Sending an Apple Event

To send an Apple event, you first create an Apple event, add parameters and attributes to the Apple event, 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 directly switch 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 in the *sendMode* parameter to **AESend**. Here are the constants that represent these options.

kAENoReply	client doesn't want reply
kAEQueueReply	client wants server to reply in event queue
kAEWaitReply	client wants a reply and will give up processor
kAENeverInteract	server application should not interact with user for this Apple event
kAECanInteract	server may interact with user for this Apple event to supply information
kAEAlwaysInteract	server may interact with user for this Apple event even if no information is required
kAECanSwitchLayer	server comes directly to foreground when appropriate
kAEDontReconnect	don't reconnect if there is a PPC sessClosedErr
kAEWantReceipt	client wants return receipt

If your application wants a reply Apple event, specify the kAEQueueReply or kAEWaitReply flag. If your application wants to receive the reply Apple event in its event queue, use kAEQueueReply. If your application wants to receive the reply Apple event in the *reply* parameter of **AESend** and is willing to give up the processor while waiting for the reply, use kAEWaitReply. If your application does not want a reply Apple event and does not need to wait for the server to handle the Apple event, specify kAENoReply.

In most cases, your application should use kAEWaitReply or kAENoReply. You should not use kAEQueueReply if your application is sending an Apple event to

itself.

If your application specifies 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 Apple event was successfully sent by the **Event Manager**; it does not mean that the server accepted or handled the Apple event.

Also, the *reply* parameter to **AESend** does not contain valid data on return from **AESend** 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 your application wants to receive the reply in its event queue rather than through the *reply* parameter of **AESend**.

If your application specifies 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. This gives 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.

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 necessary-for example, if the user needs to supply information to the server. Use the kAEAlwaysInteract flag if the server should interact with the user even when no 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 user interaction is allowed, the **Apple Event Manager** brings the server to the front if it is not already the foreground process. If the **kAECanSwitchLayer** flag is set and the principle of user control permits, the **Apple Event Manager** directly brings the server application to the front. If the action is contrary to the principle of user control, the **Apple Event Manager** posts a notification request to inform the user to bring the server application to the front.

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 unexpectedly come to the front.

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 did not accept the Apple event. If you specify this flag, the **AESend** function returns the errAEEventNotHandled result code if the server did not accept the Apple event.

The following program 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 also handle the required Apple Events.

```
// Assuming Inclusion of MacHeaders
#include <AppleEvents.h>

// Prototype your routine like this prior to calling it
OSErr MySendMultiplyEvent(AEAddressDesc *serverAddress,
                          long firstOperand, long
                          secondOperand,
                          long *replyResultLongInt);
// Constants used in procedure
#define kArithmeticClass 'ARTH'    // event class for arithmetic Apple Events
#define kMultiplyEventID 'MULT'    // event ID for Multiply event
#define keyMultOperand1  'OPN1'    // keyword for first parameter
#define keyMultOperand2  'OPN2'    // keyword for second parameter

OSErr MySendMultiplyEvent (AEAddressDesc *serverAddress,
                          long firstOperand,
                          long secondOperand,
                          long *replyResultLongInt)

{
    AppleEvent     theAppleEvent;
    AppleEvent     reply;
    DescType   returnedType;
    long     actualSize;
    OSErr  myErr;
    Str255     errStr;
    long     errNumber;

    // Function Prototypes
    void DoError(OSErr myErr);
    void MyDisplayError(Str255 errStr);

    // callback function passed to AECreateAppleEvent
    pascal Boolean MyIdleFunction(EventRecord *theEventRecord,
                long *sleepTime, RgnHandle *mouseRgn);
```

```
myErr = AECreateAppleEvent(kArithmeticClass, kMultiplyEventID,
            serverAddress, kAutoGenerateReturnID,
            kAnyTransactionID, &theAppleEvent);

if ( myErr )
    DoError(myErr);          // failed to create the event

// add the firstOperand
// Note case of firstOperand to match prototype
myErr = AEPutParamPtr(&theAppleEvent, keyMultOperand1,
            typeLongInteger, (Ptr)&firstOperand, sizeof(firstOperand));

if ( myErr )
    // failed to add first parameter--be sure
    // to dispose of the event before leaving routine
    DoError(myErr);

// add the secondOperand with the proper keyword
// Note cast of secondOperand to match prototype
myErr = AEPutParamPtr(&theAppleEvent, keyMultOperand2,
        typeLongInteger, (Ptr) &secondOperand,
        sizeof(secondOperand));

if ( myErr )
    DoError(myErr);      // be sure to dispose of the event and
                         // first parameter before leaving routine

myErr = AESend(&theAppleEvent, &reply,
            kAEWaitReply + kAENeverInteract,
            kAENormalPriority, 120, &MyIdleFunction, nil);

if ( !myErr ) {
    // Apple event successfully sent; check whether it was successfully
    // handled--get result code returned by the server's handler

    myErr = AEGetParamPtr(&reply, keyErrorNumber,
            typeLongInteger, &returnedType, (Ptr) &errNumber,
            sizeof (errNumber), &actualSize);

    if ((myErr == errAEDescNotFound) || (errNumber == noErr)) {
    // 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,
            (Ptr)&replyResultLongInt, sizeof(replyResultLongInt),
            &actualSize);
        return noErr;
    }

    else {
        // server returned an error, so get error string

        myErr = AEGetParamPtr(&reply, keyErrorString, typeChar,
```

```
                &returnedType, (Ptr)errStr, sizeof(errStr),
                &actualSize);
            if ( !myErr )
                MyDisplayError(errStr);
        }
    }
    else {
        // the Apple event wasn't successfully dispatched,
        // the request timed out, the  user  canceled,  or  other  error
    }

    return myErr;
}
```

This program 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.

The Multiply event shown in the above program contains two parameters, each of which specifies a number to multiply. 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 Multiply event is sent using normal priority, meaning it 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.

The next to last parameter specifies the address of an idle function. If you specify kAEWaitReply you should 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.

The last parameter to **AESend** specifies a filter function. You can supply a filter function to filter high-level events that your application might 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.

If **AESend** returns a noErr result code and your application specified kAEWaitReply, you should first see whether a result code was returned from the handler routine by checking the reply Apple event for the existence of the parameter whose keyword is keyErrorNumber. 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 code this program checks the function result of **AESend**. If **AESend** returns noErr, the code then checks the replyErrorNumber 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, then the code 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 are done using the Apple event specified in the **AESend** function and finished with the reply Apple event, you must dispose of their descriptor records using the **AEDisposeDesc** function.