.

Required Apple Events - Contents and Required Action

Your application receives the four required Apple events from the Finder in these circumstances:

- If your application is not open and the user elects to open your application from the Finder without opening or printing any documents, the Finder launches your application (using the Process Manager) and sends it an Open Application event.
- If your application is not open and the user elects to open one of your application's documents from the Finder, the Finder launches your application (using the Process Manager) and sends it an Open Documents event.
- If your application is not open and the user elects to print one of your application's documents from the Finder, the Finder launches your application (using the Process Manager) and sends it the Print Documents event. Your application should print the selected documents and remain open until it receives a Quit Application event from the Finder.
- If your application is open and the user elects to open or print any of your application's documents from the Finder, the Finder sends your application the Open Documents or Print Documents event.
- If your application is open and the user chooses **Restart** or **Shut Down** from the Finder's **Special** menu, the Finder sends your application the Quit Application event.

The following is a summary of the contents of the required Apple events sent by the Finder and the actions they request applications to perform:

Open Application event

Attributes

Event Class:	`kCoreEventClass`
Event ID:	`kAEOpenApplication`
Parameters:	None.
Requested Action:	Perform tasks your application normally performs when a user opens your application without opening or printing any documents, such as opening an untitled document window.

Open Documents event

Attributes

Event Class:	`kCoreEventClass`
Event ID:	`kAEOpenDocuments`

Required parameters

Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be opened.
Requested Action:	Open the documents specified in the `keyDirectObject` parameter.

Print Documents event

Attributes

Event Class:	`kCoreEventClass`
Event ID:	`kAEPrintDocuments`

Required parameters

Keyword:	`keyDirectObject`
Descriptor type:	`typeAEList`
Data:	A list of alias records for the documents to be printed.
Requested action:	Print the documents specified in the `keyDirectObject` parameter, opening windows for the documents only if your application can interact with the user.

Quit Application event

Attributes

Event Class:	`kCoreEventClass`
Event ID:	`kAEQuitApplication`
Parameters:	None
Requested Action:	Perform any tasks that your application would normally perform when the user chooses **Quit** from the application's **File** menu. (Such tasks typically include releasing memory and requesting the user to save documents which have been changed.)

Your application needs to recognise two descriptor types to handle the required Apple events: descriptor lists and alias records.

As previously stated, in the event of an Open Documents or Print Documents event, you can retrieve the data which specifies the document as an alias record, or you can request that the Apple Event Manager coerce the alias record to a file system specification record. The file system specification provides a standard method of identifying files in System 7 and later versions.

Main Apple Event Manager Constants, Data Types, and Routines Relevant to Required Apple Events

Constants

High Level Event

```
kHighLevelEvent       = 23
```

Event Class for Required Apple Event

```
kCoreEventClass       = 'aevt'   Event class for required Apple events.
```

Event IDs for Required Apple Events

```
kAEOpenApplication    = 'oapp'   Event ID for Open Application event.
kAEOpenDocuments      = 'odoc'   Event ID for Open Documents event.
kAEPrintDocuments     = 'pdoc'   Event ID for Print Documents event.
kAEQuitApplication    = 'quit'   Event ID for Quit Application event.
```

Keywords for Apple Event Attributes

```
keyMissedKeywordAttr  = 'miss'   First required parameter remaining in an Apple event.
```

Keywords for Apple Event Parameters

```
keyDirectObject       = '----'   Direct parameter
```

Apple Event Descriptor Types

```
typeAEList            = 'list'   List of descriptor records.
typeWildCard          = '****'   Matches any type.
typeFSS               = 'fss '   File system specification.
```

Result Codes

```
errAEDescNotFound     = -1701    Descriptor record was not found.
errAEParamMissed      = -1715    Handler cannot understand a parameter the client considers
                                 isrequired.
```

Data Types

```
AEEventClass = FourCharCode;          {Event class for a high level event.}
AEEventID = FourCharCode;             {Event ID for a high level event.}
AEKeyword = FourCharCode;             {Keyword for a descriptor record.}
DescType = ResType;                   {Descriptor type.}

AEDescList = AEDesc;                  {List of descriptor records.}
AERecord = AEDescList;                {List of keyword-specified descriptor records.}
AppleEvent = AERecord;                {List of attributes and parameters for Apple event.}

AEEventHandlerUPP = UniversalProcPtr; {UPP to an Apple event handler.}
```

Descriptor Record

```
AEDesc = record
  descriptorType : DescType;          {Type of data being passed.}
  dataHandle : Handle;                {Handle to data being passed.}
  end;struct AEDesc
```

Keyword-Specified Descriptor Record

```
AEKeyDesc = record
  descKey : AEKeyword;                {Keyword}
  descContent : AEDesc;               {Descriptor record.}
  end;
```

Routines

Creating and Managing Apple Event Dispatch tables

```
function  AEInstallEventHandler(theAEEventClass: AEEventClass; theAEEventID: AEEventID;
          handler: AEEventHandlerUPP; handlerRefcon: longint; isSysHandler: boolean): OSErr;
```

Dispatching Apple Events

```
function  AEProcessAppleEvent(var theEventRecord: EventRecord): OSErr;
```

Getting Data or Descriptor Records Out of Apple Event Parameters and Attributes

```
function  AEGetParamDesc(var theAppleEvent: AppleEvent; theAEKeyword: AEKeyword; desiredType:
          DescType; var result: AEDesc): OSErr;
function  AEGetAttributePtr(var theAppleEvent: AppleEvent; theAEKeyword: AEKeyword;
          desiredType: DescType; var typeCode: DescType; dataPtr: UNIV Ptr;
          maximumSize: Size; var actualSize: Size): OSErr;
```

Counting the Items in Descriptor Lists

```
function  AECountItems(var theAEDescList: AEDescList; var theCount: longint): OSErr;
```

Getting Items From Descriptor Lists

```
function  AEGetNthPtr(var theAEDescList: AEDescList; index: longint; desiredType: DescType;
          var theAEKeyword: AEKeyword; var typeCode: DescType; dataPtr: UNIV Ptr;
          maximumSize: Size; var actualSize: Size): OSErr;
```

Deallocating Memory for Descriptor Records

```
function  AEDisposeDesc(var theAEDesc: AEDesc): OSErr;
```

Demonstration Program

```
{ ############################################################################################
// AppleEventsPascal.p
// ############################################################################################
//
//
// This program:
//
// •  Installs handlers for the required Apple events.
//
// •  Responds to the receipt of required Apple events by displaying descriptive text in
//    a window opened for that purpose, and by opening simulated document windows as
//    appropriate.  These responses result from the user:
//
//    •   Double clicking on the application's icon, or selecting the icon and choosing
//        Open from the Finder's File menu, thus causing the receipt of an Open
//        Application event.
//
//    •   Double clicking on one of the document icons, selecting one or both of the
//        document icons and choosing Open from the Finder's File menu, or dragging one
//        or both of the document icons onto the application's icon, thus causing the
//        receipt of an Open Documents event.
//
//    •   Selecting one or both of the document icons and choosing Print from the
//        Finder's file menu, thus causing the receipt of a Print Documents event and,
//        if the application was not already running, a subsequent Quit Application event.
//
//    •   While the application is running, choosing Shut Down or Restart from the
//        Finder's Special menu, thus causing the receipt of a Quit Application event.
//
// The program, which is intended to be run as a built application and not from within
// CodeWarrior, utilises the following resources:
//
// •  'WIND' resources (purgeable) (initially visible) for the descriptive text display
//    window and simulated document windows.
//
// •  'MBAR' and 'MENU' resources (preload, non-purgeable).
//
// •  An 'ALRT' resource, and associated 'DITL' and 'STR#' resources, for displaying
//    error messages (purgeable).
//
// •  'ICN#', 'ics#', 'ics4', 'ics8', 'icl4', and 'icl8' resources (that is, an icon
//    family) for the application and for the application's documents. (Purgeable.)
//
// •  'FREF' resources for the application and the application's 'TEXT' documents, which
//    link the icons with the file types they represent, and which allow users to launch
//    the application by dragging the document icons to the application icon.  (Non-
//    purgeable.)
//
// •  The application's signature resource (non-purgeable), which enables the Finder to
//    identify and start up the application when the user double clicks the application's
//    document icons.
//
// •  A 'BNDL' resource (non-purgeable), which groups together the application's
//    signature, icon and 'FREF' resources.
//
// •  An 'hfdr' resource (purgeable), which provides the customised help balloon for the
//    application icon.
//
// •  A 'vers' resource (purgeable), which allows users to ascertain the version number
//    of the application.
//
// •  A 'SIZE' resource with the isHighLevelEventAware,  acceptSuspendResumeEvents, and
//    and is32BitCompatible flags set.
//
// ############################################################################################ }

program AppleEventsPascal(input, output);

```

```
{ ……… include the following Universal Interfaces }

uses

  Windows, Fonts, Menus, TextEdit, Quickdraw, Dialogs, QuickdrawText, Processes, Types,
  Memory, Events, TextUtils, ToolUtils, OSUtils, Devices, AppleEvents, Errors, Files,
  EPPC, Segload;

{ ……… define the following constants }

const

mApple = 128;
mFile = 129;
  iQuit = 11;
rMenubar = 128;
rDisplayWindow = 128;
rDocWindow = 129;
rErrorAlert = 128;
rErrorStrings = 128;
eInstallHandler = 1;
eGetRequiredParam = 2;
eGetDescriptorRecord = 3;
eMissedRequiredParam = 4;
eCannotOpenFile = 5;
eCannotPrintFile = 6;
eCannotOpenWindow = 7;
eMenus = 8;

{ ……… global variables }

var

gDone : boolean;
gWindowPtr : WindowPtr;
gWindowPtrs : array [0..10] of WindowPtr;
gNumberOfWindows : integer;
gApplicationWasOpen : boolean;
eventRec : EventRecord;
menubarHdl : Handle;
menuHdl : MenuHandle;

{ ############################################################################ DoInitManagers }

procedure DoInitManagers;

  begin
  MaxApplZone;
  MoreMasters;

  InitGraf(@qd.thePort);
  InitFonts;
  InitWindows;
  InitMenus;
  TEInit;
  InitDialogs(nil);

  InitCursor;
  FlushEvents(everyEvent, 0);
  end;
    {of procedure DoInitManagers}

{ ############################################################################ DoAppleMenu }

procedure DoAppleMenu(menuItem : integer);

  var
  itemName : string;
  daDriverRefNum : integer;

  begin
  GetMenuItemText(GetMenuHandle(mApple), menuItem, itemName);
  daDriverRefNum := OpenDeskAcc(itemName);
  end;
    {of procedure DoAppleMenu}

{ ############################################################################ DoFileMenu }
```

```
146
147 procedure DoFileMenu(menuItem : integer);
148
149   begin
150   if (menuItem  = iQuit) then
151     gDone := true;
152   end;
153     {of procedure DoFileMenu}
154
155 { ############################################################################ DoMenuChoice }
156
157 procedure DoMenuChoice(menuChoice : longint);
158
159   var
160   menuID, menuItem : integer;
161
162   begin
163   menuID := HiWord(menuChoice);
164   menuItem := LoWord(menuChoice);
165
166   if (menuID = 0) then
167     Exit(DoMenuChoice);
168
169   case (menuID) of
170
171     mApple:
172       begin
173       DoAppleMenu(menuItem);
174       end;
175
176     mFile:
177       begin
178       DoFileMenu(menuItem);
179       end;
180
181     end;
182       {of case statement}
183
184   HiliteMenu(0);
185   end;
186     {of procedure DoMenuChoice}
187
188 { ################################################################################ DoError }
189
190 procedure DoError(errorType : integer);
191
192   var
193   errorString : string;
194   ignored : OSErr;
195
196   begin
197   SetCursor(qd.arrow);
198
199   GetIndString(errorString, rErrorStrings, errorType);
200   ParamText(errorString, '', '', '');
201   if (errorType < 7)
202     thenignored := CautionAlert(rErrorAlert, nil)
203
204     elsebegin
205       ignored := StopAlert(rErrorAlert, nil);
206       ExitToShell;
207       end;
208   end;
209     {of procedure DoError}
210
211 { ################################################################ HasGotRequiredParams }
212
213 function HasGotRequiredParams(appEvent : AppleEvent) : OSErr;
214
215   var
216   returnedType : DescType;
217   actualSize : Size;
218   theErr : OSErr;
219
220   begin
221   theErr := AEGetAttributePtr(appEvent, keyMissedKeywordAttr, typeWildCard, returnedType,
222                           nil, 0, actualSize);
```

```
223
224     if (theErr = errAEDescNotFound)
225       thenHasGotRequiredParams := noErr
226       elseif (theErr = noErr)
227         thenHasGotRequiredParams := errAEParamMissed
228         elseHasGotRequiredParams := noErr;
229     end;
230       {of function HasGotRequiredParams}
231
232   { ############################################################################ DrawTextString }
233
234   procedure DrawTextString(eventString : string);
235
236     var
237     tempRegion : RgnHandle;
238
239     begin
240     tempRegion := NewRgn;
241
242     ScrollRect(gWindowPtr^.portRect, 0, -15, tempRegion);
243     DisposeRgn(tempRegion);
244
245     MoveTo(8, 176);
246     DrawString(eventString);
247     end;
248       {of procedure DrawTextString}
249
250   { ############################################################################ DoPrepareToTerminate }
251
252   procedure DoPrepareToTerminate;
253
254     var
255     finalTicks : UInt32;
256
257     begin
258     DrawTextString('Received an Apple event: QUIT APPLICATION');
259
260     if (gApplicationWasOpen)
261       thenbegin
262         DrawTextString('   • I would now ask the user to save any unsaved files');
263         DrawTextString('     before terminating myself in reponse to the event.');
264         DrawTextString('   • Click the mouse when ready to terminate.');
265
266         while not (Button) do;
267         end
268
269       elsebegin
270         DrawTextString('    • Terminating myself in response');
271         Delay(300, finalTicks);
272         end;
273
274     { If the user did not click the Cancel button in a Save dialog box: }
275
276     gDone := true;
277     end;
278       {of procedure DoPrepareToTerminate}
279
280   { ############################################################################ DoNewWindow }
281
282   function DoNewWindow : WindowPtr;
283
284     begin
285     gWindowPtrs[gNumberOfWindows] := GetNewWindow(rDocWindow, nil, WindowPtr(-1));
286     if (gWindowPtrs[gNumberOfWindows] = nil) then
287       DoError(eCannotOpenWindow);
288
289     gNumberOfWindows := gNumberOfWindows + 1;
290
291     DoNewWindow := gWindowPtrs[gNumberOfWindows - 1];
292     end;
293       {of function DoNewWindow}
294
295   { ############################################################################ DoOpenFile }
296
297   function DoOpenFile(var fileSpec : FSSpec; index, numberOfItems : longint) : boolean;
298
299     var
```

```
  myWindowPtr : WindowPtr;
  finalTicks : UInt32;

  begin
  gApplicationWasOpen := true;

  if (index = 1) then
    DrawTextString('Received an Apple event: OPEN DOCUMENTS.');

  if (numberOfItems = 1)
    thenbegin
      DrawTextString('    • The file to open is: ');
      DrawString(fileSpec.name);
      DrawTextString('    • Opening titled window in reponse.');
      Delay(100, finalTicks);
      end

    elsebegin
      if(index = 1)
        thenbegin
          DrawTextString('    • The files to open are: ');
          DrawString(fileSpec.name);
          end

        elsebegin
          DrawString('p and ');
          DrawString(fileSpec.name);
          DrawTextString('    • Opening titled windows in reponse.');
          Delay(100, finalTicks);
          end;
      end;

  myWindowPtr := DoNewWindow;
  if (myWindowPtr <> nil)
    thenbegin
      SetWTitle(myWindowPtr, fileSpec.name);
      DoOpenFile := true;
      end

    elseDoOpenFile := false;

  end;
    {of function DoOpenFile}

{ ############################################################################## DoPrintFile }

function DoPrintFile(var fileSpec : FSSpec; index, numberOfItems : longint) : boolean;

  var
  myWindowPtr : WindowPtr;
  finalTicks : UInt32;

  begin
  if (index = 1) then
    DrawTextString('Received an Apple event: PRINT DOCUMENTS');

  if (numberOfItems = 1)
    then  begin
    DrawTextString('    • The file to print is: ');
    DrawString(fileSpec.name);
    myWindowPtr := DoNewWindow;
    SetWTitle(myWindowPtr, fileSpec.name);
    DrawTextString('    • I would present the Print dialog box first and then');
    DrawTextString('      print the document when the user has made his settings.');
    Delay(100, finalTicks);
    DrawTextString('    • Assume that I am now printing the document.');
    end

    elsebegin
    if (index = 1)
      thenbegin
      DrawTextString('    • The first file to print is: ');
      DrawString(fileSpec.name);
      DrawTextString('      I would present the Print dialog box for the first file');
      DrawTextString('      only and use the users settings to print both files.');
      end
```

```
        elsebegin
        Delay(200, finalTicks);
        DrawTextString('    • The second file to print is: ');
        DrawString(fileSpec.name);
        DrawTextString('      I am using the Print dialog box settings used for the');
        DrawTextString('      first file.');
        end;

      myWindowPtr := DoNewWindow;
      SetWTitle(myWindowPtr, fileSpec.name);
      Delay(200, finalTicks);
      DrawTextString('    • Assume that I am now printing the document.');
      Delay(200, finalTicks);
      end;

    if (numberOfItems = index) then
      begin
      if not (gApplicationWasOpen)
        thenbegin
        DrawTextString('      Since the application was not already open, I expect to');
        DrawTextString('      receive a QUIT APPLICATION event when I have finished.');
        end

        elsebegin
        DrawTextString('      Since the application was already open, I do NOT expect');
        DrawTextString('      to receive a QUIT APPLICATION event when I have finished.');
        end;

      Delay(500, finalTicks);
      DrawTextString('    • Finished print job.');
      end;

    DisposeWindow(myWindowPtr);
    DoPrintFile := true;
    end;
      {of function DoPrintFiles}

  { ############################################################################ DoOpenAppEvent }

  function DoOpenAppEvent(var appEvent, reply : AppleEvent; handlerRefCon : longint)
                          : OSErr;
    var
    theErr : OSErr;
    myWindowPtr : WindowPtr;
    finalTicks : UInt32;

    begin
    gApplicationWasOpen := true;

    theErr := HasGotRequiredParams(appEvent);

    if (theErr = noErr)
      thenbegin
        DrawTextString('Received an Apple event: OPEN APPLICATION.');
        DrawTextString('    • Opening an untitled window in reponse.');
        Delay(100, finalTicks);

        myWindowPtr := DoNewWindow;
        SetWTitle(myWindowPtr, 'Untitled 1');

        DoOpenAppEvent := noErr;
        end

      elseDoOpenAppEvent := theErr;
    end;
      {of function DoOpenAppEvent}

  { ############################################################################ DoOpenDocsEvent }

  function DoOpenDocsEvent(var appEvent, reply : AppleEvent; handlerRefCon : longint)
                          : OSErr;
    var
    fileSpec : FSSpec;
    docList : AEDescList;
    theErr, ignoreErr : OSErr;
    index, numberOfItems : SInt32;
    actualSize : Size;
```

```
  keyWord : AEKeyword;
  returnedType : DescType;
  result : boolean;

  begin
  theErr := AEGetParamDesc(appEvent, keyDirectObject, typeAEList, docList);

  if (theErr = noErr)
    then  begin
      theErr := HasGotRequiredParams(appEvent);
      if (theErr = noErr)
        then  begin
        ignoreErr := AECountItems(docList, numberOfItems);
        if (theErr = noErr) then
          begin
          for index := 1 to numberOfItems do
            begin
            theErr := AEGetNthPtr(docList, index, typeFSS, keyWord, returnedType,
                      @fileSpec, sizeof(fileSpec), actualSize);
            if (theErr = noErr)
              then  begin
                result := DoOpenFile(fileSpec, index, numberOfItems);
                if (result = false) then
                  DoError(eCannotOpenFile);
                  end
                else  DoError(eGetDescriptorRecord);
            end;
              {of for loop}
          end;
        end
        else  DoError(eMissedRequiredParam);

      ignoreErr := AEDisposeDesc(docList);
      end

    else  DoError(eGetRequiredParam);

  DoOpenDocsEvent := theErr;
  end;
    {of DoOpenDocsEvent}

{ ############################################################################ DoPrintDocsEvent }

function DoPrintDocsEvent(var appEvent, reply : AppleEvent; handlerRefCon : longint)
                        : OSErr;
  var
  fileSpec : FSSpec;
  docList : AEDescList;
  theErr, ignoreErr : OSErr;
  index, numberOfItems : longint;
  actualSize : Size;
  keyWord : AEKeyword;
  returnedType : DescType;
  result : boolean;

  begin
  theErr := AEGetParamDesc(appEvent, keyDirectObject, typeAEList, docList);

  if (theErr = noErr)
    then  begin
      theErr := HasGotRequiredParams(appEvent);
      if (theErr = noErr)
        then  begin
          ignoreErr := AECountItems(docList, numberOfItems);
          if (theErr = noErr) then
            begin
            for index := 1 to numberOfItems do
            begin
            theErr := AEGetNthPtr(docList, index, typeFSS, keyWord, returnedType,
            @fileSpec, sizeof(fileSpec), actualSize);
            if (theErr = noErr)
              then  begin
                result := DoPrintFile(fileSpec, index, numberOfItems);
                if (result = false) then
                  DoError(eCannotPrintFile);
                end
              else  DoError(eGetDescriptorRecord);
```

```
            end;
              {of for loop}
            end;
          end

        else  DoError(eMissedRequiredParam);

      ignoreErr := AEDisposeDesc(docList);
      end

    else  DoError(eGetRequiredParam);

  DoPrintDocsEvent := theErr;
  end;
    {of function DoPrintDocsEvent}

{ ############################################################################ DoQuitAppEvent }

function DoQuitAppEvent(var appEvent, reply : AppleEvent; handlerRefCon : longint)
                        : OSErr;
  var
  theErr : OSErr;

  begin
  theErr := HasGotRequiredParams(appEvent);

  if (theErr = noErr)
    thenbegin
      DoPrepareToTerminate;
      DoQuitAppEvent := theErr;
      end

    elseDoQuitAppEvent := theErr;
  end;
    {of function DoQuitAppEvent}

{ ############################################################################ DoInstallAEHandlers }

procedure DoInstallAEHandlers;

  var
  err : OSErr;

  begin
  err := AEInstallEventHandler(kCoreEventClass, kAEOpenApplication,
                  AEEventHandlerUPP(@DoOpenAppEvent), 0, false);

  if (err <> noErr) then
     DoError(eInstallHandler);

  err := AEInstallEventHandler(kCoreEventClass, kAEOpenDocuments,
                  AEEventHandlerUPP(@DoOpenDocsEvent), 0, false);

  if (err <> noErr) then
     DoError(eInstallHandler);

  err := AEInstallEventHandler(kCoreEventClass, kAEPrintDocuments,
                  AEEventHandlerUPP(@DoPrintDocsEvent), 0, false);

  if (err <> noErr) then
    DoError(eInstallHandler);

  err := AEInstallEventHandler(kCoreEventClass, kAEQuitApplication,
                  AEEventHandlerUPP(@DoQuitAppEvent), 0, false);
  if (err <> noErr) then
     DoError(eInstallHandler);
  end;
    {of procedure DoInstallAEHandlers}

{ ############################################################################ DoMouseDown }

procedure DoMouseDown(eventRec : EventRecord);

  var
  myWindowPtr : WindowPtr;
  partCode : integer;
  menuChoice : longint;
```

```
608
609     begin
610     partCode := FindWindow(eventRec.where, myWindowPtr);
611
612     case (partCode) of
613
614       inMenuBar:
615         begin
616         menuChoice := MenuSelect(eventRec.where);
617         DoMenuChoice(menuChoice);
618         end;
619
620       inSysWindow:
621         begin
622         SystemClick(eventRec, myWindowPtr);
623         end;
624
625       inDrag:
626         begin
627         DragWindow(myWindowPtr, eventRec.where, qd.screenBits.bounds);
628         end;
629       end;
630         {of case statement}
631     end;
632       {of procedure DoMouseDown}
633
634   { ############################################################################## DoEvents }
635
636   procedure DoEvents(eventRec : EventRecord);
637
638     var
639     charCode : char;
640     ignored : OSErr;
641
642     begin
643     case (eventRec.what) of
644
645       kHighLevelEvent:
646         begin
647         ignored := AEProcessAppleEvent(eventRec);
648         end;
649
650       mouseDown:
651         begin
652         DoMouseDown(eventRec);
653         end;
654
655       keyDown, autoKey:
656         begin
657         charCode := chr(BAnd(eventRec.message, charCodeMask));
658         if (BAnd(eventRec.modifiers, cmdKey) <> 0) then
659           DoMenuChoice(MenuKey(charCode));
660         end;
661
662       updateEvt:
663         begin
664         BeginUpdate(WindowPtr(eventRec.message));
665         EndUpdate(WindowPtr(eventRec.message));
666         end;
667
668       osEvt:
669         begin
670         HiliteMenu(0);
671         end;
672       end;
673         {of case statement}
674     end;
675       {of procedure DoEvents}
676
677   { ################################################################## start of main program }
678
679   begin
680
681     gNumberOfWindows := 0;
682     gApplicationWasOpen := false;
683
684     { …… initialize managers }
```

```
685
686     DoInitManagers;
687
688     { ……… open a window }
689
690     gWindowPtr := GetNewWindow(rDisplayWindow, nil, WindowPtr(-1));
691     if (gWindowPtr = nil) then
692       begin
693       DoError(eCannotOpenWindow);
694       ExitToShell;
695       end;
696
697     SetPort(gWindowPtr);
698     TextSize(10);
699
700     { ……… set up menu bar and menus }
701
702     menubarHdl := GetNewMBar(rMenubar);
703     if (menubarHdl = nil) then
704        DoError(eMenus);
705     SetMenuBar(menubarHdl);
706     DrawMenuBar;
707
708     menuHdl := GetMenuHandle(mApple);
709     if (menuHdl = nil)
710       then DoError(eMenus)
711       else AppendResMenu(menuHdl,'DRVR');
712
713     { ……… install Apple event handlers }
714
715     DoInstallAEHandlers;
716
717     { ……… event loop }
718
719     gDone := false;
720
721     while not (gDone) do
722       if (WaitNextEvent(everyEvent, eventRec, 180, nil)) then
723         DoEvents(eventRec);
724
725   end.
726
727   { ########################################################################################## }
```

Demonstration Program Comments

This demonstration is not intended to be run from within CodeWarrior. Accordingly, a built application titled AppleEvents is provided. The built application, together with two simulated 'TEXT' documents (Document A and Document B) which have the AppleEvents application as their creator, are located in the chap08pascal_demo folder.

The demonstration requires that the user open the window containing the AppleEvents application in order to access the Apple Events application icon and two document icons.

Using all of the methods available in the Finder (that is, double clicking the icons, dragging document icons to the application icon, selecting the icons and choosing Open and Print from the Finder's File menu) the user should launch the application, open the simulated documents and print the documents, noting the descriptive text printed in the non-document window in response to the receipt of the resulting Apple events. The user should also choose Restart or Shut Down from the Finder's Special menu while the application is running, also noting the displayed text resulting from receipt of the Quit Application event. Opening and printing should be attempted when the application is already running and when the application is not running.

Although not related to the required Apple events aspects of the program, the following aspects of the demonstration may also be investigated:

- The help balloon for the application icon. (The 'hfdr' resource refers.)
- The version information for the application in the Finder's Get Info… window. (The 'vers' resource refers.)

The constant declaration block

Lines 81-96 establish constants relating to menu, alert box, error message string, and window resources, menus IDs and menu item numbers.

The variable declaration block

gDone controls program termination. gWindowPtr will be assigned the pointer to the text display window. The gWindowPtrs[] array will be assigned pointers to the document windows. gNumberOfWindows is used to increment the gWindowPtrs[] array element after each document window is created.

gApplicationWasOpen is used to control the manner of program termination when a Quit Application event is received, depending on whether the event followed a Print Documents event or resulted from the user choosing Restart of Shut Down from the Finder's Special menu.

The procedures DoAppleMenu, DoFileMenu, DoMenuChoice, and DoError

DoAppleMenu, DoFileMenu and DoMenuChoice handle menu selections. gDone is set to true when the user selects Quit from the application's File menu (Line 151).

DoError handles errors, displaying an alert box with descriptive text and, where necessary terminating the program.

The function HasGotRequiredParams

HasGotRequiredParams is the application-defined function called by DoOpenAppEvent to confirm that the event passed to it contains no required parameters, and by the other handlers to check that they have received all the required parameters.

The first parameter in the call to AEGetAttributePtr (Line 221) is a pointer to the Apple event in question. The second parameter is the Apple event keyword; in this case the constant keyMissedKeywordAttr is specified, meaning the first required parameter remaining in the event. The third parameter specifies the descriptor type; in this case the constant typeWildCard is specified, meaning any descriptor type. The fourth parameter receives the descriptor type of the returned data. The fifth parameter is a pointer to the data buffer which stores the returned data. The sixth parameter is the maximum length of the data buffer to be returned. Since we do not need the data itself, these parameters are set to nil and 0 respectively. The last parameter receives the actual length, in bytes, of the data buffer for the attribute.

AEGetAttributePtr returns the result code errAEDescNotFound if the specified descriptor type (typeWildCard, that is, any descriptor type) is not found, meaning that the handler extracted all the required parameters. In this case, HasGotRequiredParams returns noErr (Line 225).

If AEGetAttributePtr returns noErr (Line 226), the handler has not extracted all of the required parameters, in which case, the handler should return errAEParamMissed and not handle the event. Accordingly, errAEParamMissed is returned to the handler (and, in turn, by the handler) if noErr is returned by AEGetAttributePtr .

The procedure DrawTextString

DrawTextString draws scrolling explanatory text in the text window as each event is received.

The procedure DoPrepareToTerminate

DoPrepareToTerminate is the procedure called by the Quit Application event handler. In this demonstration, gDone will be set to true (Line 276), and the program will thus terminate immediately, if the Quit Application event resulted from the user initiating a print operation from the Finder when the application was not running.

If the application was running (Line 260) and the Quit Application event thus arose from the user selecting Restart or Shut Down from the Finder's File menu, the demonstration waits for a button click (Line 266) before setting gDone to true. (In a real application, and where appropriate, this area of the code would invoke dialog boxes to ascertain whether the user wished to save any changed documents before closing down.)

Note that, when your application is ready to quit, it must call ExitToShell from the main event loop, not from the handlers for the Quit Application event. Your application should quit only after the handler returns noErr as its function result.

The function DoNewWindow

DoNewWindow opens document windows in response to calls from the Open Application and Open Documents event handlers.

The procedure DoOpenFile

DoOpenFile takes the file system specification record and opens a window with the filename contained in that record repeated in the window's title bar (Lines 332-337). (The rest of the DoOpenFile code is related to drawing explanatory text in the text window.)

In a real application, this is the function which should open files as a result of, firstly, the receipt of the Open Documents event and, secondly, the user choosing Open from the application's File menu and then choosing a file or files from the resulting Open dialog box.

The procedure DoPrintFile

DoPrintFile is the procedure which, in a real application, would take the file system specification record passed to it from the Print Documents event handler, extract the filename and control the printing of that file. (In this demonstration, all of the DoPrintFile code is related to drawing explanatory text in the text window.)

If your application can interact with the user, it should open windows for the documents, display a print Job dialog for the first document, and use the settings entered by the user for the first document to print all documents.

Note that, if your application was not running when the user selected a document icon and chose Print from the Finder's File menu, the Finder will send a Quit Application event following the print operation.

The function DoOpenAppEvent

DoOpenAppEvent is the handler for the Open Application event.

At line 424, the global variable gApplicationWasOpen, which controls the manner of program termination when a Quit Application event is received, is set to true. (This line is required for demonstration program purposes only.)

Line 426 calls the application-defined function HasGotRequiredParams to check whether the Apple event contains any required parameters. If so, the handler returns an error because, by definition, the Open Application event should not contain any required parameters.

If noErr is returned by HasGotRequiredParams, the handler does what the user expects the application to do when it is opened, that is, it opens an untitled document window (Lines 434-435). The handler then returns noErr (Line 437).

If errAEParamMissed is returned by HasGotRequiredParams, this is returned by the handler (Line 440)

Lines 430-431 simply print some text in the text window for demonstration program purposes.

The function DoOpenDocsEvent

DoOpenDocsEvent is the handler for the Open Documents event.

At line 459, AEGetParamDesc is called to get the direct parameter (specified in the keyDirectObject keyword) out of the Apple event. The constant typeAEList specifies the descriptor type as a list of descriptor records. The descriptor list is received by the docList variable.

Before proceeding further, the handler checks that it has received all the required parameters by calling the application-defined function HasGotRequiredParams (Line 463).

Having retrieved the descriptor list from the Apple event, the handler calls AECountItems to count the number of descriptors in the list (Line 466).

Using the returned number as an index, AEGetNthPtr is called (Line 471) to get the data of each descriptor record in the list. In the AEGetNthPtr call, the parameter typeFSS specifies the desired type of the resulting data, causing the Apple Event Manager to coerce the data in the descriptor record to a file system specification record. Note also that keyWord receives the keyword of the specified descriptor record, returnedType receives the descriptor type, fileSpec receives a pointer to the file system specification record, sizeof(fileSpec) establishes the length, in bytes, of the data returned, and actualSize receives the actual length, in bytes, of the data for the descriptor record.

After extracting the file system specification record describing the document to open, the handler calls the application-defined function for opening files (Line 475). (In a real application, that function would typically be the same as that invoked when the user chooses Open from the application's File menu.)

If the call to AEGetNthPtr does not return noErr, Line 479 calls the application's error handler. (AEGetNthPtr will return an error code if there was insufficient room in the heap, the data could not be coerced, the descriptor record was not found, the descriptor was of the wrong type or the descriptor record was not a valid descriptor record.)

If the call to HasGotRequiredParams does not return noErr, Line 484 calls the application's error handler. (HasGotRequiredParams returns noErr only if you got all the required parameters.)

At Line 486, and since the handler has no further requirement for the data in the descriptor list, AEDisposeDesc is called to dispose of the descriptor list.

If the call to AEGetParamDesc does not return noErr, Line 489 calls the application's error handler. (AEGetParamDesc will return an error code for much the same reasons as will AEGetNthPtr.)

The function DoPrintDocsEvent

DoPrintDocsEvent is the handler for the Print Documents event.

The code is identical to that for the Open Documents event handler DoOpenDocs except that, at Line 526, the application-defined function for printing files is called rather than the function for simply opening files.

The function DoQuitAppEvent

DoQuitAppEvent is the handler for the Quit Application event.

After checking that it has received all the required parameters by calling the application-defined function HasGotRequiredParams (Line 555), the handler calls the application-defined procedure DoPrepareToTerminate (Line 559).

The procedure DoInstallAEHandlers

DoInstallAEHandlers installs the handlers for the four required Apple events in the application's Apple event dispatch table.

The procedure DoMouseDown

DoMouseDown performs such mouse-down processing as is necessary to support the demonstration aspects of the program.

The procedure DoEvents

DoEvents branches according to the event type received.

At Line 645, the constant kHighLevelEvent (defined in EPPC) accommodates the receipt of a high-level event, in which case AEProcessAppleEvent is called. (AEProcessAppleEvent looks in the application's Apple event dispatch table for a match to the event class and event ID contained in, respectively, the event record's message and where fields, and calls the appropriate handler.)

The main program block

The main function initialises the system software managers (Line 686), opens the text display window (Line 690), makes that window the current graphics port (Line 697), sets the text size (Line 698) and sets up the menus (Lines 702-711). Note that here, and in other areas of the program, an error will cause the application-defined error-handling procedure DoError to be called.

At Line 715, the required Apple event handlers are installed before the main event loop is entered (Lines 719-723).

Creating Finder Interface Resources Using ResEdit

The following describes the creation of the icon family, the 'BNDL' resource, the 'FREF' resources, the signature resource, the 'vers' resource, and the 'hfdr' resource for the AppleEvents demonstration program using ResEdit. (As stated at Chapter 2 — LowLevel and Operating System Events, 'SIZE' resources are created automatically by CodeWarrior. As stated at Chapter 7 — Finder Interface, missing application name string resources and application missing message string resources are adressed at, respectively, Chapter — 14 Files and Chapter 15 — More on Resources.)

Preliminaries - Setting the Creator and Type in CodeWarrior

With the AppleEvents.µ project open in CodeWarrior, choose **Edit/Project Settings/** Project/68K Project, enter KJBB at the Creator item and APPL at the Type item.

Creating the Icon Family

To create the icon family resources for AppleEvents, proceed as follows.

Double-click the AppleEvents.µ.rsrc icon to start ResEdit and open the existing AppleEvents.µ.rsrc file. Choose **Resource/Create New Resource**. In the resulting dialog, select **ICN#** and click the **OK** button. The **ICN#s from AppleEvents.µ.rsrc** window opens, followed by the **Icon Family ID = 128 ...** window.

Click the icl8 box at the right and use the icon editor at left to draw a large 8-bit colour icon for the application. Then drag the thumbnail in the icl8 box at right to the other seven boxes to automatically create the remaining icons in the family, together with the masks. Choose **Resource/GetResource Info** and click the **Purgeable** checkbox in the **Info for icl8 128 ...** window. Then click the close box of this window, followed by the **Yes** button in the resulting dialog, to make all of the icon resources in the family purgeable. Finally, close the **Icon Family ID = 128 ...** window.

Choose **Resource/Create New Resource** again. The **Icon Family ID = 129 ...** window opens. Repeat the above process to create an icon family for a text document. Close the **Icon Family ID = 129 ...** window, followed by the **ICN#s from AppleEvents.µ.rsrc** window.

The **AppleEvents.µ.rsrc** window now contains icons representing the icon family resources just created.

Creating the 'BNDL', 'FREF', and Signature Resources

To create the 'BNDL', 'FREF', and signature resources for AppleEvents, proceed as follows.

Choose **Resource/Create New Resource**. In the resulting dialog, select **BNDL** and click **OK**. The **BNDLs from AppleEvents.µ.rsrc** window opens, followed by the **BNDL ID = 128 ...** window.

Choose **BNDL/Extended View**. Enter the application's signature (KJBB) in the **Signature:** item at top. Enter 1995, K. J. Bricknell at the **©String:** item. (Relate these last two entries to the example signature resource in Rez input format at Chapter 7.)

Choose **Resource/Create New File Type**. A row is added to the **FREF/Finder Icons** list. Enter APPL in the **Type** column. Click in the icon family column and then choose **BNDL/Choose Icon**. In the resulting dialog, click on icon 128, then click **OK**. The icon family appears in the icon family column and the family resource ID appears in the **resID** column.

Choose **Resource/Create New File Type**. A second row is added to the **FREF/Finder Icons** list. Repeat the previous process, except enter TEXT in the **Type** column and assign the icon family with ID 129. (Before closing the **BNDL ID = 128 ...** window, relate the rows and columns in the **BNDL ID = 128 ...** window to the example 'FREF' and 'BNDL' resources in Rez input format, and to Fig 2, at Chapter 7.)

Close the **BNDL ID = 128 ...** window. Close the **BNDLs from AppleEvents.µ.rsrc** window. Note the BNDL, FREF and KJBB icons (the latter represents the signature resource) in the **AppleEvents.µ.rsrc** window. Close the **AppleEvents.µ.rsrc** window, saving the file.

In CodeWarrior, compile/link/run AppleEvents.µ. The custom application icon will appear in the AppleEvents folder window. Open the application AppleEvents in ResEdit and choose **File/Get Info for AppleEvents**. The **Info for AppleEvents** window opens. Note that the **Has BNDL** checkbox is checked and that the **Inited** checkbox is not checked. Close the **Info for AppleEvents** window, restart the computer, and repeat the previous procedure. The **Inited** checkbox is now checked.

Sometimes, it may be necessary to rebuild the desktop database (by holding down the Option and Command keys during startup) to cause the custom application icon to appear.

As an aside, the two document files used by the AppleEvents demonstration program were created using SimpleText. Both files were opened in ResEdit, **File/Get Info for ...** was chosen, and the file's creator was changed to KJBB.

The 'vers' Resource

Double-click the AppleEvents.µ.rsrc icon to start ResEdit and open the existing AppleEvents.µ.rsrc file. The **AppleEvents.µ.rsrc** window opens.

Double-click the vers icon. The **verss from AppleEvents.µ.rsrc** window opens. A 'vers' resources with ID 1 appears in the list. Double-click that list entry. The **vers ID = 1 from AppleEvents.µ.rsrc** window opens.

The following relates the first example 'vers' resource in Rez input format in Chapter 7 — Finder Interface to the ResEdit display and interface:

`resource 'vers'`	This was established when the resource was created by choosing **Resource/Create New Resource**. A small dialog opened, the item **vers** was clicked, and the dialog's **OK** button was clicked.
`1,`	`1` is the `'vers'` resource ID. Choose **Resource/Get Resource Info**. The **Info for vers 1 ...** window opens. Note the editable text item titled **ID:**. This is where you set the `'vers'` resource ID.
`purgeable)`	While the **Info for vers 1 ...** window is open, note that the **Purgeable** checkbox is checked. Close the **Info for vers 1 ...** window.
`0x01,`	Minor revision level. Note the first editable text item against **Version number:**.
`0x00,`	Minor revision level. Note the second and third editable text items against **Version number:**.
`release,`	Development stage. Note the pop-up menu **Release:**.
`0x00,`	Prerelease revision level. Note the editable text item **Non-release:**.
`verUS,`	Region Code. Note the pop-up menu **Country Code:**.
`"1.0",`	Version number. Note the editable text item **Short version string:**.
`"1.1 (US) ...`	Version message. Note the editable text item **Long version string (visible in Get Info):**.

Close the **vers ID = 1 from ...** window. Close the **verss from AppleEvents.µ.rsrc** window.

Creating the 'hfdr' Resource

ResEdit does not support the creation of 'hfdr' resources; however, a work-around is available. To create the 'hfdr' resource for the AppleEvents demonstration, proceed as follows.

Firstly, copy a 'hfdr' resource from another application into the **AppleEvents.µ.rsrc** window and double-click the resulting hfdr icon. The **hfdrs from AppleEvents.µ.rsrc** window opens. One 'hfdr' resource appears in the list. Note that the resource ID is -5696. Click the list entry and choose **Resource/Open Using Hex Editor**. The **hfdr ID = -5696 from ...** window opens. The first three rows in the window will be similar to the following:

```
000000  0002 0000 0000 0000  ########
000008  0000 0001 0096 0001  ########
000010  7E55 7365 2074 6865  éUse the
```

In the hexadecimal display (the four columns in the centre), highlight and cut everything after 7E, leaving the following:

```
000000  0002 0000 0000 0000  ########
000008  0000 0001 0096 0001  ########
000010  7E                   é
```

In the ASCII display (the column at the right), type in the following after the é:

```
The AppleEvents application demonstrates the required Apple events (Open
Application, Open Documents, Print Documents and Quit Application).
```

There are 141 characters in this text (8D in hexadecimal). Accordingly, in the hexadecimal display, change 7E to 8D. The first three rows in the window should appear as follows:

```
000000  0002 0000 0000 0000  ########
000008  0000 0001 0096 0001  ########
000010  8D54 6865 2041 7070  éThe App
```

Close the **hfdr ID = -5696 from ...** window. Close the **hfdrs from AppleEvents.µ.rsrc** window. Close the **AppleEvents.µ.rsrc** window, saving the file.