Technote 1164

Native Scripting Additions

CONTENTS

This Technote describes the native scripting additions mechanism for AppleScript introduced in Mac OS 8.6.

The native scripting additions mechanism described here addresses limitations in the classic code-resource-based mechanism, and provides a way to package the handlers in a scripting addition as a single shared library. This packaging allows the sharing of code and the ability to maintain a persistent state between calls.

This Technote is directed at application developers who create scripting additions.

Classic Scripting Additions

AppleScript provides a mechanism for extending the syntax of the language through the use of scripting additions (also called osaxen). This classic scripting addition mechanism was designed in the days when the Mac OS ran only on 68K machines, and so is based on code resources. These code resources are of type `'osax'` (hence, the name osaxen), each containing the code for a single handler in the scripting addition.

At startup time, AppleScript installs an event handler in the system dispatch table, which is responsible for loading scripting additions and gathering their terminologies. This handler scans the Scripting Additions folder looking for files of type `'osax'`. When AppleScript finds one, it scans the file's resource fork for `'osax'` resources and loads any it finds as handlers. After startup, AppleScript checks the modification date of the Scripting Additions folder every time a script is compiled. If the modification date changes, AppleScript will rescan the Scripting Additions folder and load any new scripting additions it finds.

Note:
The Scripting Additions folder scanning mechanism will only load, or reload, handlers for scripting additions added to, or replaced in, the folder. If a scripting addition is removed from the Scripting Additions folder, its handlers can only be removed by restarting the system.

The `'osax'` resources found are not directly installed as handlers, but rather are indirectly called by AppleScript's master loader. A special dispatch handler is installed for each scripting addition handler installed by AppleScript. It is the dispatch handler's job to ensure that when a scripting addition handler is called, the required `'osax'` resource is loaded and ready to be called. The `'osax'` resources are loaded in the system heap, and are marked as purgeable to help reduce AppleScript's memory footprint.

The handler in the `'osax'` resource is identified by its resource name, which encodes the type of handler (event or coercion) and the ID codes needed to install the handler. A source of difficulty when building scripting additions is keeping the resource name and the handler's terminology in sync. You also need to be concerned about resource ID conflicts and other packaging issues. See the AppleScript Scripting Additions Guide: Writing Scripting Additions for more information about the format of `'osax'` resource names.

Being resource-based, the classic scripting addition mechanism has a number of limitations which restrict

how scripting additions can be written and what they can do. Since 'osax' resources are marked as unlocked and purgeable when they are not in use, they can move around or be purged between uses. This means that each 'osax' resource needs to be completely self-contained. Furthermore, since the resources can, and will, be purged from memory, it is more difficult to maintain persistent state between calls.

Accelerated Scripting Addition

In the classic scripting addition mechanism, the only way to create a native scripting addition is to package it as an accelerated code resource (ACR). This method has its own complications and does not address any of the limitations of the classic mechanism. Because of the complications accelerated code resource cause, we strongly ***discourage*** anyone from writing an ACR-based scripting addition.

The problem with ACRs is caused by the way AppleScript loads and prepares 'osax' resources. The classic scripting addition mechanism looks only for 68K 'osax' resources, and since each of these must be self-contained, AppleScript doesn't need to worry about what happens when they are purged. It only cares that the 'osax' resource is in memory before it's called.

It is this lack of concern for what happens when an 'osax' resource is unloaded that is at the root of the ACR problem. Since AppleScript doesn't know about ACRs, it doesn't do anything special when the ACR is unloaded. This means that any connections the ACR has to other CFM libraries it uses are never closed. When the ACR is reloaded by AppleScript, it will open a new connection to the CFM libraries it needs. This causes a small (8-byte) leak in the system heap each time the accelerated code resource is unloaded and reloaded. If the ACR-based scripting addition is used frequently, this leak can cause the system to run out of memory.

Back to top

Native Scripting Additions

A native scripting addition, as the name implies, contains PowerPC code. This code must be in a single PEF container in the data fork of the scripting addition file. The packaging of the executable code is the only difference between a native and classic scripting addition file (other than the 'cfrg' resource). This allows you to build a fat scripting addition by simply including both the classic 68K 'osax' resources and the native code fragment.

Whether native or classic, scripting addition files will contain resources of other types. A terminology resource (type 'aete') is required to provide the terminology for the handlers in the scripting addition. The scripting addition file may also contain any other resources needed by its handlers. A scripting addition size resource ('osiz') is not needed or used by native scripting additions, but may be included if required for the 68K handlers in a fat scripting addition.

The changes needed to support native scripting additions also have the pleasant side effect of allowing AppleScript to remove handlers for scripting additions that are removed from the Scripting Additions folder. This benefit results from the way native scripting additions are written. See CFM termination below for more information.

Note:
The ability to completely unload a scripting addition applies only to the new native scripting additions. Classic scripting additions have the same limitations they've always had.

Back to top

Runtime Environment

A native scripting addition needs to provide routines for:

- CFM initialization

Back to top

CFM Initialization

A native scripting addition needs to define a CFM initialization routine for the code fragment containing the scripting addition's handlers. This initialization routine should check to ensure that any needed services and system resources are available before installing any handlers. If requirements are met, the scripting addition should install its handlers.

The scripting addition's handlers must be installed in the system dispatch table:

```
Boolean isSysHandler = true;

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

Note:
It is critical that no handlers be installed if anything but `noErr` is returned by the initialization routine. A returned error will prevent the code fragment from being loaded, which means that if handlers are installed there won't be any code around when they get called. When this happens, you will see a very spectacular crash resulting in a visit to your friend, MacsBug. One way to ensure that handlers are uninstalled would be to place the removal code in a common routine, which can then be called by both the initialization and termination routines.

In the classic scripting addition model, a flag bit in the `'osiz'` resource indicates to AppleScript that the resource fork of the scripting addition file should be opened before any handler it contains is called. In the native scripting addition model, it is the responsibility of the handler itself to open the scripting addition's resource fork if needed.

The scripting addition's initialization routine should save the `FSSpec` passed to it in the `CFragInitBlock` so the resource fork can be opened when needed. This `FSSpec` is for the library file itself, and can be used later to open the scripting addition's resource fork.

Back to Runtime Environment

Event and Coercion Handling

Scripting additions are a packaging mechanism for event and coercion handlers used to extend the basic capabilities of AppleScript. These handlers are written in basically the same way that handlers used inside an application would be written. What differs between scripting additions and applications is the packaging of the handlers.

In the classic scripting addition model, handlers need to be a self-contained `'osax'` resource. Being self-contained, there is no way to share code between handlers. This often lead to the duplication of substantial amounts of code in each `'osax'` resource.

With native scripting additions, duplication of code is no longer a problem. When a native scripting addition contains more than one handler, those handlers can share common code. Native scripting additions are packaged as a single code fragment containing all handlers, which means there is no need to duplicate code. It is often possible with the native scripting addition model to install the same handler for multiple events or coercions, and vary the behavior as needed based on the input parameters.

Note:
A native scripting addition **must** be built as a single code fragment contained in the data fork. The loading mechanism assumes the data fork contains a single fragment. Data forks containing multiple fragments will result in unpredictable behavior - most likely a crash.

In the classic scripting addition model, a flag bit in the `'osiz'` resource indicates to AppleScript whether or not the scripting addition should accept events from remote systems. This flag is used to provide a level of security by preventing remote systems from performing potentially annoying or harmful actions.

For native scripting additions, each handler 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;
}
```

If the native scripting addition needs to open its resource fork, it must open it after a handler has been called, and close it before the handler exits. It should also have save the `FSSpec` for the scripting addition file, as describe in the CFM initialization section.

```
SInt16    oldResFile;
SInt16    osaxResRef;

oldResFile = CurResFile();
osaxResRef = FSpOpenResFile( iconFileFSSPtr, fsRdWrPerm );

// Do your handler stuff here

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

Note:
When opening and closing a scripting addition's resource fork, it is important to restore the resource chain to the same state it was in when the scripting addition was called. This means that you must save and restore the current resource file around the opening and closing of the resource fork, as shown in the above example.

Back to Runtime Environment

CFM Termination

A native scripting addition needs to define a CFM termination routine for the code fragment containing the scripting addition's handlers. The scripting addition's termination routine is called when AppleScript closes its connection to the scripting addition. This closure will happen when the system is shutdown or restarted. It will also happen when the scripting addition is removed from the Scripting Additions folder by the user.

When the termination routine is called, it must perform any actions needed to close down the scripting addition. One very important termination action is to uninstall any handlers, either event or coercion, that the scripting addition has installed. Failure to do this will most likely result in a crash when a script attempts to call a handler in the scripting addition, since the handler will no longer be available.

```
Boolean isSysHandler = true;

(void) AERemoveEventHandler( theAEEventClass, theAEEventID,
                             theHandlerUPP, isSysHandler );
(void) AERemoveCoercionHandler( fromType, toType, theHandlerUPP,
                             fromTypeIsDesc, isSysHandler);
```

Note:
While it is legal to pass `nil` for the `AEEventHandlerUPP` parameter when calling `AERemoveEventHandler`, and for the `AECoercionHandlerUPP` parameter when calling `AERemoveCoercionHandler`, you should always pass the same `UniversalProcPtr` you used when installing the handler. This allows the Apple Event Manager to do an extra level of sanity checking, which helps ensure that your termination routine doesn't uninstall a handler that some other scripting addition mistakenly installed over the top of yours.

Back to Runtime Environment

Back to top

Further References

- Inside Macintosh: Mac OS Runtime Architectures / CFM-Based Runtime Architecture
- AppleScript Scripting Additions Guide: Writing Scripting Additions

Back to top

Downloadables


Acrobat version of this Note (K).

To contact us, please use the Contact Us page.
Updated: 10-May-99