Technical Note TN1164

Native Scripting Additions

CONTENTS

This Technote describes the scripting addition mechanism for AppleScript introduced in Mac OS 8.6 and it describes extensions to the scripting additions API that allow developers to create scripting additions for Mac OS X.

This Technote is directed at application developers who are interested in creating scripting additions.

[Sep 13 2001]

What are Scripting Additions?

Scripting additions provide a mechanism for delivery of additional functionality that can be used in AppleScripts. The two primary types of functionality that a scripting addition can provide are Apple event handling and Apple event data coercion handling. The techniques used inside a scripting addition to provide Apple event handling services are the same as the techniques used in an application; however, since a scripting addition is not an application, a scripting addition must implement a few additional routines that are used to set up its Apple event handlers and internal state variables. These routines include:

- Initialization operations such as installing event handlers, installing coercion handlers, allocating memory, and setting up global variables.
- Reference counting so the AppleScript environment can determine if it is safe to unload your scripting addition.
- A termination routine where your scripting addition removes handlers, releases allocated memory, and performs any other necessary cleanup operations.

The Apple event handlers and Apple event data coercion handlers installed by a scripting addition are written in basically the same way that handlers used inside an application are written. What differs between scripting additions and applications is the packaging of the code and the mechanisms by which the scripting addition is called to install itself in the system. These differences are discussed in the sections that follow.

Packaging Scripting Additions

Currently, there are two formats for packaging scripting additions. These formats are:

1. Mach-O style Mac OS X bundles.

   Mach-O style scripting additions are packaged as Mac OS X bundles. A scripting addition bundle is identified in one of two ways: the bundle has either (a) a `CFBundleSignature` of `"osax"` or (b) a bundle name ending with the file name extension `".osax"`.

2. CFM single file binaries linked with `CarbonLib` or `InterfaceLib`.

   Scripting additions linked with `CarbonLib` can be used with Mac OS X. CFM binaries are packaged as a single file and are identified by file type and creator. A scripting addition's file type should be set to the value `'osax'`, while the creator code can be either the generic `'ascr'` value or some other value defined by a `'BNDL'` resource included in the file.

   Note:

Additions linked with `InterfaceLib` won't work on Mac OS X, and ones linked with `CarbonLib` are not safe to use on Mac OS 9.

Back to top

Initialization

Your scripting addition's initialization routine is responsible for installing your scripting addition's handler routines, allocating memory, and performing any other set up operations required. Your initialization routine should proceed as follows:

1. Use `Gestalt`, `sysctl`, or other appropriate means to verify that the resources your scripting addition requires to run are installed and available for use. This includes allocating memory, finding files, and so on.
2. Install your scripting addition's Apple event handlers and Coercion handlers. All handlers installed by a scripting addition must be installed in the system dispatch table. For example, the code snippet shown in Listing 1 illustrates the correct parameter values to install handlers in the system dispatch table.

Listing 1. Installing event and coercion handlers in the system dispatch table.

```
    Boolean isSysHandler = true;

    anErr = AEInstallEventHandler( theAEEventClass, theAEEventID,
                            theHandlerUPP, refcon, isSysHandler);

    anErr = AEInstallCoercionHandler( fromType, toType, theHandlerUPP,
                            refcon, fromTypeIsDesc, isSysHandler);
```

IMPORTANT:
If a scripting addition's initialization routine returns any result value other than the value `noErr`, then the scripting addition should not leave any of its handlers installed in the system dispatch table.

3. Perform any other initialization, such as setting up constant values, storing away copies of parameters passed to your initialization routine, and so on.

The actual implementation of the scripting addition's initialization depends on which type of binary executable format your scripting addition has been stored in. Scripting additions saved in CFM format use the code fragment initialization routine for initialization, while scripting additions saved in the Mach-O format must export a routine named `SAInitialize`.

Scripting additions saved in CFM format use the code fragment initialization routine for initialization. If your scripting addition may need to open it's resource file later during execution, then it should store a reference to its resource file at this time. The code snippet in Listing 2 shows a declaration of a scripting addition's initialization routine.

Listing 2. CFM initialization routine for a scripting addition.

```
OSErr CFMSAInitialize(InitBlockPtr initBlkPtr) {
    OSErr err;

    ...initialization statements...

    return err;
}
```

The `initBlkPtr` parameter passed to the scripting addition's initialization routine contains information that can be used to locate the addition's resource file. If any of your handlers will need to access the addition's resource file, then the initialization routine can store a reference to this file among it's globals. Listing 3 illustrates how this can be done.

Listing 3. CFM initialization routine for a scripting addition.

```
static AliasHandle gMyAdditionLocation;
```

```
OSErr CFMSAInitialize(InitBlockPtr initBlkPtr) {
    OSErr err;

        /* if we will need to open the scripting addition's resource
        file inside of one of our handlers,  then save a reference
        to the scripting addition's file in the globals so we can
        access it later. */
    err = NewAlias(NULL, initBlkPtr->fragLocator.u.onDisk.fileSpec,
                  &gMyAdditionLocation);
    if (err == noErr) ...

    return err;
}
```

Scripting additions saved in the Mach-O format bundle for Mac OS X must export a routine named `SAInitialize`. This routine will be called by AppleScript to initialize your scripting addition. Listing 4 provides a sketch of a scripting addition's Mach-O initialization routine:

Listing 4. Mach-O initialization routine for a scripting addition.

```
OSErr SAInitialize(CFBundleRef additionBundle) {
    OSErr err;

    ...initialization statements...

    return err;
}
```

The `additionBundle` parameter passed to `SAInitialize` is a reference to the scripting addition's bundle. This bundle reference can be used to locate the addition's bundle resources during initialization and from inside of the scripting addition's handlers. If any of the scripting addition's handlers will need to access the addition's bundle resources, then the initialization routine should store a copy of the bundle reference among its globals. There is no need to call `CFRetain` on this reference as it will remain valid as long as your scripting addition is loaded.

For more information about how to set the initialization routine for your compiled CFM Scripting Addition, consult the documentation included with your development environment. Information about CFM code fragments, CFM initialization routines, bundle references, and the Mach-O bundle format can be found in the References section at the end of this article.

Back to top

Termination

When the termination routine is called, it must perform any actions needed to close down the scripting addition. Tasks your termination routine must perform include:

- Remove any event handlers or coercion handlers that were installed by your initialization routine.
- Deallocate any memory and release any resources allocated by your scripting addition.

The scripting addition's termination routine is called when AppleScript no longer requires a scripting addition. This will happen the next time AppleScript is initialized after the scripting addition has been removed from the Scripting Additions folder.

Loading/unloading happens whenever someone handles a "gdut" event. AppleScript does this when a component connection is opened and before compiling any script. It actually does *not* happen on system shutdown, so the only time your termination function will ever get called is the next "gdut" after your addition has been removed from the scripting additions folder. This is the same on Mac OS 9 and X, except that on X, additions must be unloaded from each process separately. (i.e., after removing an addition, you must send a "gdut" event to every process it was loaded into to completely get rid of it. On Mac OS 9, sending a "gdut" event to any one application will do.)

Listing 5 shows a hypothetical termination routine for a scripting addition. If your scripting addition is compiled as a CFM binary, then you must set the CFM termination routine to your scripting addition's termination routine; Mach-O binaries export the symbol `SATerminate` that is called by AppleScript when it no longer requires the scripting addition.

Listing 5. Sample Termination routine for a scripting addition.

```
void SATerminate(void) {

    AERemoveEventHandler(theAEEventClass,
                                theAEEventID, gTheHandler, true);

    DisposeAEEventHandlerUPP(gTheHandler);

    ...other cleanup operations...

}
```

For information about how to set the termination routine for your compiled CFM Scripting Addition, consult the documentation included with your development environment. Information about CFM code fragments, CFM initialization routines, bundle references, and the Mach-O bundle format can be found in the References section at the end of this article.

Reference Counting

When AppleScript would like to unload a scripting addition it first queries the scripting addition to determine if there are any outstanding calls to the addition that are still running. If there are, then it is not safe to unload the addition and the unloading process will either be canceled or deferred until the outstanding calls have been completed.

A scripting addition communicates its current execution status back to AppleScript in one of two ways: CFM based scripting additions export a global variable reference named `gAdditionReferenceCount` and Mach-O based scripting additions export a routine named `SAIsBusy`. The value of `gAdditionReferenceCount`, exported by a CFM addition, is used by AppleScript in the same way it is used in the example `SAIsBusy` routine shown in Listing 6. Essentially, if `gAdditionReferenceCount` contains any non-zero value, then the addition is understood to be in the process of completing some outstanding call (and therefore it cannot be unloaded).

Listing 6. Sample `SAIsBusy` routine for a scripting addition.

```
UInt32  gAdditionReferenceCount = 0;

Boolean SAIsBusy(void) {

    return (gAdditionReferenceCount != 0);

}
```

Inside all of your application's Apple event handlers and Apple event data coercion handlers, you should increment the value of gAdditionReferenceCount while executing and decrement the value immediately before your handler returns. Listing 7 illustrates how this is done in a typical handler routine.

Listing 7. Maintaining `gAdditionReferenceCount` in a typical Scripting Addition Apple event handler.

```
UInt32  gAdditionReferenceCount = 0;

     ....

OSErr MyEventHandler(const AppleEvent *ev,
                     AppleEvent *reply,
                     long refcon) {
    OSErr err;

        /* increment the value as the first operation inside
        of your handler */
    gAdditionReferenceCount++;

          ...other handler code goes here...

        /* decrement the value as the last operation before
        your handler returns */
    --gAdditionReferenceCount;
```

```
    return err;
}
```

Back to top

Helpful Tips

Mac OS X Runtime Considerations

In the Mac OS X runtime environment, scripting additions are loaded separately into each application partition that connects to AppleScript. As a result, you should design your scripting addition keeping in mind that there may be many instances of your scripting addition open in many different applications at the same time. As a result, some scripting additions may require additional code if they have been designed to share a single resource such as a printer or a serial port.

Locating Your Scripting Addition's Bundle Resources

Scripting additions written in Mach-O bundle format may want to access resources and files located inside of their bundle. In order to do this, the scripting addition should cache a copy of the `CFBundleRef` passed to the `SAInitialize` so it can access its bundle inside of its handlers.

```
CFBundleRef gMyAdditionBundle;

OSErr SAInitialize(CFBundleRef additionBundle) {

        /* if we will need to open the scripting addition's bundle in
        one of our handlers,  then save a reference to it
        in the globals so we can access it later. */
    gMyAdditionBundle = additionBundle;

    ....
```

Your scripting addition does not need to call `CFRetain` on the bundle reference passed to it, as the reference will remain valid as long as the scripting addition remains open (i.e., until `SATerminate` is called).

Information describing how to access resources inside of your scripting addition's bundle provided in the References section at the end of this article.

Locating Your Scripting Addition's Resource File

A scripting addition provided as a single file CFM binary may need to access its resource fork during the execution of one of its handlers. In order to do so, it should save a reference to its file's location on disk in its initialization routine. Later, when your scripting addition needs to access resources inside this file, it can use this reference to open the resource fork and retrieve the data it requires.

Before the Code Fragment Manager calls your initialization routine, it sets up a pointer to a File Specification Record (`FSSpec`) in the `CFragInitBlock` passed to the initialization routine. Your scripting addition can cache this value among its globals for later use.

```
AliasHandle gMyAdditionLocation;

OSErr ConnectionInitializationRoutine(InitBlockPtr initBlkPtr) {
    OSErr err;

        /* if we will need to open the scripting addition's resource
        file in one of our handlers,  then save a reference
        to the scripting addition's file in the globals so we can
```

```
        access it later. */
    err = NewAlias(NULL,
                   initBlkPtr->fragLocator.u.onDisk.fileSpec,
                   &gMyAdditionLocation);
    if (err == noErr) {
```

Scripting additions should not leave resource files open that were opened inside of their handlers. Also, if a scripting addition does open any resource files inside of any of its handlers, it should take special steps ensure that it does not change the current resource search chain. The following example illustrates steps that should be taken to preserve the current resource chain when opening a resource file in a handler:

```
    SInt16    oldResFile;
    SInt16    osaxResRef;
    FSRef     ref;

    oldResFile = CurResFile();
    osaxResRef = FSOpenResFile( &ref, fsRdPerm );

    // Do your handler stuff here

    CloseResFile( osaxResRef );
    UseResFile( oldResFile );
```

Note:
The file reference used to locate your scripting addition's resource file would have been established in the initialization routine where your scripting addition installed its handlers. This is described in the Initialization section.

Local and Remote Requests

Each of your scripting addition's handler routines is responsible for detecting and rejecting events from remote systems (if appropriate). A handler can determine the source of an event by examining the `keyEventSourceAttr` attribute in the incoming event. An event from a remote system will have an attribute value of `kAERemoteProcess`.

```
    DescType   sourceAttr;
    DescType   actualType;
    Size       actualSize;

    anErr = AEGetAttributePtr( eventPtr, keyEventSourceAttr, typeType,
                               &actualType, &sourceAttr,
                               sizeof( sourceAttr ), &actualSize);

    if ( sourceAttr == kAERemoteProcess ) {

         return errAEEventNotHandled;

    }
```

Back to top

References

- Apple Event Manager Documentation.
- The Code Fragment Manager chapter of Inside Macintosh: PowerPC System Software.
- The CFM-Based Runtime Architecture chapter of Mac OS Runtime Architectures.
- The Core Foundation Bundle Services sections of the Core Foundation Documentation suite.

Back to top

Downloadables

Acrobat version of this Note (64K). Download

Back to top
