This chapter describes how your application can use the Apple Event Manager to create and send Apple events. If you want to factor your application for recording, or if you want your application to send Apple events directly to other applications, you can use Apple Event Manager routines to create and send Apple events.

Before you read this chapter, you should be familiar with the chapters "Introduction to Interapplication Communication," "Introduction to Apple Events," and "Responding to Apple Events" in this book. If you are factoring your application, you should also be familiar with the chapter "Recording Apple Events" in this book.

This chapter provides the basic information you need to create and send Apple events from your application. To send core and functional-area Apple events, your application must also be able to create object specifier records. For information about object specifier records, see the chapter "Resolving and Creating Object Specifier Records" in this book.

To allow your application to send Apple events to applications on other computers, you may wish to use the `PPCBrowser` function, which is described in the chapter "Program-to-Program Communications Toolbox" in this book.

The first section in this chapter, "Creating an Apple Event," describes how to

- create an Apple event
- add parameters to an Apple event
- specify optional Apple event parameters
- specify a target address

The section "Sending an Apple Event" describes how to

- send an Apple event
- deal with timeouts
- write an idle function
- write a reply filter function

Creating an Apple Event

You create an Apple event by using the `AECreateAppleEvent` function. You supply parameters that specify the event class and event ID, the target address, the return ID, and the transaction ID, and a buffer for the returned Apple event. The `AECreateAppleEvent` function creates and returns, in the buffer you specify, an Apple event with the attributes set as your application requested. You should not directly manipulate the contents of the Apple event; rather, use Apple Event Manager functions to add additional attributes or parameters to it.

The example that follows creates an imaginary Multiply event using the `AECreateAppleEvent` function.

```
myErr := AECreateAppleEvent(kArithmeticClass, kMultEventID,
                            targetAddress, kAutoGenerateReturnID,
                            kAnyTransactionID, theAppleEvent);
```

The event class, specified by the `kArithmeticClass` constant, identifies this event as belonging to a class of Apple events for arithmetic operations. The event ID specifies the particular Apple event within the class—in this case, an Apple event that performs multiplication.

You specify the target of the Apple event in the third parameter to `AECreateAppleEvent`. The target address can identify an application on the local computer or another computer on the network. You can specify the address using a target ID record or session reference number. For processes on the local computer, you can also use a process serial number or application signature to specify the address. See "Specifying a Target Address" on page 5-10 for more information.

In the fourth parameter, you specify the return ID of the Apple event, which associates this Apple event with the server's reply. The `AECreateAppleEvent` function assigns the specified return ID value to the `keyReturnIDAttr` attribute of the Apple event. If a server returns a standard reply Apple event (that is, an event of event class `'aevt'` and event ID `'ansr'`) in response to this event, the Apple Event Manager assigns the reply event the same return ID. When you receive a reply Apple event, you can check the `keyReturnIDAttr` attribute to determine which outstanding Apple event the reply is responding to. You can use the `kAutoGenerateReturnID` constant to request that the Apple Event Manager generate a return ID that is unique to this session for the Apple event. Otherwise, you are responsible for making it unique.

The fifth parameter specifies the transaction ID attribute of the Apple event. A ***transaction*** is a sequence of Apple events that are sent back and forth between the client and server applications, beginning with the client's initial request for a service. All Apple events that are part of one transaction must have the same transaction ID.

You can use a transaction ID to indicate that an Apple event is one of a sequence of Apple events related to a single transaction. The `kAnyTransactionID` constant indicates that the Apple event is not part of a transaction.

The `AECreateAppleEvent` function creates an Apple event with only the specified attributes and no parameters. To add parameters or additional attributes, you can use other Apple Event Manager functions.

Adding Parameters to an Apple Event

You can use the `AEPutParamPtr` or `AEPutParamDesc` function to add parameters to an Apple event. When you use either of these functions, the Apple Event Manager adds the specified parameter to the Apple event.

Use the `AEPutParamPtr` function when you want to add data specified in a buffer as the parameter of an Apple event. You specify the Apple event, the keyword of the parameter to add, the descriptor type, a buffer that contains the data, and the size of this buffer as parameters to the `AEPutParamPtr` function. The `AEPutParamPtr` function adds the data to the Apple event as a parameter with the specified keyword.

For example, this code adds a parameter to the Multiply event using the `AEPutParamPtr` function:

```
CONST keyOperand1 = 'OPN1';
VAR
   number1:          LongInt;
   theAppleEvent:    AppleEvent;
   myErr:            OSErr;

number1 := 10;
myErr := AEPutParamPtr(theAppleEvent, keyOperand1,
                       typeLongInteger, @number1,
                       SizeOf(number1));
```

In this example, the Apple Event Manager adds the parameter containing the first number to the specified Apple event.

Use the `AEPutParamDesc` function to add a descriptor record to an Apple event. The descriptor record you specify must already exist. To create or get a descriptor record, you can use the `AECreateDesc`, `AEDuplicateDesc`, and other Apple Event Manager functions that return a descriptor record.

When you create a descriptor record using the `AECreateDesc` function, you specify the descriptor type, a buffer that contains the data, and the size of this buffer as parameters. The `AECreateDesc` function returns the descriptor record that describes the data.

This example creates a descriptor record for the second parameter of the Multiply event:

```
VAR
   number2:          LongInt;
   multParam2Desc:   AEDesc;
   myErr:            OSErr;

number2 := 8;
myErr := AECreateDesc(typeLongInteger, @number2, SizeOf(number2),
                      multParam2Desc);
```

In this example, the `AECreateDesc` function creates a descriptor record with the `typeLongInteger` descriptor type and the data identified in the `number2` variable.

Once you have created a descriptor record, you can use `AEPutParamDesc` to add the data to an Apple event parameter. You specify the Apple event to add the parameter to, the keyword of the parameter, and the descriptor record of the parameter as parameters to the `AEPutParamDesc` function.

This example adds a second parameter to the Multiply event using the `AEPutParamDesc` function:

```
CONST keyOperand2 = 'OPN2';

myErr := AEPutParamDesc(theAppleEvent, keyOperand2,
                        multParam2Desc);
```

This example adds the `keyOperand2` keyword and the descriptor record created in the previous example as the second parameter to the specified Apple event.

You can also create a descriptor record without using Apple Event Manager routines. For example, this example generates an alias descriptor record from an existing alias handle:

```
WITH myAliasDesc DO
   BEGIN
      descriptorType := typeAlias;
      dataHandle := myAliasHandle;
   END;
```

Whatever method you use to create a descriptor record, you can add it to an Apple event parameter by using `AEPutParamDesc`.

After adding parameters to an Apple event, you can send the Apple event using the `AESend` function. See "Sending an Apple Event," which begins on page 5-13, for information about using this function.

Specifying Optional Parameters for an Apple Event

The parameters for a given Apple event are listed in the *Apple Event Registry: Standard Suites* as either required or optional. Your application does not usually have to include Apple event parameters that are listed as optional; the target application uses default values for parameters that are listed as optional if your application does not provide them. The *Apple Event Registry: Standard Suites* defines the default value a target application should use for each optional parameter of a specific Apple event.

The guidelines listed in the *Apple Event Registry: Standard Suites* for which parameters should be considered optional and which should be considered required are not enforced by the Apple Event Manager. Instead, the source application indicates which Apple event parameters it considers optional by listing the keywords for those parameters in the `keyOptionalKeywordAttr` attribute.

The `keyOptionalKeywordAttr` attribute does not contain the optional parameters; it simply lists the keywords of any parameters for the Apple event that the source application wants to identify as optional. Although the source application is responsible for providing this information in the `keyOptionalKeywordAttr` attribute of an Apple event, it is not required to provide this attribute.

If a keyword for an Apple event parameter is not included in the `keyOptionalKeywordAttr` attribute, the source application expects the target application to understand the Apple event parameter identified by that keyword. If a target application cannot understand the parameter, it should return the result code `errAEParamMissed` and should not attempt to handle the event.

If a keyword for an Apple event parameter is included in the `keyOptionalKeywordAttr` attribute, the source application does not require the target application to understand the Apple event parameter identified by that keyword. If the target application cannot understand a parameter whose keyword is included in the `keyOptionalKeywordAttr` attribute, it should ignore that parameter and attempt to handle the Apple event as it normally does.

A source application can choose not to list the keyword for an Apple event parameter in the `keyOptionalKeywordAttr` attribute even if that parameter is listed in the *Apple Event Registry: Standard Suites* as an optional parameter. This has the effect of forcing the target application to treat the parameter as required for a particular Apple event. If the target application supports the parameter, it should handle the Apple event as the client application expects. If the target application does not support the parameter and calls an application-defined routine such as `MyGotRequiredParams` to check whether it has received all the required parameters, it finds that there's another parameter that the client application considered required, and should return the result code `errAEParamMissed`.

If a source application wants a target application to attempt to handle an Apple event regardless of whether the target application supports a particular Apple event parameter included in that Apple event, the source application should list the keyword for that parameter in the `keyOptionalKeywordAttr` attribute.

It is up to the source application to decide whether to list a parameter that is described as optional in the *Apple Event Registry: Standard Suites* in the `keyOptionalKeywordAttr` attribute of an Apple event. For example, suppose a source application has extended the definition of the Print event to include an optional `keyColorOrGrayscale` parameter that specifies printing in color or gray scale rather than black and white. The source application might decide whether or not to list the keyword `keyColorOrGrayscale` in the `keyOptionalKeywordAttr` attribute according to the characteristics of the print request. If the source application requires the target application to print a document in color, the source application could choose not to add the keyword `keyColorOrGrayscale` to the `keyOptionalKeywordAttr` attribute; in this case, only target applications that supported the `keyColorOrGrayscale` parameter would attempt to handle the event. If the source application does not require the document printed in color, it could choose to add the keyword `keyColorOrGrayscale` to the `keyOptionalKeywordAttr` attribute; in this case, the target application will attempt to handle the event regardless of whether it supports the `keyColorOrGrayscale` parameter.

Your application can add optional parameters to an Apple event the same way it adds required parameters, using the `AECreateDesc`, `AEPutParamPtr`, and `AEPutParamDesc` functions as described in the previous section, "Adding Parameters to an Apple Event." If your application chooses to provide the `keyOptionalKeywordAttr` attribute for an Apple event, it should first create a descriptor list that specifies the keywords of the optional parameters, then add it to the Apple event as a `keyOptionalKeywordAttr` attribute.

Listing 5-1 shows an application-defined routine, `MyCreateOptionalKeyword`, that creates the `keyOptionalKeywordAttr` attribute for the Create Publisher event.

Listing 5-1 Creating the optional keyword for the Create Publisher event

```
FUNCTION MyCreateOptionalKeyword
                        (VAR createPubAppleEvent: AppleEvent)
                         : OSErr;
VAR
   optionalList:  AEDescList;
   myOptKeyword1: AEKeyword;
   myOptKeyword2: AEKeyword;
   myErr:         OSErr;
   ignoreErr:     OSErr;
BEGIN
   myOptKeyword1 := keyDirectObject;
   {create an empty descriptor list}
   myErr := AECreateList(NIL, 0, FALSE, optionalList);
   IF myErr = noErr THEN
   BEGIN
      {add the keyword of the first optional parameter}
      myErr := AEPutPtr(optionalList, 1, typeKeyword,
                        @myOptKeyword1, SizeOf(myOptKeyword1));
      IF myErr = noErr THEN
      BEGIN
         {add the keyword of the next optional parameter}
         myOptKeyword2 := keyAEEditionFileLoc;
         myErr := AEPutPtr(optionalList, 2, typeKeyword,
                           @myOptKeyword2, SizeOf(myOptKeyword2));
      END;
      IF myErr = noErr THEN
         {create the keyOptionalKeywordAttr attribute and add it }
         { to the Create Publisher event}
         myErr := AEPutAttributeDesc(createPubAppleEvent,
                                     keyOptionalKeywordAttr,
                                     optionalList);
   END;
   ignoreErr := AEDisposeDesc(optionalList);
   MyCreateOptionalKeyword := myErr;
END;
```

The `MyCreateOptionalKeyword` function shown in Listing 5-1 adds to a descriptor list the keyword of each parameter that the source application considers optional. Each keyword is added as a descriptor record with the descriptor type `typeKeyword`. The function specifies that the target application can handle the Create Publisher event without supporting parameters identified by the keywords `keyDirectObject` and `keyAEEditionFileLoc`. (These are the parameters that specify the Apple event object to publish and the location of the edition container; if these parameters are missing, the target application creates a publisher for the current selection using the application's default edition container.) After adding these keywords to the descriptor list, the function creates the `keyOptionalKeywordAttr` attribute using the `AEPutAttributeDesc` function.

Typically a target application does not examine the `keyOptionalKeywordAttr` attribute directly. Instead, a target application that supports a parameter listed as optional in the *Apple Event Registry: Standard Suites* attempts to extract it from the Apple event (using `AEGetParamDesc`, for example). If it can't extract the parameter, the target application uses the default value, if any, listed in the *Apple Event Registry.* A target application can use the `keyMissedKeywordAttr` attribute to return the first required parameter (that is, considered required by the source application), if any, that it did not retrieve from the Apple event. The `keyMissedKeywordAttr` attribute does not return any parameters whose keywords are listed in the `keyOptionalKeywordAttr` attribute of the Apple event.

Specifying a Target Address

When you create an Apple event, you must specify the address of the target. The ***target address*** identifies the particular application or process to which you want to send the Apple event. You can send Apple events to applications on the local computer or on remote computers on the network.

These are the descriptor types that identify the four methods of addressing an Apple event:

`typeApplSignature`	The application signature of the target
`typeSessionID`	The session reference number of the target
`typeTargetID`	The target ID record of the target
`typeProcessSerialNumber`	The process serial number of the target

To address an Apple event to a target on a remote computer on the network, you must use either the `typeSessionID` or `typeTargetID` descriptor type.

If your application sends an Apple event to itself, it should address the Apple event using a process serial number of `kCurrentProcess`. This is the fastest way for your application to send an Apple event to itself. For more information, see "Addressing an Apple Event for Direct Dispatching" on page 5-13.

You can use any of the four address types when sending an Apple event to another application on the local computer. The chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* describes all four types of addresses. Your application can

also use another address type if it provides a coercion handler that coerces the address type into one of the four address types that the Apple Event Manager recognizes. See "Writing and Installing Coercion Handlers," which begins on page 4-41, for more information.

To allow the user to choose the target of an Apple event, use the `PPCBrowser` function. This function presents a standard user interface for choosing a target application, much as the Standard File Package provides a standard user interface for opening and saving files. The `PPCBrowser` function returns, in a target ID record, information about the application the user chose. Listing 5-3 on page 5-12 shows how to use the `PPCBrowser` function to let the user choose a target.

Creating an Address Descriptor Record

You specify the address using an address descriptor record (a descriptor record of data type `AEAddressDesc`). You must create a descriptor record of this type and then add it to the Apple event using the `AECreateAppleEvent` function.

You can use the `AECreateDesc` function to create address descriptor records for any of the four types of target addresses. Listing 5-2 shows four possible ways to create an address, each using a different address type.

Listing 5-2 Creating a target address

```
PROCEDURE MySetTargetAddresses(VAR targetAddress1,
                               targetAddress2, targetAddress3,
                               targetAddress4: AEAddressDesc;
                               toTargetID: TargetID;
                               thePSN: ProcessSerialNumber;
                               theSignature: OSType;
                               theSessionRef: PPCSessRefNum);
VAR
   myErr:   OSErr;
BEGIN
   myErr := AECreateDesc(typeTargetID, @toTargetID,
                         SizeOf(toTargetID), targetAddress1);
   myErr := AECreateDesc(typeProcessSerialNumber, @thePSN,
                         SizeOf(thePSN), targetAddress2);
   myErr := AECreateDesc(typeApplSignature, @theSignature,
                         SizeOf(theSignature), targetAddress3);
   myErr := AECreateDesc(typeSessionID, @theSessionRef,
                         SizeOf(theSessionRef), targetAddress4);
   {add your own error checking}
END;
```

To create an address descriptor record, specify the following as parameters to `AECreateDesc`: the descriptor type for the address, a pointer to the buffer containing the address, and the size of the buffer. The `AECreateDesc` function returns an address descriptor record with the specified characteristics.

After creating an address, you can specify it as a parameter to the `AECreateAppleEvent` function. See "Creating an Apple Event," which begins on page 5-3, for an example using the `AECreateAppleEvent` function.

When you specify an address to the `AECreateAppleEvent` function, the Apple Event Manager stores the address in the `keyAddressAttr` attribute of the Apple event.

If you use the `PPCBrowser` function to allow the user to choose an Apple event's target, your application must create a target ID record based on the user's choice. Listing 5-3 shows how to create a target ID record using the information returned from the `PPCBrowser` function and create an address descriptor record using the `AECreateDesc` function.

Listing 5-3 Specifying a target address in an Apple event by using the `PPCBrowser` function

```
FUNCTION MyGetTargetAddress (myPrompt: Str255; myAppStr: Str255;
                             VAR myPortInfo: PortInfoRec;
                             VAR targetAddress: AEAddressDesc;
                             VAR toTargetID: targetID): OSErr;
VAR
   myErr:   OSErr;
BEGIN
   {use PPCBrowser to let user choose the target}
   myErr := PPCBrowser(myPrompt, myAppStr, FALSE,
                       toTargetID.location,
                       myPortInfo, NIL, '');
   MyGetTargetAddress := myErr;
   IF myErr <> noErr THEN Exit(MyGetTargetAddress);

   toTargetID.name := myPortInfo.name;

   {create the descriptor record for the target address}
   MyGetTargetAddress := AECreateDesc(typeTargetID, @toTargetID,
                                      SizeOf(toTargetID),
                                      targetAddress);
END;
```

See the chapter "Program-to-Program Communications Toolbox" in this book for more information on using the `PPCBrowser` function.

Addressing an Apple Event for Direct Dispatching

As described in the chapter "Recording Apple Events" in this book, a recordable application must send itself Apple events in response to user actions. Your application can send itself Apple events by using an address descriptor record of descriptor type `typeProcessSerialNumber` with the `lowLongOfPSN` field set to `kCurrentProcess` and the `highLongOfPSN` set to 0. The Apple Event Manager processes such Apple events immediately, executing the appropriate Apple event handler directly without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends Apple events to itself.

Apple events your application sends to itself this way do not appear in your application's high-level event queue. This not only speeds up delivery of the event but also avoids situations in which an Apple event sent in response to a user action arrives in the event queue after some other event that really occurred later than the user action. For example, suppose a user chooses Cut from the Edit menu and then clicks in another window. If the Cut event arrives in the queue after the window activate event, a selection in the wrong window might be cut.

Your application can send events to itself using other forms of addressing, such as the true process serial number returned by `GetCurrentProcess`. Because direct dispatching avoids event sequence problems, applications should generally send events to themselves by using an address descriptor record of descriptor type `typeProcessSerialNumber` with the `kCurrentProcess` constant rather than using a true process serial number or an application signature.

IMPORTANT
When Apple event recording has been turned on, the Apple Event Manager records every event that your application sends to itself unless you specify the `kAEDontRecord` flag in the `sendMode` parameter of the `AESend` function. ▲

Sending an Apple Event

To send an Apple event, you first create an Apple event, add parameters and attributes to it, and then use the `AESend` function to send it.

When you send an Apple event, you specify various options to indicate how the server should handle the Apple event. You request a user interaction level from the server and specify whether the server can switch directly to the foreground if user interaction is needed, whether your application is willing to wait for a reply Apple event, whether reconnection is allowed, and whether your application wants a return receipt for the Apple event.

You specify these options by setting flags in the `sendMode` parameter for `AESend`. Here are the constants that represent these flags:

```
CONST kAENoReply        = $00000001;{client doesn't want reply}
      kAEQueueReply     = $00000002;{client wants Apple Event }
                                    { Manager to return }
                                    { reply in event queue}
      kAEWaitReply      = $00000003;{client wants a reply and }
                                    { will give up processor}

      kAENeverInteract  = $00000010;{server application }
                                    { should not interact }
                                    { with user for this }
                                    { Apple event}
      kAECanInteract    = $00000020;{server may interact with }
                                    { user for this Apple }
                                    { event to supply }
                                    { information}
      kAEAlwaysInteract = $00000030;{server may interact with }
                                    { user for this Apple }
                                    { event even if no }
                                    { information is required}

      kAECanSwitchLayer = $00000040;{server should come }
                                    { directly to foreground }
                                    { when appropriate}
      kAEDontReconnect  = $00000080;{don't reconnect if there }
                                    { is a PPC session closed }
                                    { error}
      kAEWantReceipt    = nReturnReceipt;{client wants return }
                                         { receipt}
      kAEDontRecord     = $00001000;{don't record this event}
      kAEDontExecute    = $00002000;{don't execute this event}
```

If you want your application to receive a reply Apple event, specify the `kAEQueueReply` or `kAEWaitReply` flag. If you want your application to receive the reply Apple event in its event queue, use `kAEQueueReply`. If you want your application to receive the reply Apple event in the `reply` parameter for `AESend` and you are willing to give up the processor while it is waiting for the reply, use `kAEWaitReply`. If you don't want your application to receive a reply Apple event and your application doesn't need to wait for the server to handle the Apple event, specify `kAENoReply`.

Note
Before the Apple Event Manager sends a reply event back to the client application, the `keyAddressAttr` attribute contains the address of the client application. After the client receives the reply event, the `keyAddressAttr` attribute contains the address of the server application. ◆

If you specify `kAENoReply` or `kAEQueueReply`, the `AESend` function returns immediately after using the Event Manager to send the event. In this case, a `noErr` result code from `AESend` indicates that the Event Manager sent the Apple event; it does not mean that the server accepted or handled the Apple event.

When `AESend` returns, the `reply` parameter does not contain valid data if your application specifies `kAENoReply` or `kAEQueueReply`. The `kAENoReply` flag indicates that the Apple Event Manager will not return the reply Apple event to your application. The `kAEQueueReply` flag indicates that you want your application to receive the reply via its event queue rather than the `reply` parameter of `AESend`. If you specify `kAEQueueReply`, you must install a handler for the reply Apple event (event class `kCoreEventClass` and event ID `kAEAnswer`).

If you specify `kAEWaitReply`, the Apple Event Manager uses the Event Manager to send the event. The Apple Event Manager then calls the `WaitNextEvent` function on behalf of your application, causing your application to yield the processor and giving the server application a chance to receive and handle the Apple event. Your application continues to yield the processor until the server handles the Apple event or the request times out.

If you specify `kAEWaitReply`, you must provide an idle function. This function should process any update events, null events, operating-system events, or activate events that occur while your application is waiting for a reply. See "Writing an Idle Function," which begins on page 5-22, for sample code that shows an idle function.

You use one of the three flags—`kAENeverInteract`, `kAECanInteract`, and `kAEAlwaysInteract`—to specify whether the server should interact with the user when handling the Apple event. Specify `kAENeverInteract` if the server should not interact with the user when handling the Apple event. You might specify this constant if you don't want the user to be interrupted while the server is handling the Apple event.

Use the `kAECanInteract` flag if the server should interact with the user when the user needs to supply information to the server. Use the `kAEAlwaysInteract` flag if the server should interact with the user whenever the server normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. Note that it is the responsibility of the server and client applications to agree on how to interpret the `kAEAlwaysInteract` flag.

If the client application does not set any one of the user interaction flags, the Apple Event Manager sets a default, depending on the location of the target of the Apple event. If the server application is on a remote computer, the Apple Event Manager sets the `kAENeverInteract` flag as the default. If the target of the Apple event is on the local computer, the Apple Event Manager sets the `kAECanInteract` flag as the default.

The server application should call `AEInteractWithUser` if it needs to interact with the user. If both the client and the server allow user interaction, the Apple Event Manager attempts to bring the server to the foreground if it is not already the foreground process. If both the `kAECanSwitchLayer` and the `kAEWaitReply` flags are set, and if the client application is the active application on the local computer, the Apple Event Manager brings the server application directly to the front. Otherwise, the Apple Event Manager posts a notification request asking the user to bring the server application to the front, regardless of whether the `kAECanSwitchLayer` flag is set. This ensures that the user will not be interrupted by an unexpected application switch.

You should specify the `kAECanSwitchLayer` flag only when the client and server applications reside on the same computer. In general, you should not set this flag if it would be confusing or inconvenient to the user for the server application to come to the front unexpectedly. This flag is ignored if you are sending an Apple event to a remote computer.

Specify the `kAEDontReconnect` flag if the Apple Event Manager should not reconnect if it receives a session closed error from the PPC Toolbox. If you don't set this flag, the Apple Event Manager automatically attempts to reconnect and reestablish the session.

Specify the `kAEWantReceipt` flag if your application wants notification that the server application has accepted the Apple event. If you specify this flag, your application receives a return receipt as a high-level event.

If you specify the `kAEWantReceipt` flag and the server application does not accept the Apple event within the time specified by the `timeOutInTicks` parameter to `AESend`, the `AESend` function returns a timeout error. Note that `AESend` also returns a timeout error if your application sets the `kAEWaitReply` flag and does not receive the reply Apple event within the time specified by the `timeOutInTicks` parameter.

Specify the `kAEDontRecord` flag if your application is sending an Apple event to itself that you don't want to be recorded. When Apple event recording has been turned on, every event that your application sends to itself will be automatically recorded by the Apple Event Manager except those sent with the `kAEDontRecord` flag set.

Specify the `kAEDontExecute` flag if your application is sending an Apple event to itself for recording purposes only—that is, if you want the Apple Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event. (For more information about when to use the `kAEDontExecute` flag, see the chapter "Recording Apple Events" in this book.)

Listing 5-4 illustrates how to send a Multiply event (an imaginary Apple event for multiplying two long integers). It first creates an Apple event, adds parameters containing the numbers to multiply, then sends it, specifying various options. It also illustrates how to handle the reply Apple event that contains the result.

Note

If you want to send Apple events, your application must set flags in its `'SIZE'` resource indicating that it can handle high-level events, and it must provide handlers for the required Apple events. See "Accepting an Apple Event" on page 4-5 for information on setting the appropriate flags in the `'SIZE'` resource and "Handling the Required Apple Events" on page 4-11 for information on supporting the required Apple events. ◆

Listing 5-4 Sending an Apple event

```
FUNCTION MySendMultiplyEvent (serverAddress: AEAddressDesc;
                              firstOperand: LongInt; secondOperand: LongInt;
                              VAR replyResultLongInt: LongInt): OSErr;
CONST
   kArithmeticClass  =  'ARTH';  {event class for arithmetic }
                                 { Apple events}
   kMultiplyEventID  =  'MULT';  {event ID for Multiply event}
   keyMultOperand1   =  'OPN1';  {keyword for first parameter}
   keyMultOperand2   =  'OPN2';  {keyword for second parameter}
VAR
   theAppleEvent:    AppleEvent;
   reply:            AppleEvent;
   returnedType:     DescType;
   actualSize:       LongInt;
   myErr:            OSErr;
   ignoreErr:        OSErr;
   errStr:           Str255;
   errNumber:        LongInt;
BEGIN
   myErr := AECreateAppleEvent(kArithmeticClass, kMultiplyEventID,
                               serverAddress, kAutoGenerateReturnID,
                               kAnyTransactionID, theAppleEvent);
   IF myErr = noErr THEN
      {add the first operand}
      myErr := AEPutParamPtr(theAppleEvent, keyMultOperand1,
                             typeLongInteger, @firstOperand,
                             SizeOf(firstOperand));
   {add the second operand with the proper keyword}
   IF myErr = noErr THEN
      myErr := AEPutParamPtr(theAppleEvent, keyMultOperand2,
                             typeLongInteger, @secondOperand,
                             SizeOf(secondOperand));

   IF myErr = noErr THEN
      myErr := AESend(theAppleEvent, reply, kAEWaitReply + kAENeverInteract,
                      kAENormalPriority, 120, @MyIdleFunction, NIL);
   IF myErr = noErr THEN {Apple event successfully sent}
   BEGIN    {Check whether it was successfully handled-- }
            { get result code returned by the server's handler}
      myErr := AEGetParamPtr(reply, keyErrorNumber, typeLongInteger,
                             returnedType, @errNumber, SizeOf(errNumber),
                             actualSize);
```

```
      IF (myErr = errAEDescNotFound) OR (errNumber = noErr) THEN
      {if keyErrorNumber doesn't exist or server returned noErr }
      { then the Apple event was successfully handled--the reply Apple }
      { event contains the result in the direct parameter}
         myErr := AEGetParamPtr(reply, keyDirectObject, typeLongInteger,
                                returnedType, @replyResultLongInt,
                                SizeOf(replyResultLongInt), actualSize)
      ELSE
      BEGIN    {server returned an error, so get error string}
         myErr := AEGetParamPtr(reply, keyErrorString, typeChar,
                                returnedType, @errStr[1], SizeOf(errStr)-1,
                                actualSize);
         IF myErr = noErr THEN
         BEGIN
            IF actualSize > 255 THEN
               actualSize := 255;
            errStr[0] := chr(actualSize);
            MyDisplayError(errStr);
         END;
      END;
   ignoreErr := AEDisposeDesc(reply);
   END
   ELSE
   BEGIN
      {the Apple event wasn't successfully dispatched, }
      { the request timed out, the user canceled, or other error}
   END;
   ignoreErr := AEDisposeDesc(theAppleEvent);
   MySendMultiplyEvent := myErr;
END;
```

The code in Listing 5-4 first creates an Apple event with `kArithmeticClass` as the event class and `kMultiplyEventID` as the event ID. It also specifies the server of the Apple event. See "Specifying a Target Address" on page 5-10 for information on specifying a target address and "Creating an Apple Event," which begins on page 5-3, for more information on creating an Apple event.

The Multiply event shown in Listing 5-4 contains two parameters, each specifying a number to multiply. See "Adding Parameters to an Apple Event" on page 5-5 for examples of how to specify the parameters for the `AEPutParamPtr` function.

After adding the parameters to the event, the code uses `AESend` to send the event. The first parameter to `AESend` specifies the Apple event to send—in this example, the Multiply event. The next parameter specifies the reply Apple event.

This example specifies `kAEWaitReply` in the third parameter, indicating that the client is willing to yield the processor for the specified timeout value (120 ticks, or 2 seconds). The `kAENeverInteract` flag indicates that the server should not interact with the user when processing the Apple event. The fourth parameter specifies that the Multiply event is to be sent using normal priority (that is, placed at the end of the event queue). You can specify the `kAEHighPriority` flag to place the event in the front of the event queue, but this is not usually recommended.

The next to last parameter specifies the address of an idle function. If you specify `kAEWaitReply`, you must provide an idle function. This function should process any update events, null events, operating-system events, or activate events that occur while your application is waiting for a reply. See "Writing an Idle Function," which begins on page 5-22, for sample code that shows an idle function.

The last parameter to `AESend` specifies a filter function. You can supply a filter function to filter high-level events that your application may receive while waiting for a reply Apple event. You can specify `NIL` for this parameter if you do not need to filter high-level events while waiting for a reply. See "Writing a Reply Filter Function" on page 5-24 for more information.

If you specify `kAEWaitReply`, a `noErr` result code from `AESend` indicates that the Apple event was sent successfully, not that the server has completed the requested action successfully. Therefore, you should find out whether a result code was returned from the handler by checking the reply Apple event for the existence of either the `keyErrorNumber` or `keyErrorString` parameter. If the `keyErrorNumber` parameter does not exist or contains the `noErr` result code, you can use `AEGetParamPtr` to get the parameter you're interested in from the reply Apple event.

The `MySendMultiplyEvent` function in Listing 5-4 checks the function result of `AESend`. If it is `noErr`, `MySendMultiplyEvent` checks the `keyErrorNumber` parameter of the reply Apple event to determine whether the server successfully handled the Apple event. If this parameter exists and indicates that an error occurred, `MySendMultiplyEvent` gets the error string out of the `keyErrorString` parameter. Otherwise, the server performed the request, and the reply Apple event contains the answer to the multiplication request.

When you have finished using the Apple event specified in the `AESend` function and no longer need the reply Apple event, you must dispose of both the original event and the reply by calling the `AEDisposeDesc` function.

IMPORTANT

If your application sends Apple events to itself using a `typeProcessSerialNumber` address descriptor record with the `lowLongOfPSN` field set to `kCurrentProcess`, the Apple Event Manager jumps directly to the appropriate Apple event handler without going through the normal event-processing sequence. For this reason, your application will not appear to run more slowly when it sends Apple events to itself. For more information, see "Addressing an Apple Event for Direct Dispatching" on page 5-13. ▲

Dealing With Timeouts

When your application calls `AESend` and chooses to wait for the server application to handle the Apple event, it can also specify the maximum amount of time it is willing to wait for a response. You can specify a timeout value in the `timeOutInTicks` parameter to `AESend`. You can either specify a particular length of time, in ticks, that your application is willing to wait, or you can specify the `kNoTimeOut` constant or the `kAEDefaultTimeout` constant.

Use the `kNoTimeOut` constant to indicate that your application is willing to wait forever for a response from the server. You should use this value only if you are sure that the server will respond in a reasonable amount of time. You should also implement a method of checking whether the user wants to cancel. The idle function that you specify as a parameter to `AESend` should check the event queue for any instances of Command-period and immediately return `TRUE` as its function result if it finds a request to cancel in the event queue.

Use the `kAEDefaultTimeout` constant if you want the Apple Event Manager to use a default timeout value. The Apple Event Manager uses a timeout value of about one minute if you specify this constant.

If you set the `kAEWaitReply` flag and the server doesn't have a handler for the Apple event, the server immediately returns the `errAEEventNotHandled` result code. If the server doesn't respond within the length of time specified by the timeout value, `AESend` returns the `errAETimeout` result code and a reply Apple event that contains no data. This result code does not necessarily mean that the server failed to perform the requested action; it means only that the server did not complete processing within the specified time. The server might still be processing the Apple event, and it might still send a reply.

If the server finishes processing the Apple event sometime after the time specified in the `keyTimeoutAttr` attribute has expired, it returns a reply Apple event to `AEProcessAppleEvent`. The Apple Event Manager then adds the actual data to the reply. Thus, your application can continue to check the reply Apple event to see if the server has responded, even after the time expires. If the server has not yet sent the reply when the client attempts to extract data from the reply Apple event, the Apple Event Manager functions return the `errAEReplyNotArrived` result code. After the reply Apple event returns from the server, the client can extract the data in the reply.

Additionally, the server can examine the `keyTimeoutAttr` attribute of the Apple event to determine the timeout value specified by the client. You can use the value of this attribute as a rough estimate of how much time your handler has to respond. You can assume that your handler has less time to respond than the timeout value, because transmitting the Apple event uses some of the available time, as does transmitting the reply Apple event back to the client, and the event may have been in the queue for a while already.

If you set the `kAENoReply` or `kAEQueueReply` flag, the Apple Event Manager ignores any timeout value you specify, because your application is not waiting for the reply. An attempt by the server to examine the `keyTimeoutAttr` attribute in this situation generates the error `errAEDescNotFound`.

If your handler needs more time than is specified in the `keyTimeoutAttr` attribute, you can reset the timer by using the `AEResetTimer` function. This function resets the timeout value of an Apple event to its starting value.

Writing an Idle Function

This section describes how to write an idle function for use with the `AESend` or `AEInteractWithUser` function.

When your application sends an Apple event, you can set one of three flags in the `sendMode` parameter to `AESend` that specify how you want to deal with the reply: `kAENoReply` if you don't want your application to receive a reply, `kAEQueueReply` if you want it to receive the reply in its event queue, or `kAEWaitReply` if you want the reply returned in the `reply` parameter of `AESend` and you are willing to give up the processor while your application is waiting for the reply.

If you specify `kAENoReply` or `kAEQueueReply`, the `AESend` function returns immediately after using the Event Manager to send the event. If you specify `kAEWaitReply`, the `AESend` function does not return until either the server application finishes handling the Apple event or a specified amount of time expires. In this case the `AESend` function calls `WaitNextEvent` on behalf of your application. This yields the processor to other processes, so that the server has an opportunity to receive and process the Apple event sent by your application. While your application is waiting for a reply, it cannot receive events unless it provides an idle function.

If you provide a pointer to an idle function as a parameter to the `AESend` function, `AESend` calls your idle function whenever an update event, null event, operating-system event, or activate event is received for your application. To allow your application to process high-level events that it receives while waiting for a reply, provide a reply filter function. See the next section, "Writing a Reply Filter Function," for more information.

Your application can yield the processor in a similar manner when it calls the `AEInteractWithUser` function. If `AEInteractWithUser` needs to post a notification request to bring your application to the front, your application yields the processor until the user brings your application to the front. To receive events while waiting for the user to bring your application to the front, you must provide an idle function.

If you provide a pointer to an idle function as a parameter to the `AEInteractWithUser` function, `AEInteractWithUser` calls your idle function whenever an update event, null event, operating-system event, or activate event is received for your application.

An idle function must use this syntax:

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleepTime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
```

The `event` parameter is the event record of the event to process. The `sleepTime` parameter and `mouseRgn` parameter are values that your idle function sets the first time it is called; thereafter they contain the values your function set. Your idle function should return a Boolean value that indicates whether your application wishes to continue waiting. Set the function result to `TRUE` if your application is no longer willing to wait for a reply from the server or for the user to bring the application to the front. Set the function result to `FALSE` if your application is still willing to wait.

You use the `sleepTime` and `mouseRgn` parameters in the same way as the `sleep` and `mouseRgn` parameters of the `WaitNextEvent` function. Specify in the `sleepTime` parameter the amount of time (in ticks) during which your application agrees to relinquish the processor if no events are pending for it.

In the `mouseRgn` parameter, you specify a screen region that determines the conditions under which your application is to receive notice of mouse-moved events. Your idle function receives mouse-moved events only if your application is the front application and the cursor strays outside the region you specify.

Your idle function receives only update events, null events, operating-system events, and activate events. When your idle function receives a null event, it can use the idle time to update a status dialog box, animate cursors, or perform similar tasks. If your idle function receives any of the other events, it should handle the event as it normally would if received in its event loop.

Listing 5-5 shows an example of an idle function for use with `AESend` or `AEInteractWithUser`. The idle function processes update events, null events, operating-system events, and activate events. The first time the function is called it receives a null event. At this time, it sets the `sleepTime` and `mouseRgn` parameters. The function continues to process events until the server finishes handling the Apple event or the user brings the application to the front.

Your application should implement a method of checking whether the user wants to cancel. The `MyCancelInQueue` function in Listing 5-5 checks the event queue for any instances of Command-period and immediately returns `TRUE` as its function result if it finds a request to cancel in the event queue.

Listing 5-5 An idle function

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleeptime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
BEGIN
   MyIdleFunction := FALSE;
```

```
   {the MyCancelInQueue function checks for Command-period}
   IF MyCancelInQueue THEN
      BEGIN
         MyIdleFunction := TRUE;
         Exit(MyIdleFunction);
      END;
   CASE event.what OF
      updateEvt,
      activateEvt,   {every idle function should handle }
      osEvt:         { these kinds of events}
         BEGIN
            MyAdjustCursor(event.where, gCursorRgn);
            DoEvent(event);
         END;
      nullEvent:
         BEGIN
            {set the sleepTime and mouseRgn parameters}
            mouseRgn := gCursorRgn;
            sleeptime := 10;  {use the correct value for your }
                              { app}
            DoIdle;           {the application's idle handling}
         END;
   END; {of CASE}
END;
```

Writing a Reply Filter Function

If your application calls `AESend` and chooses to yield the processor to other processes while waiting for a reply, you can provide an idle function to process update, null, operating-system, and activate events, and you can provide a reply filter function to process high-level events. The previous section describes how an idle function processes events.

Your reply filter function can process any high-level events that it is willing to handle while waiting for a reply Apple event. For example, your application can choose to handle Apple events from other processes while waiting. Note, however, that your application must maintain any necessary state information. Your reply filter function must not accept any Apple events that can change the state of your application and make it impossible to return to its previous state.

A reply filter function must use this syntax:

```
FUNCTION MyReplyFilter (VAR event: EventRecord;
                        returnID: LongInt;
                        transactionID: LongInt;
                        sender: AEAddressDesc): Boolean;
```

The `event` parameter is the event record for a high-level event. The next three parameters contain valid information only if the event is an Apple event. The `returnID` parameter is the return ID for the Apple event. The `transactionID` parameter is the transaction ID for the Apple event. The `sender` parameter contains the address of the application or process that sent the Apple event.

Your reply filter function should return `TRUE` as the function result if you want to accept the Apple event; otherwise, it should return `FALSE`. If your filter function returns `TRUE`, the Apple Event Manager calls the `AEProcessAppleEvent` function on behalf of your application, and your handler routine is called to process the Apple event. In this case, make sure your handler is not called while it is still being used by an earlier call.

Reference to Creating and Sending Apple Events

This section describes the basic Apple Event Manager routines that your application can use to create and send Apple events. It also describes application-defined idle functions and reply filter functions that your application can provide for use by the Apple Event Manager.

For information about data structures used with the routines described in this chapter, see the section "Data Structures Used by the Apple Event Manager," which begins on page 4-56.

Routines for Creating and Sending Apple Events

This section describes the Apple Event Manager routines you can use to create Apple events, create and duplicate descriptor records, create and add items to descriptor lists and AE records, add parameters and attributes to Apple events, and send Apple events. The section "Routines for Responding to Apple Events," which begins on page 4-61, describes other Apple Event Manager routines used for both responding to and creating Apple events.

Creating Apple Events

The `AECreateAppleEvent` function allows you to create an Apple event.

AECreateAppleEvent

You can use the `AECreateAppleEvent` function to create an Apple event with several important attributes but no parameters. You add parameters to the Apple event after you create it.

```
FUNCTION AECreateAppleEvent (theAEEventClass: AEEventClass;
                             theAEEventID: AEEventID;
                             target: AEAddressDesc;
                             returnID: Integer;
                             transactionID: LongInt;
                             VAR result: AppleEvent): OSErr;
```

`theAEEventClass`
The event class of the Apple event to be created.

`theAEEventID`
The event ID of the Apple event to be created.

`target` The address of the server application.

`returnID` The return ID for the Apple event; if you specify `kAutoGenerateReturnID`, the Apple Event Manager assigns a return ID that is unique to the current session.

`transactionID`
The transaction ID for this Apple event. A transaction is a sequence of Apple events that are sent back and forth between the client and server applications, beginning with the client's initial request for a service. All Apple events that are part of a transaction must have the same transaction ID.

`result` The `AECreateAppleEvent` function returns, in this parameter, the Apple event that it creates.

DESCRIPTION

The `AECreateAppleEvent` function creates an Apple event.Your application is responsible for using the `AEDisposeDesc` function to dispose of the Apple event when you no longer need it.

If `AECreateAppleEvent` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone

SEE ALSO

See "Creating an Apple Event," which begins on page 5-3, for more information on how to create an Apple event.

See "Specifying a Target Address" on page 5-10 for information on how to address an Apple event.

Creating and Duplicating Descriptor Records

The `AECreateDesc` function converts data into a descriptor record, and the `AEDuplicateDesc` function makes a copy of a descriptor record.

AECreateDesc

You can use the `AECreateDesc` function to convert data into a descriptor record.

```
FUNCTION AECreateDesc (typeCode: DescType; dataPtr: Ptr;
                       dataSize: Size; VAR result: AEDesc): OSErr;
```

`typeCode`	The descriptor type for the descriptor record.
`dataPtr`	A pointer to the data for the descriptor record.
`dataSize`	The length, in bytes, of the data for the descriptor record.
`result`	The descriptor record that the `AECreateDesc` function creates.

DESCRIPTION

The `AECreateDesc` function creates a new descriptor record that incorporates the specified data. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If `AECreateDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone

SEE ALSO

For examples of the use of `AECreateDesc`, see "Adding Parameters to an Apple Event," which begins on page 5-5, and Listing 5-2 on page 5-11.

AEDuplicateDesc

You can use the `AEDuplicateDesc` function to make a copy of a descriptor record.

```
FUNCTION AEDuplicateDesc (theAEDesc: AEDesc;
                          VAR result: AEDesc): OSErr;
```

`theAEDesc` The descriptor record to be duplicated.

`result` The duplicate descriptor record.

DESCRIPTION

The `AEDuplicateDesc` function creates a new descriptor record by copying the descriptor record from the parameter `theAEDesc`. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If `AEDuplicateDesc` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

It is common for applications to send Apple events that have one or more attributes or parameters in common. For example, if you send a series of Apple events to the same application, the address attribute is the same. In these cases, the most efficient way to create the necessary Apple events is to make a template Apple event that you can then copy—by calling the `AEDuplicateDesc` function—as needed. You then fill in or change the remaining parameters and attributes of the copy, send the copy by calling `AESend`, and dispose of the copy—by calling `AEDisposeDesc`—after `AESend` returns a result code.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone

Creating Descriptor Lists and AE Records

The `AECreateList` function allows you to create an empty descriptor list or AE record.

AECreateList

You can use the `AECreateList` function to create an empty descriptor list or AE record.

```
FUNCTION AECreateList (factoringPtr: Ptr; factoredSize: Size;
                       isRecord: Boolean;
                       VAR resultList: AEDescList): OSErr;
```

`factoringPtr`
A pointer to the data at the beginning of each descriptor that is the same for all descriptor records in the list. If there is no common data, or if you decide not to isolate the common data, specify `NIL` as the value of this parameter.

`factoredSize`
The size of the common data. If there is no common data, or if you decide not to isolate the common data, the value of `factoredSize` must be 0. (See the description that follows for more information.)

`isRecord` A Boolean value that specifies the kind of list to create. If you set it to `TRUE`, the Apple Event Manager creates an AE record. If you set it to `FALSE`, the Apple Event Manager creates a descriptor list.

`resultList`
The descriptor list or AE record that the `AECreateList` function creates.

DESCRIPTION

The `AECreateList` function creates an empty descriptor list or AE record. Your application is responsible for using the `AEDisposeDesc` function to dispose of the resulting descriptor record when you no longer need it. You normally do this after receiving a result code from the `AESend` function.

If you intend to use a descriptor list for a factored Apple event array, you must provide, in the `factoringPtr` parameter, a pointer to the data shared by all items in the array and, in the `factoredSize` parameter, the size of the common data. The common data must be 4, 8, or more than 8 bytes in length because it always consists of (a) the descriptor type (4 bytes); (b) the descriptor type (4 bytes) and the size of each item's data (4 bytes); or (c) the descriptor type (4 bytes), the size of each item's data (4 bytes), and some portion of the data itself (1 or more bytes).

If `AECreateList` returns a nonzero result code, it returns a null descriptor record unless the Apple Event Manager is not available because of limited memory.

RESULT CODES

`noErr`	0	No error
`paramErr`	–50	Parameter error (value of handler pointer is `NIL` or odd)
`memFullErr`	–108	Not enough room in heap zone

SEE ALSO

For an example of the use of `AECreateList`, see Listing 5-1 on page 5-9.

For information about data types used with Apple event arrays, see "Apple Event Array Data Types" on page 4-60.

Adding Items to Descriptor Lists

The Apple Event Manager provides three routines that allow you to add descriptor records to any descriptor list, including an Apple event record. The `AEPutPtr` function converts data specified in a buffer to a descriptor record and adds the descriptor record to a descriptor list. The `AEPutDesc` function adds a descriptor record to a descriptor list. The `AEPutArray` function puts the data for an Apple event array into a descriptor list.

AEPutPtr

You can use the `AEPutPtr` routine to add data specified in a buffer to any descriptor list as a descriptor record.

```
FUNCTION AEPutPtr (theAEDescList: AEDescList; index: LongInt;
                   typeCode: DescType; dataPtr: Ptr;
                   dataSize: Size): OSErr;
```

`theAEDescList`
The descriptor list to which to add a descriptor record.

`index` The position of the descriptor record in the descriptor list. (For example, the value 2 specifies the second descriptor record in the list.) If there is already a descriptor record in the specified position, it is replaced. If the value of `index` is 0, the descriptor record is added to the end of the list.

`typeCode` The descriptor type for the resulting descriptor record.

`dataPtr` A pointer to the data for the descriptor record.

`dataSize` The length, in bytes, of the data for the descriptor record.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed
`errAEIllegalIndex`	–1719	Not a valid list index

SEE ALSO

For an example of the use of `AEPutPtr`, see Listing 5-1 on page 5-9.

AEPutDesc

You can use the `AEPutDesc` function to add a descriptor record to any descriptor list.

```
FUNCTION AEPutDesc (theAEDescList: AEDescList; index: LongInt;
                    theAEDesc: AEDesc): OSErr;
```

`theAEDescList`
The descriptor list to which to add a descriptor record.

`index` The position of the descriptor record in the descriptor list. (For example, the value 2 specifies the second descriptor record in the list.) If there is already a descriptor record in the specified position, it is replaced. If the value of `index` is 0, the descriptor record is added to the end of the list.

`theAEDesc` The descriptor record to be added to the list.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed
`errAEIllegalIndex`	–1719	Not a valid list index

AEPutArray

You can use the `AEPutArray` function to put the data for an Apple event array into any descriptor list.

```
FUNCTION AEPutArray (theAEDescList: AEDescList;
                     arrayType: AEArrayType;
                     arrayPtr: AEArrayDataPointer;
                     itemType: DescType;
                     itemSize: Size; itemCount: LongInt): OSErr;
```

`theAEDescList`
: The descriptor list into which to put the Apple event array. If there are any items already in the descriptor list, they are replaced.

`arrayType`
: The Apple event array type to be created. This is specified by one of the following constants: `kAEDataArray`, `kAEPackedArray`, `kAEHandleArray`, `kAEDescArray`, or `kAEKeyDescArray`.

`arrayPtr`
: A pointer to the buffer containing the array.

`itemType`
: For arrays of type `kAEDataArray`, `kAEPackedArray`, or `kAEHandleArray`, the descriptor type of array items to be created.

`itemSize`
: For arrays of type `kAEDataArray` or `kAEPackedArray`, the size (in bytes) of the array items to be created.

`itemCount`
: The number of elements in the array.

DESCRIPTION

When you use `AEPutArray` to put an array into a factored descriptor list, each array item must include the data that is common to all the descriptor records in the list. The Apple Event Manager automatically isolates the data you specified in the call to `AECreateList` that is common to all the elements of the array.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record

SEE ALSO

For information about data types and constants used with `AEPutArray`, see "Apple Event Array Data Types" on page 4-60.

For more information about creating descriptor lists for Apple event arrays, see the description of `AECreateList` on page 5-29.

Adding Data and Descriptor Records to AE Records

The Apple Event Manager provides two routines that allow you to add data and descriptor records to AE records. The `AEPutKeyPtr` function takes a pointer to data, a descriptor type, and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record. The `AEPutKeyDesc` function takes a descriptor record and a keyword and converts them into a keyword-specified descriptor record that it adds to an AE record.

AEPutKeyPtr

You can use the `AEPutKeyPtr` function to add a pointer to data, a descriptor type, and a keyword to an AE record as a keyword-specified descriptor record.

```
FUNCTION AEPutKeyPtr (theAERecord: AERecord;
                      theAEKeyword: AEKeyword;
                      typeCode: DescType; dataPtr: Ptr;
                      dataSize: Size): OSErr;
```

`theAERecord`
The AE record to which to add a keyword-specified descriptor record.

`theAEKeyword`
The keyword that identifies the descriptor record. If the AE record already includes a descriptor record with this keyword, it is replaced.

`typeCode` The descriptor type for the keyword-specified descriptor record.

`dataPtr` A pointer to the data for the keyword-specified descriptor record.

`dataSize` The length, in bytes, of the data for the keyword-specified descriptor record.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

AEPutKeyDesc

You can use the `AEPutKeyDesc` function to add a descriptor record and a keyword to an AE record as a keyword-specified descriptor record.

```
FUNCTION AEPutKeyDesc (theAERecord: AERecord;
                       theAEKeyword: AEKeyword;
                       theAEDesc: AEDesc): OSErr;
```

`theAERecord`
The AE record to which to add the keyword-specified descriptor record.

`theAEKeyword`
The keyword specifying the descriptor record. If there was already a keyword-specified descriptor record with this keyword, it is replaced.

`theAEDesc` The descriptor record for the keyword-specified descriptor record.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

Adding Parameters and Attributes to Apple Events

The Apple Event Manager provides four functions that allow you to add Apple event parameters and attributes to an Apple event. The `AEPutParamPtr` and `AEPutParamDesc` functions add parameters to a specified Apple event. The `AEPutAttributePtr` and `AEPutAttributeDesc` functions add attributes to a specified Apple event.

AEPutParamPtr

You can use the `AEPutParamPtr` function to add a pointer to data, a descriptor type, and a keyword to an Apple event as an Apple event parameter.

```
FUNCTION AEPutParamPtr (theAppleEvent: AppleEvent;
                        theAEKeyword: AEKeyword;
                        typeCode: DescType; dataPtr: Ptr;
                        dataSize: Size): OSErr;
```

`theAppleEvent`
The Apple event to which to add a parameter.

`theAEKeyword`
The keyword for the parameter to be added. If the Apple event already included a parameter with this keyword, the parameter is replaced.

`typeCode` The descriptor type for the parameter.

`dataPtr` A pointer to the data for the parameter.

`dataSize` The length, in bytes, of the data for the parameter.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For an example of the use of `AEPutParamPtr`, see "Adding Parameters to an Apple Event," which begins on page 5-5.

AEPutParamDesc

You can use the `AEPutParamDesc` function to add a descriptor record and a keyword to an Apple event as an Apple event parameter.

```
FUNCTION AEPutParamDesc (theAppleEvent: AppleEvent;
                         theAEKeyword: AEKeyword;
                         theAEDesc: AEDesc): OSErr;
```

`theAppleEvent`
The Apple event to which to add a parameter.

`theAEKeyword`
The keyword for the parameter to be added. If the Apple event already included a parameter with this keyword, the parameter is replaced.

`theAEDesc` The descriptor record for the parameter.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAEWrongDataType`	–1703	Wrong descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record
`errAEBadListItem`	–1705	Operation involving a list item failed

SEE ALSO

For an example of the use of `AEPutParamDesc`, see "Adding Parameters to an Apple Event," which begins on page 5-5.

AEPutAttributePtr

You can use the `AEPutAttributePtr` function to add a pointer to data, a descriptor type, and a keyword to an Apple event as an attribute.

```
FUNCTION AEPutAttributePtr (theAppleEvent: AppleEvent;
                            theAEKeyword: AEKeyword;
                            typeCode: DescType;
                            dataPtr: Ptr; dataSize: Size): OSErr;
```

`theAppleEvent`
The Apple event to which to add an attribute.

`theAEKeyword`
The keyword for the attribute to be added.

```
TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;
```

The keyword can be any of the constants listed in the description that follows. If the Apple event already included an attribute with this keyword, the attribute is replaced.

`typeCode` The descriptor type for the attribute.

`dataPtr` A pointer to the buffer containing the data to be assigned to the attribute.

`dataSize` The length, in bytes, of the data to be assigned to the attribute.

DESCRIPTION

The `AEPutAttributePtr` function adds the specified pointer to data, descriptor type, and keyword to the specified Apple event as an attribute. You can specify the parameter `theAEKeyWord` using any of the following constants:

```
CONST
   keyAddressAttr          = 'addr'; {address of target }
                                     { application}
   keyEventClassAttr       = 'evcl'; {event class}
   keyEventIDAttr          = 'evid'; {event ID}
   keyEventSourceAttr      = 'esrc'; {source application}
   keyInteractLevelAttr    = 'inte'; {settings to allow the }
                                     { Apple Event Manager to }
                                     { bring server application }
                                     { to the foreground}
```

```
keyMissedKeywordAttr    = 'miss'; {first required parameter }
                                  { remaining in Apple event}
keyOptionalKeywordAttr  = 'optk'; {list of optional }
                                  { parameters for Apple }
                                  { event}
keyOriginalAddressAttr  = 'from'; {address of original source }
                                  { of Apple event}
keyReturnIDAttr         = 'rtid'; {return ID for reply Apple }
                                  { event}
keyTimeoutAttr          = 'timo'; {length of time in ticks }
                                  { that client will wait }
                                  { for reply or result from }
                                  { the server}
keyTransactionIDAttr    = 'tran'; {transaction ID identifying }
                                  { a series of Apple events}
```

RESULT CODES

noErr	0	No error
memFullErr	–108	Not enough room in heap zone
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record

AEPutAttributeDesc

You can use the `AEPutAttributeDesc` function to add a descriptor record and a keyword to an Apple event as an attribute.

```
FUNCTION AEPutAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             theAEDesc: AEDesc): OSErr;
```

`theAppleEvent`
The Apple event to which you are adding an attribute.

`theAEKeyword`
The keyword for the attribute to be added.

```
TYPE AEKeyword = PACKED ARRAY[1..4] OF Char;
```

The keyword can be any of the constants listed in the description of `AEPutAttributePtr` on page 5-36. If the Apple event already included an attribute with this keyword, the attribute is replaced.

`theAEDesc` The descriptor record to be assigned to the attribute. The descriptor type of the specified descriptor record should match the defined descriptor type for that attribute. For example, the `keyEventSourceAttr` attribute has the `typeShortInteger` descriptor type.

DESCRIPTION

The `AEPutAttributeDesc` function takes a descriptor record and a keyword and adds them to an Apple event as an attribute. If the descriptor type required for the attribute is different from the descriptor type of the descriptor record, the Apple Event Manager attempts to coerce the descriptor record into the required type, with one exception: the Apple Event Manager does not attempt to coerce the data for an address attribute, thereby allowing applications to use their own address types.

RESULT CODES

`noErr`	0	No error
`memFullErr`	–108	Not enough room in heap zone
`errAECoercionFail`	–1700	Data could not be coerced to the requested descriptor type
`errAENotAEDesc`	–1704	Not a valid descriptor record

SEE ALSO

For an example of the use of `AEPutAttributeDesc`, see Listing 5-1 on page 5-9.

Sending Apple Events

The `AESend` function allows you to send an Apple event that you have previously created with the `AECreateAppleEvent` function.

AESend

You can use the `AESend` function to send an Apple event.

```
FUNCTION AESend (theAppleEvent: AppleEvent;
                 VAR reply: AppleEvent; sendMode: AESendMode;
                 sendPriority: AESendPriority;
                 timeOutInTicks: LongInt; idleProc: IdleProcPtr;
                 filterProc: EventFilterProcPtr): OSErr;
```

`theAppleEvent`
The Apple event to be sent.

`reply` The reply Apple event returned by the `AESend` function if you specify the `kAEWaitReply` flag in the `sendMode` parameter. (If you specify the `kAEQueueReply` flag in the `sendMode` parameter, you receive the reply Apple event in your event queue.) If you specify `kAENoReply` flag, the reply Apple event returned by this function is a null descriptor record. If you specify `kAEWaitReply` in the `sendMode` parameter, your application is responsible for using the `AEDisposeDesc` function to dispose of the descriptor record returned in the `reply` parameter.

`sendMode` Specifies the following: the reply mode for the Apple event (set with one of the constants `kAENoReply`, `kAEQueueReply`, or `kAEWaitReply`); the interaction level (set with one of the constants `kAENeverInteract`, `kAECanInteract`, or `kAEAlwaysInteract`, which represent flags in the `keyInteractLevelAttr` attribute); the application switch mode (set with the `kAECanSwitchLayer` constant); the reconnection mode (set with the `kAEDontReconnect` constant); and the return receipt mode (set with the `kAEWantReceipt` constant). You obtain the value for this parameter by adding the appropriate constants. (The description that follows provides more details about the `sendMode` flags.)

`sendPriority`
An integer of data type `AESendPriority` that specifies whether the Apple event is put at the back of the event queue (indicated by the `kAENormalPriority` flag) or at the front of the queue (indicated by the `kAEHighPriority` flag).

`timeOutInTicks`
If the reply mode specified in the `sendMode` parameter is `kAEWaitReply`, or if a return receipt is requested, this parameter specifies the length of time (in ticks) that the client application is willing to wait for the reply or return receipt from the server application before timing out. Most applications should use the `kAEDefaultTimeout` constant, which tells the Apple Event Manager to provide an appropriate timeout duration. If the value of this parameter is `kNoTimeOut`, the Apple event never times out.

`idleProc` A pointer to a function that handles events (such as update, operating-system, activate, and null events) that your application receives while waiting for a reply. Your application can also perform other tasks (such as displaying a wristwatch or spinning beachball cursor) while waiting for a reply or a return receipt. Your application must provide an idle function if it specifies the `kAEWaitReply` flag in the `sendMode` parameter.

`filterProc`
A pointer to a function that accepts certain incoming Apple events that are received while the handler waits for a reply or a return receipt and filters out the rest.

DESCRIPTION

You can use one of the following flags in the `sendMode` parameter to specify the reply mode for an Apple event. Only one of these flags may be set.

Flag	Description
`kAENoReply`	Your application does not want a reply Apple event; the server processes your Apple event as soon as it has the opportunity.
`kAEQueueReply`	Your application wants a reply Apple event; the reply appears in your event queue as soon as the server has the opportunity to process and respond to your Apple event.
`kAEWaitReply`	Your application wants a reply Apple event and is willing to give up the processor while waiting for the reply; for example, if the server application is on the same computer as your application, your application yields the processor to allow the server to respond to your Apple event. If you specify `kAEWaitReply`, you should provide an idle function.

You can communicate your user interaction preferences to the server application by specifying one of the following flags in the `sendMode` parameter. Only one of these flags may be set.

Flag	Description
`kAENeverInteract`	The server application should never interact with the user in response to the Apple event. If this flag is set, `AEInteractWithUser` returns the `errAENoUserInteraction` result code. This flag is the default when an Apple event is sent to a remote application.
`kAECanInteract`	The server application can interact with the user in response to the Apple event—by convention, if the user needs to supply information to the server. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request. This flag is the default when an Apple event is sent to a local application.
`kAEAlwaysInteract`	The server application can interact with the user in response to the Apple event—by convention, whenever the server application normally asks a user to confirm a decision or interact in any other way, even if no additional information is needed from the user. If this flag is set and the server allows interaction, `AEInteractWithUser` either brings the server application to the foreground or posts a notification request.

The flags in the following list specify the application switch mode, the reconnection mode, and the return receipt mode. Any of these flags may be set.

Flag	Description
`kAECanSwitchLayer`	If both the client and server allow interaction, and if the client application is the active application on the local computer and is waiting for a reply (that is, it has set the `kAEWaitReply` flag), `AEInteractWithUser` brings the server directly to the foreground. Otherwise, `AEInteractWithUser` uses the Notification Manager to request that the user bring the server application to the foreground.
`kAEDontReconnect`	The Apple Event Manager must not automatically try to reconnect if it receives a `sessClosedErr` result code from the PPC Toolbox.
`kAEWantReceipt`	The sender wants to receive a return receipt for this Apple event from the Event Manager. (A return receipt means only that the receiving application accepted the Apple event; the Apple event may or may not be handled successfully after it is accepted.) If the receiving application does not send a return receipt before the request times out, `AESend` returns `errAETimeout` as its function result.

If the Apple Event Manager cannot find a handler for an Apple event in either the application or system Apple event dispatch table, it returns the result code `errAEEventNotHandled` to the server application (as the result of the `AEProcessAppleEvent` function). If the client application is waiting for a reply, the Apple Event Manager also returns this result code to the client.

The `AESend` function returns `noErr` as its function result if the Apple event was successfully sent by the Event Manager. A `noErr` result from `AESend` does not indicate that the Apple event was handled successfully; it indicates only that the Apple event was successfully sent by the Event Manager. If the handler returns a result code other than `noErr`, and if the client is waiting for a reply, it is returned in the `keyErrorNumber` parameter of the reply Apple event.

If your application is sending an event to itself, you can set one of these flags to prevent the event from being recorded or to ask the Apple Event Manager to record the event without your application actually receiving it. Only one of these flags may be set.

Flag	Description
`kAEDontRecord`	Your application is sending an event to itself but does not want the event recorded. When Apple event recording is on, the Apple Event Manager records a copy of every event your application sends to itself except for those events for which this flag is set.
`kAEDontExecute`	Your application is sending an Apple event to itself for recording purposes only—that is, you want the Apple Event Manager to send a copy of the event to the recording process but you do not want your application actually to receive the event.

RESULT CODES

`noErr`	0	No error
`eLenErr`	–92	Buffer too big to send
`memFullErr`	–108	Not enough room in heap zone
`userCanceledErr`	–128	User canceled an operation
`procNotFound`	–600	No eligible process with specified process serial number
`connectionInvalid`	–609	Nonexistent signature or session ID
`noUserInteractionAllowed`	–610	Background application sends event requiring authentication
`noPortErr`	–903	Client hasn't set `'SIZE'` resource to indicate awareness of high-level events
`destPortErr`	–906	Server hasn't set `'SIZE'` resource to indicate awareness of high-level events, or else is not present
`sessClosedErr`	–917	The `kAEDontReconnect` flag in the `sendMode` parameter was set and the server quit, then restarted
`errAEEventNotHandled`	–1708	Event wasn't handled by an Apple event handler
`errAEUnknownSendMode`	–1710	Invalid sending mode was passed
`errAEWaitCanceled`	–1711	User canceled out of wait loop for reply or receipt
`errAETimeout`	–1712	Apple event timed out
`errAEUnknownAddressType`	–1716	Unknown Apple event address type

SEE ALSO

For more information on sending Apple events, see "Sending an Apple Event," which begins on page 5-13.

For information on writing an idle function, see "Writing an Idle Function," which begins on page 5-22.

For information on writing a reply filter function, see "Writing a Reply Filter Function," which begins on page 5-24.

For information on when to use the `kAEDontExecute` flag, see the chapter "Recording Apple Events" in this book.

Application-Defined Routines

If your application sends an Apple event using `AESend` and is waiting for a reply, or if it calls `AEInteractWithUser`, you can provide an idle function to handle update events, null events, operating-system events, and activate events. You can also provide a reply filter function that can handle any high-level events that you want your application to handle while it is waiting for a reply or for user interaction.

MyIdleFunction

An idle function has the following syntax:

```
FUNCTION MyIdleFunction (VAR event: EventRecord;
                         VAR sleepTime: LongInt;
                         VAR mouseRgn: RgnHandle): Boolean;
```

`event` The event record of the event to process.

`sleepTime` Amount of time (in ticks) during which your application agrees to relinquish the processor if no events are pending.

`mouseRgn` A screen region that determines the conditions under which your application is to receive notice of mouse-moved events.

DESCRIPTION

If your application provides a pointer to an idle function (`MyIdleFunction`) as a parameter to `AESend` or `AEInteractWithUser`, the Apple Event Manager will call the idle function to handle any update event, null event, operating-system event, or activate event received for your application while it is waiting for a reply.

Set the function result to `TRUE` if your application is no longer willing to wait for a reply from the server or for the user to bring the application to the front. Set the function result to `FALSE` if your application is still willing to wait.

SEE ALSO

For more information, see "Writing an Idle Function," which begins on page 5-22.

MyReplyFilter

A reply filter function has the following syntax:

```
FUNCTION MyReplyFilter (VAR event: EventRecord;
                        returnID: LongInt;
                        transactionID: LongInt;
                        sender: AEAddressDesc): Boolean;
```

`event` The event record for a high-level event. The next three parameters contain valid information only if the event is an Apple event.

`returnID` Return ID for the Apple event.

`transactionID` Transaction ID for the Apple event.

`sender` Address of process that sent the Apple event.

DESCRIPTION

If your application provides a pointer to a reply filter function as a parameter to the `AESend` function, the reply filter function can process any high-level events that it is willing to handle while your application is waiting for a reply.

Your reply filter function should return `TRUE` as the function result if you want to accept the Apple event; otherwise, it should return `FALSE`.

SEE ALSO

For more information, see "Writing a Reply Filter Function" on page 5-24.

Summary of Creating and Sending Apple Events

Pascal Summary

Constants

```
CONST
   gestaltAppleEventsAttr     = 'evnt';   {selector for Apple events}
   gestaltAppleEventsPresent  = 0;        {if this bit is set, then Apple }
                                          { Event Manager is available}

   {Apple event descriptor types}
   typeBoolean             = 'bool';      {1-byte Boolean value}
   typeChar                = 'TEXT';      {unterminated string}
   typeSMInt               = 'shor';      {16-bit integer}
   typeInteger             = 'long';      {32-bit integer}
   typeSMFloat             = 'sing';      {SANE single}
   typeFloat               = 'doub';      {SANE double}
   typeLongInteger         = 'long';      {32-bit integer}
   typeShortInteger        = 'shor';      {16-bit integer}
   typeLongFloat           = 'doub';      {SANE double}
   typeShortFloat          = 'sing';      {SANE single}
   typeExtended            = 'exte';      {SANE extended}
   typeComp                = 'comp';      {SANE comp}
   typeMagnitude           = 'magn';      {unsigned 32-bit integer}
   typeAEList              = 'list';      {list of descriptor records}
   typeAERecord            = 'reco';      {list of keyword-specified }
                                          { descriptor records}
   typeAppleEvent          = 'aevt';      {Apple event record}
   typeTrue                = 'true';      {TRUE Boolean value}
   typeFalse               = 'fals';      {FALSE Boolean value}
   typeAlias               = 'alis';      {alias record}
   typeEnumerated          = 'enum';      {enumerated data}
   typeType                = 'type';      {four-character code for }
                                          { event class or event ID}
   typeAppParameters       = 'appa';      {Process Manager launch parameters}
   typeProperty            = 'prop';      {Apple event property}
   typeFSS                 = 'fss ';      {file system specification}
```

```
typeKeyword             = 'keyw';      {Apple event keyword}
typeSectionH            = 'sect';      {handle to a section record}
typeWildCard            = '****';      {matches any type}
typeApplSignature       = 'sign';      {application signature}
typeSessionID           = 'ssid';      {session reference number}
typeTargetID            = 'targ';      {target ID record}
typeProcessSerialNumber = 'psn ';      {process serial number}
typeNull                = 'null';      {NULL or nonexistent data}

{keywords for Apple event parameters}
keyDirectObject         = '----';      {direct parameter}
keyErrorNumber          = 'errn';      {error number parameter}
keyErrorString          = 'errs';      {error string parameter}
keyProcessSerialNumber  = 'psn ';      {process serial number param}

{keywords for Apple event attributes}
keyTransactionIDAttr    = 'tran';      {transaction ID}
keyReturnIDAttr         = 'rtid';      {return ID}
keyEventClassAttr       = 'evcl';      {event class}
keyEventIDAttr          = 'evid';      {event ID}
keyAddressAttr          = 'addr';      {address of target or }
                                       { client application}
keyOptionalKeywordAttr  = 'optk';      {list of optional parameters }
                                       { for the Apple event}
keyTimeoutAttr          = 'timo';      {number of ticks the client }
                                       { will wait}
keyInteractLevelAttr    = 'inte';      {settings to allow Apple Event }
                                       { Manager to bring server }
                                       { to foreground}
keyEventSourceAttr      = 'esrc';      {nature of source }
                                       { application}
keyMissedKeywordAttr    = 'miss';      {first required parameter }
                                       { remaining in an Apple event}
keyOriginalAddressAttr  = 'from';      {address of original source; }
                                       { available only in version }
                                       { 1.01 and later versions of }
                                       { the Apple Event Manager}

{keywords for special handlers}
keyPreDispatch          = 'phac';      {identifies a handler routine }
                                       { called immediately before the }
                                       { Apple Event Manager dispatches }
                                       { an Apple event}
```

```
keySelectProc           = 'selh';      {selector used with }
                                       { AERemoveSpecialHandler to }
                                       { disable the OSL}

{keywords for use with AEManagerInfo; available only in version }
{ 1.0.1 and later versions of the Apple Event Manager}
keyAERecorderCount      = 'recr';      {keyword for recording info}
keyAEVersion            = 'vers';      {keyword for version info}

{event class}
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}
kAEAnswer               = 'ansr';      {event ID for Apple event replies}
kAEApplicationDied      = 'obit';      {event ID for Application Died }
                                       { event}

{constants for setting the sendMode parameter of AESend}
kAENoReply              = $00000001;   {client doesn't want reply}
kAEQueueReply           = $00000002;   {client wants server to }
                                       { reply in event queue}
kAEWaitReply            = $00000003;   {client wants a reply and }
                                       { will give up processor}
kAENeverInteract        = $00000010;   {server application should }
                                       { not interact with user }
                                       { for this Apple event}
kAECanInteract          = $00000020;   {server may interact with }
                                       { user for this Apple event }
                                       { to supply information}

kAEAlwaysInteract       = $00000030;   {server may interact with user }
                                       { for this Apple event even if }
                                       { no information is required}
```

```
kAECanSwitchLayer       = $00000040;   {server should come directly }
                                       { to foreground when appropriate}
kAEDontReconnect        = $00000080;   {don't reconnect if there }
                                       { is a PPC session closed error}
kAEWantReceipt          = nReturnReceipt; {client wants return }
                                       { receipt}
kAEDontRecord           = $00001000;   {don't record this event}
kAEDontExecute          = $00002000;   {don't excecute this event}

{constants for setting the sendPriority parameter of AESend}
kAENormalPriority       = $00000000;   {put event at back of }
                                       { event queue}
kAEHighPriority         = nAttnMsg;    {put event at front of }
                                       { the event queue}

{event IDs for recording events; available only in version 1.01 and }
{ later versions of the Apple Event Manager}
kAEStartRecording       = 'reca';      {event ID for Start Recording }
                                       { event}
kAEStopRecording        = 'recc';      {event ID for Stop Recording }
                                       { event}
kAENotifyStartRecording = 'rec1';      {event ID for Recording On event}
kAENotifyStopRecording  = 'rec0';      {event ID for Recording Off event}
kAENotifyRecording      = 'recr';      {event ID for Receive Recordable }
                                       { Event event}

{constant for the returnID parameter of AECreateAppleEvent}
kAutoGenerateReturnID   = -1;          {tells Apple Event Manager to }
                                       { generate a unique return ID}

{constant for transaction IDs}
kAnyTransactionID       = 0;           {the Apple event is not }
                                       { part of a transaction}

{constants for timeout durations}
kAEDefaultTimeout       = -1;          {use default timeout value}
kNoTimeOut              = -2;          {never time out}

{constants for the dispatcher parameter of AEResumeTheCurrentEvent}
kAENoDispatch           = 0;           {don't redispatch the Apple event}
kAEUseStandardDispatch  = -1;          {redispatch the Apple event }
                                       { by using its entry in the }
                                       { Apple event dispatch table}
```

Data Types

```
TYPE
   AEEventClass =
      PACKED ARRAY[1..4] OF Char;            {event class for a high-level }
                                             { event}
   AEEventID =
      PACKED ARRAY[1..4] OF Char;            {event ID for a high-level }
                                             { event}

   AEKeyword =
      PACKED ARRAY[1..4] OF Char;            {keyword for a descriptor }
                                             { record}

   DescType                   = ResType;     {descriptor type}

   AEDesc =                                  {descriptor record}
   RECORD
      descriptorType:         DescType;      {type of data being passed}
      dataHandle:             Handle;        {handle to data being passed}
   END;

   AEKeyDesc =                               {keyword-specified }
   RECORD                                    { descriptor record}
      descKey:                AEKeyword;     {keyword}
      descContent:            AEDesc;        {descriptor record}
   END;

   AEAddressDesc              = AEDesc;      {address descriptor record}

   AEDescList                 = AEDesc;      {list of descriptor records}

   AERecord                   = AEDescList;  {list of keyword-specified }
                                             { descriptor records}

   AppleEvent                 = AERecord;    {list of attributes and }
                                             { parameters necessary for }
                                             { an Apple event}

   AESendMode                 = LongInt;     {flags that determine how }
                                             { an Apple event is sent}

   AESendPriority             = Integer;     {send priority of an Apple }
                                             { event}
```

```
AEInteractAllowed = (kAEInteractWithSelf, kAEInteractWithLocal,
                     kAEInteractWithAll); {what processes may }
                                          { interact with the user}

AEEventSource = (kAEUnknownSource, kAEDirectCall, kAESameProcess,
                 kAELocalProcess, kAERemoteProcess);
                                          {the source of an Apple }
                                          { event}

AEArrayType = (kAEDataArray, kAEPackedArray, kAEHandleArray,
               kAEDescArray, kAEKeyDescArray);
                                          {type of an Apple event array}
AEArrayData =
RECORD                                    {data for an Apple event array}
   CASE AEArrayType OF
   kAEDataArray:
      (AEDataArray: ARRAY[0..0] OF Integer);
   kAEPackedArray:
      (AEPackedArray: Packed Array[0..0] OF Char);
   kAEHandleArray:
      (AEHandleArray: Array[0..0] OF Handle);
   kAEDescArray:
      (AEDescArray: Array[0..0] OF AEDesc);
   kAEKeyDescArray:
      (AEKeyDescArray: Array[0..0] OF AEKeyDesc);
END;

AEArrayDataPointer = ^AEArrayData;

EventHandlerProcPtr = ProcPtr;            {pointer to an Apple event }
                                          { handler}
IdleProcPtr = ProcPtr;                    {pointer to an application's }
                                          { idle function}
EventFilterProcPtr = ProcPtr;             {pointer to an application's }
                                          { filter function}
```

Routines for Creating and Sending Apple Events

Creating Apple Events

```
FUNCTION AECreateAppleEvent  (theAEEventClass: AEEventClass;
                              theAEEventID: AEEventID;
                              target: AEAddressDesc; returnID: Integer;
                              transactionID: LongInt;
                              VAR result: AppleEvent): OSErr;
```

Creating and Duplicating Descriptor Records

```
FUNCTION AECreateDesc        (typeCode: DescType; dataPtr: Ptr;
                              dataSize: Size; VAR result: AEDesc): OSErr;
FUNCTION AEDuplicateDesc     (theAEDesc: AEDesc; VAR result: AEDesc): OSErr;
```

Creating Descriptor Lists and AE Records

```
FUNCTION AECreateList        (factoringPtr: Ptr; factoredSize: Size;
                              isRecord: Boolean;
                              VAR resultList: AEDescList): OSErr;
```

Adding Items to Descriptor Lists

```
FUNCTION AEPutPtr            (theAEDescList: AEDescList; index: LongInt;
                              typeCode: DescType; dataPtr: Ptr;
                              dataSize: Size): OSErr;
FUNCTION AEPutDesc           (theAEDescList: AEDescList; index: LongInt;
                              theAEDesc: AEDesc): OSErr;
FUNCTION AEPutArray          (theAEDescList: AEDescList;
                              arrayType: AEArrayType;
                              arrayPtr: AEArrayDataPointer;
                              itemType: DescType; itemSize: Size;
                              itemCount: LongInt): OSErr;
```

Adding Data and Descriptor Records to AE Records

```
FUNCTION AEPutKeyPtr         (theAERecord: AERecord;
                              theAEKeyword: AEKeyword; typeCode: DescType;
                              dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutKeyDesc        (theAERecord: AERecord;
                              theAEKeyword: AEKeyword;
                              theAEDesc: AEDesc): OSErr;
```

Adding Parameters and Attributes to Apple Events

```
FUNCTION AEPutParamPtr      (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword; typeCode: DescType;
                             dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutParamDesc     (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             theAEDesc: AEDesc): OSErr;
FUNCTION AEPutAttributePtr  (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword; typeCode: DescType;
                             dataPtr: Ptr; dataSize: Size): OSErr;
FUNCTION AEPutAttributeDesc (theAppleEvent: AppleEvent;
                             theAEKeyword: AEKeyword;
                             theAEDesc: AEDesc): OSErr;
```

Sending Apple Events

```
FUNCTION AESend             (theAppleEvent: AppleEvent;
                             VAR reply: AppleEvent; sendMode: AESendMode;
                             sendPriority: AESendPriority;
                             timeOutInTicks: LongInt;
                             idleProc: IdleProcPtr;
                             filterProc: EventFilterProcPtr): OSErr;
```

Application-Defined Routines

```
FUNCTION MyIdleFunction     (VAR event: EventRecord;
                             VAR sleepTime: LongInt;
                             VAR mouseRgn: RgnHandle): Boolean;
FUNCTION MyReplyFilter      (VAR event: EventRecord;
                             returnID: LongInt; transactionID: LongInt;
                             sender: AEAddressDesc): Boolean;
```

C Summary

Constants

```
enum {
      #define gestaltAppleEventsAttr     'evnt' /*selector for Apple events*/
      gestaltAppleEventsPresent        = 0   /*if this bit is set, then */
                                             /* Apple Event Manager is */
};                                           /* available*/
```

```
      /*Apple event descriptor types*/
enum  {
      typeBoolean             = 'bool',      /*1-byte Boolean value*/
      typeChar                = 'TEXT',      /*unterminated string*/
      typeSMInt               = 'shor',      /*16-bit integer*/
      typeInteger             = 'long',      /*32-bit integer*/
      typeSMFloat             = 'sing',      /*SANE single*/
      typeFloat               = 'doub',      /*SANE double*/
      typeLongInteger         = 'long',      /*32-bit integer*/
      typeShortInteger        = 'shor',      /*16-bit integer*/
      typeLongFloat           = 'doub',      /*SANE double*/
      typeShortFloat          = 'sing',      /*SANE single*/
      typeExtended            = 'exte',      /*SANE extended*/
      typeComp                = 'comp',      /*SANE comp*/
      typeMagnitude           = 'magn',      /*unsigned 32-bit integer*/
      typeAEList              = 'list',      /*list of descriptor records*/
      typeAERecord            = 'reco',      /*list of keyword-specified */
                                             /* descriptor records*/
      typeAppleEvent          = 'aevt',      /*Apple event record*/
      typeTrue                = 'true',      /*TRUE Boolean value*/
      typeFalse               = 'fals',      /*FALSE Boolean value*/
      typeAlias               = 'alis',      /*alias record*/
      typeEnumerated          = 'enum'       /*enumerated data*/
};

enum {
      typeType                = 'type',      /*four-character code for */
                                             /* event class or event ID*/
      typeAppParameters       = 'appa',      /*Process Manager launch */
                                             /* parameters*/
      typeProperty            = 'prop',      /*Apple event property*/
      typeFSS                 = 'fss ',      /*file system specification*/
      typeKeyword             = 'keyw',      /*Apple event keyword*/

      typeSectionH            = 'sect',      /*handle to a section record*/
      typeWildCard            = '****',      /*matches any type*/
      typeApplSignature       = 'sign',      /*application signature*/
      typeSessionID           = 'ssid',      /*session ID*/
      typeTargetID            = 'targ',      /*target ID record*/
      typeProcessSerialNumber = 'psn ',      /*process serial number*/
      typeNull                = 'null'       /*NULL or nonexistent data*/
};
```

```
      /*keywords for Apple event parameters*/
enum {
      keyDirectObject         = '----',      /*direct parameter*/
      keyErrorNumber          = 'errn',      /*error number parameter*/
      keyErrorString          = 'errs',      /*error string parameter*/
      keyProcessSerialNumber  = 'psn '       /*process serial number param*/
};
      /*keywords for Apple event attributes*/
enum {
      keyTransactionIDAttr    = 'tran',      /*transaction ID*/
      keyReturnIDAttr         = 'rtid',      /*return ID*/
      keyEventClassAttr       = 'evcl',      /*event class*/
      keyEventIDAttr          = 'evid',      /*event ID*/
      keyAddressAttr          = 'addr',      /*address of target or */
                                             /* client application*/
      keyOptionalKeywordAttr  = 'optk',      /*list of optional parameters */
                                             /* for the Apple event*/
      keyTimeoutAttr          = 'timo',      /*number of ticks the client */
                                             /* will wait*/
      keyInteractLevelAttr    = 'inte',      /*settings to allow Apple */
                                             /* Event Mgr to bring */
                                             /* server to foreground*/
      keyEventSourceAttr      = 'esrc',      /*nature of source */
                                             /* application*/
      keyMissedKeywordAttr    = 'miss',      /*first required parameter */
                                             /* remaining in an Apple */
                                             /* event*/
      keyOriginalAddressAttr  = 'from'       /*address of original source; */
                                             /* available only in version */
                                             /* 1.01 and later versions of */
                                             /* the Apple Event Manager*/
};

      /*keywords for special handlers*/
enum {
      keyPreDispatch          = 'phac',      /*identifies a handler */
                                             /* routine that is called */
                                             /* immediately before the */
                                             /* Apple Event Manager */
                                             /* dispatches an Apple event*/
      keySelectProc           = 'selh',      /*selector used with */
                                             /* AERemoveSpecialHandler to */
                                             /* disable the OSL*/
```

```
      /*keywords for use with AEManagerInfo, available only in version */
      /* 1.0.1 and later versions of the Apple Event Manager*/
      keyAERecorderCount      = 'recr',      /*keyword for recording info*/
      keyAEVersion            = 'vers',      /*keyword for version info*/

      /*event class*/
      kCoreEventClass         = 'aevt'       /*event class for required */
                                             /* Apple events*/
};

      /*event IDs for required Apple events*/
enum {
      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*/
      kAEAnswer               = 'ansr',      /*event ID for Apple event */
                                             /* replies*/
      kAEApplicationDied      = 'obit'       /*event ID for Application */
                                             /* Died event*/
};
      /*constants for setting the sendMode parameter of AESend*/
enum {
      kAENoReply              = 0x00000001,  /*client doesn't want reply*/
      kAEQueueReply           = 0x00000002,  /*client wants server to */
                                             /* reply in event queue*/
      kAEWaitReply            = 0x00000003,  /*client wants a reply and */
                                             /* will give up processor*/
      kAENeverInteract        = 0x00000010,  /*server application should */
                                             /* not interact with user */
                                             /* for this Apple event*/
      kAECanInteract          = 0x00000020,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* to supply information*/
      kAEAlwaysInteract       = 0x00000030,  /*server may interact with */
                                             /* user for this Apple event */
                                             /* even if no information */
                                             /* is required*/
```

```
      kAECanSwitchLayer       = 0x00000040,  /*server should come */
                                             /* directly to foreground */
                                             /* when appropriate*/
      kAEDontReconnect        = 0x00000080,  /*don't reconnect if there */
                                             /* is a PPC session closed */
                                             /* error*/
      kAEWantReceipt          = nReturnReceipt, /*client wants return */
                                             /* receipt*/
      kAEDontRecord           = 0x00001000,  /*don't record this event*/
      kAEDontExecute          = 0x00002000,  /*don't excecute this event*/

      /*constants for setting the sendPriority parameter of AESend*/
      kAENormalPriority       = 0x00000000,  /*post message at end of */
                                             /* event queue*/
      kAEHighPriority         = nAttnMsg     /*post message at front of */
                                             /* event queue*/
};
      /*event IDs for recording events; available only in version 1.01 and */
      /* later versions of the Apple Event Manager*/
enum {
      kAEStartRecording       = 'reca',      /*event ID for Start */
                                             /* Recording event*/
      kAEStopRecording        = 'recc',      /*event ID for Stop */
                                             /* Recording event*/
      kAENotifyStartRecording = 'rec1',      /*event ID for Recording On */
                                             /* event*/
      kAENotifyStopRecording  = 'rec0',      /*event ID for Recording Off */
                                             /* event*/
      kAENotifyRecording      = 'recr'       /*event ID for Receive */
                                             /* Recordable Event event*/
};
enum {
      /*constant for the returnID parameter of AECreateAppleEvent*/
      kAutoGenerateReturnID   = -1,          /*tells Apple Event Manager */
                                             /* to generate a unique */
                                             /* return ID*/
      /*constant for transaction IDs*/
      kAnyTransactionID       = 0,           /*the Apple event is not */
                                             /* part of a transaction*/

      /*constants for timeout durations*/
      kAEDefaultTimeout       = -1,          /*use default timeout value*/
      kNoTimeOut              = -2,          /*never time out*/
```

```
      /*constants for the dispatcher parameter of AEResumeTheCurrentEvent*/
      kAENoDispatch           = 0,          /*don't redispatch the */
                                            /* Apple event*/
      kAEUseStandardDispatch  = -1        /*redispatch the Apple event */
                                            /* by using its entry in the */
                                            /* Apple event dispatch table*/
};
```

Data Types

```
typedef unsigned long AEEventClass;         /*event class for a */
                                            /* high-level event*/
typedef unsigned long AEEventID;            /*event ID for a high-level */
                                            /* event*/
typedef unsigned long AEKeyword;            /*keyword for a descriptor */
                                            /* record*/
typedef ResType DescType;                   /*descriptor type*/

struct AEDesc {                             /*descriptor record*/
   DescType descriptorType;                 /*type of data being passed*/
   Handle dataHandle;                       /*handle to data being passed*/
};
typedef struct AEDesc AEDesc;

struct AEKeyDesc {                          /*keyword-specified */
                                            /* descriptor record*/
   AEKeyword descKey;                       /*keyword*/
   AEDesc descContent;                      /*descriptor record*/
};
typedef struct AEKeyDesc AEKeyDesc;

typedef AEDesc AEAddressDesc;               /*address descriptor record*/
typedef AEDesc AEDescList;                  /*list of descriptor records*/
typedef AEDescList AERecord;                /*list of keyword-specified */
                                            /* descriptor records*/
typedef AERecord AppleEvent;                /*list of attributes and */
                                            /* parameters necessary for */
                                            /* an Apple event*/
typedef long AESendMode;                    /*flags that determine how */
                                            /* an Apple event is sent*/
```

```
typedef short AESendPriority;               /*send priority of an Apple */
                                            /* event*/

enum { kAEInteractWithSelf, kAEInteractWithLocal,
       kAEInteractWithAll };                /*what processes may */
typedef unsigned char AEInteractAllowed;    /* interact with the user*/

enum { kAEUnknownSource, kAEDirectCall, kAESameProcess, kAELocalProcess,
       kAERemoteProcess };                  /*the source of an Apple */
typedef unsigned char AEEventSource;        /* event*/
enum { kAEDataArray, kAEPackedArray, kAEHandleArray,
       kAEDescArray, kAEKeyDescArray };     /*type of an Apple event */
typedef unsigned char AEArrayType;          /* array*/

union AEArrayData {                         /*data for an Apple event */
   short kAEDataArray[1];                   /* array*/
   char kAEPackedArray[1];
   Handle kAEHandleArray[1];
   AEDesc kAEDescArray[1];
   AEKeyDesc kAEKeyDescArray[1];
};
typedef union AEArrayData AEArrayData;

typedef AEArrayData *AEArrayDataPointer;

typedef ProcPtr EventHandlerProcPtr;        /*pointer to an Apple event */
                                            /* handler*/
typedef ProcPtr IdleProcPtr;                /*pointer to an application's */
                                            /* idle function*/
typedef ProcPtr EventFilterProcPtr;         /*pointer to an application's */
                                            /* filter function*/
```

Routines for Creating and Sending Apple Events

Creating Apple Events

```
pascal OSErr AECreateAppleEvent
                            (AEEventClass theAEEventClass,
                             AEEventID theAEEventID,
                             const AEAddressDesc *target, short returnID,
                             long transactionID, AppleEvent *result);
```

Creating and Duplicating Descriptor Records

```
pascal OSErr AECreateDesc   (DescType typeCode, const void* dataPtr,
                             Size dataSize, AEDesc *result);
pascal OSErr AEDuplicateDesc
                            (const AEDesc *theAEDesc, AEDesc *result);
```

Creating Descriptor Lists and AE Records

```
pascal OSErr AECreateList   (const void* factoringPtr, Size factoredSize,
                             Boolean isRecord, AEDescList *resultList);
```

Adding Items to Descriptor Lists

```
pascal OSErr AEPutPtr       (const AEDescList *theAEDescList, long index,
                             DescType typeCode, const void* dataPtr,
                             Size dataSize);
pascal OSErr AEPutDesc      (const AEDescList *theAEDescList, long index,
                             const AEDesc *theAEDesc);
pascal OSErr AEPutArray     (const AEDescList *theAEDescList,
                             AEArrayType arrayType,
                             const AEArrayDataPointer *arrayPtr,
                             DescType itemType, Size itemSize,
                             long itemCount);
```

Adding Data and Descriptor Records to AE Records

```
pascal OSErr AEPutKeyPtr    (const AERecord *theAERecord,
                             AEKeyword theAEKeyword, DescType typeCode,
                             const void* dataPtr, Size dataSize);
pascal OSErr AEPutKeyDesc   (const AERecord *theAERecord,
                             AEKeyword theAEKeyword,
                             const AEDesc *theAEDesc);
```

Adding Parameters and Attributes to Apple Events

```
pascal OSErr AEPutParamPtr  (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType typeCode,
                             const void* dataPtr, Size dataSize);
pascal OSErr AEPutParamDesc (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword,
                             const AEDesc *theAEDesc);
pascal OSErr AEPutAttributePtr
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword, DescType typeCode,
                             const void* dataPtr, Size dataSize);
```

```
pascal OSErr AEPutAttributeDesc
                            (const AppleEvent *theAppleEvent,
                             AEKeyword theAEKeyword,
                             const AEDesc *theAEDesc);
```

Sending Apple Events

```
pascal OSErr AESend         (const AppleEvent *theAppleEvent,
                             AppleEvent *reply, AESendMode sendMode,
                             AESendPriority sendPriority,
                             long timeOutInTicks, IdleProcPtr idleProc,
                             EventFilterProcPtr filterProc);
```

Application-Defined Routines

```
pascal Boolean MyIdleFunction
                            (const EventRecord *event,
                             long *sleepTime, RgnHandle *mouseRgn);
pascal Boolean MyReplyFilter
                            (const EventRecord *event,
                             long returnID, long transactionID,
                             AEAddressDesc sender);
```

Assembly-Language Summary

Trap Macros

Trap Macros Requiring Routine Selectors

```
_Pack8
```

Selector	Routine
$0405	AEDuplicateDesc
$0609	AEPutDesc
$0610	AEPutKeyDesc
$0610	AEPutParamDesc
$0627	AEPutAttributeDesc
$0706	AECreateList
$0825	AECreateDesc
$0A08	AEPutPtr
$0A0F	AEPutKeyPtr

Selector	Routine
$0A0F	AEPutParamPtr
$0A16	AEPutAttributePtr
$0B0D	AEPutArray
$0B14	AECreateAppleEvent
$0D17	AESend

Result Codes

noErr	0	No error
paramErr	–50	Parameter error (for example, value of handler pointer is NIL or odd)
eLenErr	–92	Buffer too big to send
memFullErr	–108	Not enough room in heap zone
userCanceledErr	–128	User canceled an operation
procNotFound	–600	No eligible process with specified process serial number
bufferIsSmall	–607	Buffer is too small
noOutstandingHLE	–608	No outstanding high-level event
connectionInvalid	–609	Nonexistent signature or session ID
noUserInteractionAllowed	–610	Background application sends event requiring authentication
noPortErr	–903	Client hasn't set 'SIZE' resource to indicate awareness of high-level events
destPortErr	–906	Server hasn't set 'SIZE' resource to indicate awareness of high-level events, or else is not present
sessClosedErr	–917	The kAEDontReconnect flag in the sendMode parameter was set, and the server quit and then restarted
errAECoercionFail	–1700	Data could not be coerced to the requested descriptor type
errAEDescNotFound	–1701	Descriptor record was not found
errAECorruptData	–1702	Data in an Apple event could not be read
errAEWrongDataType	–1703	Wrong descriptor type
errAENotAEDesc	–1704	Not a valid descriptor record
errAEBadListItem	–1705	Operation involving a list item failed
errAENewerVersion	–1706	Need a newer version of the Apple Event Manager
errAENotAppleEvent	–1707	Event is not an Apple event
errAEEventNotHandled	–1708	Event wasn't handled by an Apple event handler
errAEReplyNotValid	–1709	AEResetTimer was passed an invalid reply
errAEUnknownSendMode	–1710	Invalid sending mode was passed
errAEWaitCanceled	–1711	User canceled out of wait loop for reply or receipt
errAETimeout	–1712	Apple event timed out
errAENoUserInteraction	–1713	No user interaction allowed
errAENotASpecialFunction	–1714	The keyword is not valid for a special function
errAEParamMissed	–1715	Handler cannot understand a parameter the client considers required
errAEUnknownAddressType	–1716	Unknown Apple event address type

`errAEHandlerNotFound`	–1717	No handler found for an Apple event or a coercion, or no object callback function found
`errAEReplyNotArrived`	–1718	Reply has not yet arrived
`errAEIllegalIndex`	–1719	Not a valid list index
`errAEImpossibleRange`	–1720	The range is not valid because it is impossible for a range to include the first and last objects that were specified; an example is a range in which the offset of the first object is greater than the offset of the last object
`errAEWrongNumberArgs`	–1721	The number of operands provided for the `kAENot` logical operator is not 1
`errAEAccessorNotFound`	–1723	There is no object accessor function for the specified object class and token descriptor type
`errAENoSuchLogical`	–1725	The logical operator in a logical descriptor record is not `kAEAnd`, `kAEOr`, or `kAENot`
`errAEBadTestKey`	–1726	The descriptor record in a test key is neither a comparison descriptor record nor a logical descriptor record
`errAENotAnObjectSpec`	–1727	The `objSpecifier` parameter of `AEResolve` is not an object specifier record
`errAENoSuchObject`	–1728	A run-time resolution error, for example: object specifier record asked for the third element, but there are only two
`errAENegativeCount`	–1729	Object-counting function returned negative value
`errAEEmptyListContainer`	–1730	The container for an Apple event object is specified by an empty list
`errAEUnknownObjectType`	–1731	Descriptor type of token returned by `AEResolve` is not known to server application
`errAERecordingIsAlreadyOn`	–1732	Attempt to turn recording on when it is already on