Apple Event XCMDs

Version 1.0b2

Ed Lai
November 1992

Apple Event XCMDS

Apple Developer Tools Engineering

Introduction

This stack contains a collection of Apple Event XCMDs and handlers which can be used to send Apple events to Apple event-aware applications from HyperCard. It replaces the *SendAppleEvent XCMD* stack which has been previously distributed via AppleLink and Apple's Developer CD. Object specifiers and AERecords are now supported in this stack! With this HyperCard stack, you may easily create and send Apple events to an application. By reading an application's 'aete' resource, which generates glue routines, you can "test drive" the application by sending it events that it supports.

Essentially, this is a mini development environment for scripting other applications using HyperTalk. Since it is entirely implemented in HyperTalk and HyperCard, it may be a bit slower for the low-end machines, but nevertheless, it is a rich environment in which new features can easily be added.

How To Install It

Since some of the Apple Event features are implemented using HyperCard’s built-in Apple Events support, be sure to copy the

appropriate HyperTalk handlers from this stack, as well as the XCMDs and XFCN resources. This is necessary if you plan to create your own HyperCard stack for sending Apple events. Please note that this stack requires HyperCard version 2.1 or later since earlier versions of HyperCard do not have built-in support of Apple Events.

How To Use It

Balloon help is available throughout this stack. This feature will be particularly useful when first learning to use this stack. With the XCMDs and handlers contained in the script of this stack, you can send Apple events to any Apple event-aware application. However, you will first need to know which Apple events and parameters that application supports. This task is facilitated by reading the 'aete' resource of the application, which automatically generates glue routines.

Each application you choose to send Apple events to can have its own card dedicated to it in this stack. When you click on the "Æ Applications" button, located on a number of cards in this stack, you are taken to a card which lists all the applications this stack can send Apple events to. Clicking on an application name in this list will take you to that application's card. To create a new application card, click on the "Add an Application…" button.

To generate glue routines for an application, click on the "Read aete" button on the application card, and then choose the application from standard file. The 'aete' resource from this application will then be read and converted into glue routines. The handler interfaces will be displayed in the top field of the card. The corresponding HyperTalk handlers will be put into the script of the card. Please note that if a handler is greater than 32K, you will be

asked to save it to a file, since HyperCard fields are limited to a maximum of 32K. Following is an example of a typical handler prototype:

```
AEDate ( ["DateOfValue",somelong] [,"format is",{Abbreviated|
Short}] [,"only",{day|time}] [,"asNum",true] )
-- write the date and time
--  DateOfValue: -c seconds; # write date corresponding to seconds
--  format is: ; abbreviated date (e.g.  Wed)
--  only: ; date only or time only
--  asNum: -n; # write seconds since January 1
```

The lines starting with "--" are the comments, with each line describing a single parameter. You can click on the handler name to display a copy of the handler in the "Messages:" field located at the bottom of the card. This field works like a message box for sending Apple events.

Optional parameters are enclosed in square brackets (“[]”). If an optional parameter is italicized, it will be considered inactive and will not get copied to the "Messages:" field. You can toggle between the active and inactive state of an optional parameter by clicking on it. When an optional parameter is in the active state, it will be displayed in plain text rather than italic. An active optional parameter will be copied to the "Messages:" field without the square brackets. Where there are enumerator parameter lists, all choices are separated by a vertical bar (“|”). If you click on one of these items, it is displayed in bold text to signify the active choice. Choosing a parameter through this process is similar to choosing an item from a family of radio buttons. When you copy the handler to the "Messages:" field below, the bolded parameter will be copied too. In this way, much of your editing tasks can be done by simply clicking on the handler prototypes located in the upper field.

Clicking on a command will copy it to the "Messages:" field. If this field contains a selection, the existing selection will be replaced with the handler being copied. Otherwise, the command will be placed beneath any existing text. Please be aware that if you click anyplace on the card, except in the top field, any selection in the message field will be lost.

To specify an object, you may use the object specifier functions or the user interface provided through the "Obj" button. If the application you are examining supports objects, an "Obj" button will appear in the bottom right hand corner of the card. If you

select this button, all of the top level containers in the application will be displayed. Clicking an item in the list will result in going down the container chain until you have fully specified the object or property of interest. A HyperTalk expression will be generated and appended to (or pasted to the current selection of) the "Messages:" field.

As mentioned earlier, the bottom field of this card can be thought of as a large message box. A field is used instead of HyperCard's message box because it is larger, and therefore, makes editing much easier. When you click on the "Do It" button, the script in the "Messages:" field will be executed. If a result is returned, it will be displayed in the same location, but in a different field titled "Results:". If the result is a descriptor of type "PICT", you can use balloon help to look at its contents. Displaying the results in a separate field avoids overwriting your working script. The result field may be a descriptor in the application heap. Clicking on the "Clear" button from the result field, or executing another script from the message field, will dispose of the descriptor automatically. When the

result field is displayed, the "Do It" button is replaced by a button named "Script". Click on this button to view the script in the message field once again.

You may want to save your working scripts for later use. If you click on the "Save" button, the script will be saved as a handler in the "Test" button. This button will hold a number of handlers, so you will be asked to name the handler. If you wish to save it to an existing handler, you may enter the name as "?" to choose from a list of handlers. If you are saving to an existing handler, either by choosing it from a list or entering a pre-existing name, you have the choice to replace or append the script to the existing handler.

When you click on the "Test" button, it will look up the names of all the handlers in the button and show them to you. You may select a handler and then choose to execute, edit or export it to a new stack. If you choose to execute or edit it, the handler will be moved to the top of the list for easier access the next time. If you click on the "Test" button with the option key down, you will be executing the first handler inside the button. This may be a handy shortcut because displaying the list of handlers can be slow.

Selecting "Export" will save the selected handler(s) to a stack. Multiple handlers will be saved to multiple stacks. When you execute a handler, its stack is executed as part of OpenStack. If you ever need to edit such a stack again, hold down the option key when you open it. This will prevent execution of the script. In order for

the scripts to work correctly, it is necessary to have the appropriate XCMDs and glue routines. If each stack were to have its own copy of these XCMDs and glue routines, it would take up a large amount of space. Alternatively, we place all the XCMDs and glue routines into a single stack for each application. If the name of the application is "CoreSample", then the name of the application stack will be "CoreSample glue stack". If the glue stack does not exist, it will be generated automatically when these self-executing script stacks are created. If for some reason you need to generate a new set of glue routines, just throw away the glue stack and a new glue stack will be generated. The script stacks make use of the glue stacks by using the "Start Using" and "Stop Using" commands. If there are other XCMDs you want to use from your script, the glue stacks may be a good place to put them.

Object Specifiers.

The Apple Event Manager allows very general data types, including non-textual data, known as **descriptors**. Descriptors contain a four-letter code representing the data's type, along with a handle to the data. An **object specifier** is a special kind of descriptor used to specify an Apple event object. The XCMDs in this stack have been designed to support descriptors and object specifiers.

A descriptor is heap-based data with no simple textual representation. In this stack, they have the form of "æXXXXæY", where "XXXX" is the descriptor type (such as PICT) and Y is the handle, stored as a number. Descriptors are disposed after being passed on to another Apple event XCMD, until finally, there are no descriptors left on the heap. However, if an operation is aborted, some handles might get left on the heap. A global, **CreatedDesc**, is used to list all such descriptors. Those descriptors remaining on the heap are disposed of during idle time.

Sometimes you may wish to keep a descriptor on the heap to reuse it multiple times. For this purpose, there is an "*Æ*XXXX*Æ*" form of the descriptor which is not disposed of after being passed on to another Apple event XCMD. To generate this more permanent type of descriptor, use the function **AEKeepDesc(theDesc)**. It should be pointed out that this more permanent type of descriptors only exist on HyperCard's heap. As soon as HyperCard quits, these descriptors will disappear. Under no circumstances should you attempt to store descriptors permanently as "*Æ*XXXX*Æ*Y" in a HyperCard field. This will result in the XCMDs crashing if they attempt to use these bogus descriptors the next time HyperCard is launched.

XCMDs and XFCNs.

These XCMDs will take care of most of the functions required in using Apple events. It is even possible to create an Apple Event record representing an insertion location. Please note that these XCMDs have not been tested thoroughly, so use them with caution. Following is a complete list of the XCMDs and XFCNs included in this stack.

AECreateDesc
AECreateList
AECreateRecord
AEDisposeDesc
AEDuplicateDesc
AEKeepDesc
AEObjectSpec
AECompareSpec
AELogicSpec
AESend
AEInstallHandler
AEReadFromDesc
MoveDescToScrap

Following is a complete list of all the relevant Apple event HyperTalk handlers included in the script of this stack. Please copy them to the script of the stack you wish to use for sending Apple events:

AESendMode
exmn
nullDesc
AEChunk
AEPropertyOfChunk
AEAndTest
AEOrTest
AENotTest
AELongType

AEShortType

AEBooleanType
AEPointType
AERectangleType
AEExtendedType
AEHandleError -- used by the generated handlers
AEFreeUnusedDesc
AEStartTargeting
AEStopTargeting
MakeSureLaunched

The following pseudo data types have been included and should prove useful:

"**algn**" is a special pseudo data type without data. It will increase the size of the descriptor until it is word-aligned.

"**pstr**" is used to create a Pascal string. It has a length byte followed by the text data.

Following is a summary of the XCMDs/XFCNs available to support Apple Events in HyperCard:

Function AECreateDesc descriptorType[, type1,data1] … [, typeN,dataN]

This function lets you create an arbitrary descriptor, (i.e., QDpt, long). **DescriptorType** is the type of descriptor you want created. The data is specified as a list of [**type, data**] pairs, which will be concatenated together. The function result is the descriptor record that is created. For example, to create a descriptor record that contains a QDpt with x=25, y=40, you use:

```
AECreateDesc("QDpt","shor",y,"shor",x)
```

This will create a descriptor that is 4 bytes long. The first two bytes represent the integer 40 and the next two bytes represent the integer 25. Note that this function can also be used to coerce a descriptor to another type, so there is no need to provide another XCMD to do coercions. You can do this as follows:

```
AECreateDesc(toType, toType, fromDesc)
```

If you wish, you may write coercion routines such as:

```
function AELongType data
 return AECreateDesc("long","long",data)
end AELongType
```

Then, if you want a long integer of value 25 you can write AELongType(25).
We have put AELongType in the script of this stack, we also have:

function AEShortType data
function AEBooleanType data example: AEBooleanType(false)
function AEPointType data example: AEPointType(200,350)
function AERectangleType data
f**unction AEExtendedType data**

You may add others if you wish.

Note that in this call, as well as all other calls that have the (type, data) pairs, if you have some data that is already in descriptor form, you can always add it as ("****", theDesc).

On AEDisposeDesc theDesc

This handler disposes of theDesc. Both the æXXXXæY and ÆXXXXÆY forms of descriptors will be disposed of.

Function AEDuplicateDesc theDesc

DuplicateDesc creates and returns a copy of the descriptor. Unlike the other Apple Event XCMDs, the source descriptor is not disposed of (which would make this call useless).

Function CreateList elementType, elementSize

The first parameter is the data type allowed in the data list. If any data type is allowed, use "****" as the dataType. If the first parameter is not "****", then all the elements in the list are of the same type and potentially the same size. If the elements are of uniform size, you may append the size to the data type. For example, to create a list of long integers, you would use the type "long4". If the first parameter is not "****", there will be no elementSize parameter.

If the data type is "****", then each element in the list takes up two parameters as a (type, data) pair. If the first parameter is not "****", then type is already specified so each element will only take up one parameter.

For example, the following creates a list containing text and integer elements:

```
AECreateList("****", "TEXT", "this is item 1", "shor", 2, "TEXT", "this is item 3")
```

This creates a list of text strings:

```
AECreateList("TEXT", "this is item 1", "this is item 2", "this is item 3")
```

This creates a list of 5 integers:

```
AECreateList("shor2", 1, 2, 3, 4, 5)
```

This list has two elments, a text string and another list:

```
AECreateList("****", "TEXT", "this is item 1", "****", AECreateList("shor2", 1, 2, 3, 4))
```

Function AECreateRecord key1,data1,key2,data2,...,keyn,datan

Each parameter in the list is a keyword, data pair. The keyword may be a 4 character id, or it may be 8 characters, in which case it is the keyword followed by the data type. Here is how you would create the Create, and Clone events for a window. These event contains an insertion location record.

```
AECreate("cwin","At",
```

AECreateRecord("kobj",NullDesc(),"kpos","bgng"))

```
  AECreate("cwin", "At", AECreateRecord("kobj",      AEObjectSpec("cwin","indx",2"),
"kpos", "befo"))
  AEClone(AEObjectSpec("cwin","indx",5), "To", AECreateRecord("kobj",
        AEObjectSpec("cwin", "indx", 1", "kpos", "afte"))
```

If you prefer, you may add the data type to the keyword id as:
"kobjnull" and "kposenum".

Function AEObjectSpec wantType, form, data[,container]

An object specifier is returned as a result of this function. **WantType** is the class of the object. **Form** is how you want to get it, usually it is "indx", "name" or "test" (in the case of the whose clause). **Container** is the the container of the object. If it is missing, then the default container is used. Any descriptor in the parameter will be disposed of automatically. This is how you would specify "window 1":

```
   AEObjectSpec("cwin","indx",1)
```

And this is "word 2 of window 1":

```
  AEObjectSpec("cwor","indx",2, AEObjectSpec("cwin","indx",1))
```

Function CompareSpec compareType, obj1, obj2

AECompareSpec returns a comparison specifier. **CompareType** is the compare operation, and **obj1** and **obj2** are the objects to be compared. Any descriptors in the parameters will be disposed of automatically. There is a HyperTalk function, **Exmn**(), that returns a specifier indicating the object being examined. Here is how you would specify 'whose first character = "w"':

```
   AECompareSpec("=   ", AEObjectSpec("cha ","indx",1,Exmn()),"w")
```

Function LogicSpec logicType, objSpec1, … objSpecN

AELogicSpec returns a logical specifier. **LogicType** is the logic operation. The **objSpec** parameters are the terms of the logic operation. Any descriptor in the parameter will be disposed of automatically. Following is how you would specify "whose character 1 = "w" and character 3 = "t"":

```
AELogicSpec("AND ",AECompareSpec("=   ", ¬
   AEObjectSpec("cha ","indx",1,Exmn()),"w"),  ¬
  AECompareSpec("=   ", AEObjectSpec("cha ","indx",3,Exmn()),"t"))
```

On AESend event, target, sendMode, timeout[, directObjectKey]
[, directObject] [, key1,data1] ... [, key2,data2]

This handler lets you send an Apple event out to the specified target. If a reply is requested, it will be returned in 'it'. **Event** is an 8 byte ID, and **target** is the target application. The target can be either the name of the application or a descriptor. You can use "*" to denote the default target application specified in the HyperTalk global **curTargetApp**. (See below for more details on the curTargetApp).

SendMode is the sending mode to use in the Apple Event Manager's call, **AESend**. Please look at "Inside Mac, vol. 6" for more details. There is also a HyperTalk function called **AESendMode** that returns a longint that may be used here. You may write sendMode in a more meaningful manner such as:

```
SendMode("WaitReply","AlwaysInteract").
```

The **KeyWord** is either a 4 byte ID or an 8 byte ID plus type code. TEXT/descriptor is the default data type so you only need the 4 byte ID if you want the data to be sent as TEXT/descriptor. Otherwise, it will be coerced to the data type you specified. If coercion fails, the data will be kept as TEXT/descriptor. For **directObject**, no keyword is necessary unless you want to coerce to some other data type. In that case, you need to put in the keyword just for the sake of specifying the data type. All the descriptors will be automatically disposed of. Following is an example of how to write a handler that moves a window in CoreSample:

```
On SetWindowPosition x, y
 AESend "coresetd", "CoreSample", 1, 300, ¬
 AEObjectSpec("prop","prop","ppos", AEObjectSpec("cwin","indx",1)),  ¬
```

```
 "data",AEPointType(x, y)
end SetWindowPosition
```

On AEStartTargeting appName, launching

This handler will make the **appName** application the current default target, which is put into the global **curTargetApp**. This will be the target in AESend when the target name is "*". If the application has not been launched, then it will attempt to launch the application (unless launching is false).

On AEStopTargeting

The current application in the global **curTargetApp** will be removed from the list. The previously default target application will now become the default target application.

Using **AEStartTargeting** and **AEStopTargeting**, you can do the following:

```
AEStartTargeting AppA          -- now AppA is the default target
AESend "xxxxyyyy","*",1,300    -- send to AppA
AESend "aaaabbbb","*",1,300    -- send to AppA
AEStartTargeting AppB          -- now AppB is the default target
AESend "xxxxyyyy","*",1,300    -- send to AppB
AESend "aaaabbbb","*",1,300    -- send to AppB
AEStopTargeting                -- now AppA is the default target
AESend "xxxxyyyy","*",1,300    -- send to AppA
AESend "aaaabbbb","*",1,300    -- send to AppA
AEStopTargeting                -- now there is no default target
```

Function AEKeepDesc(desc)

Takes a descriptor of the form æXXXXæY and converts it into the form ÆXXXXÆY. The

descriptor will no longer be in the global **createdDesc** list.

On AEFreeUnusedDesc

Takes all the temporary descriptors in the **createdDesc** list and disposes of them.

On AEInstallHandler CodeResourceID/Name [resource_type [,handlerType]]

This installs event handlers or coercion handlers that are written as code resource. If the resource is to be loaded in the application heap then it will be loaded as an application handler, otherwise it will be loaded as a system handler. There is a naming convention for resource. The first four letter of the name indicates the type of handler.
'AEVT' - Event handler
'CSDC' - Coercion handler from AEDesc to AEDesc
'CSPT' - Coercion handler from Pointer to AEDesc

If a handler with the same name is already installed, the handler will not be installed. So you need not worry about the same handler being installed multiple times.

If the resource name already follow this naming convetion, and the resource type is 'PROC', then you need to only pass in the resource ID or name

AEInstallHandler 128
AEInstallHandler "CSPTTEXTalis"
AEInstallHandler "*" - install all handlers of resource type 'PROC'

If the code resource is not 'PROC', let's say it is 'HDLR', then you can install it as

AEInstallHandler 128, "HDLR".

If the code resource's name does not follow the naming convention, then you can pass in the name with the correct naming convetion in the following way

AEInstallHandler 128,"PROC", "CSPTTEXTalis"

As an example we have included a Text to Alias coercion handler code resource in this stack.

As another example of how this can be used, we have include a wild card coercion handler "CSDC********". This handler would in turn send an Apple Event to itself (i.e. HyperCard) with the event class 'WILD' and event ID 'COER'. The direct parameter will be the data to be converted, the type is in the keyword 'FROM' and the type to be converted to will in in the keyword 'TO '.

Let us use "cRGB" as an example. cRGB has 3 short integers so it is natural to represent it in HyperTalk as a list of 3 items. So if we want to coerce from "TEXT" to cRGB, the HyperTalk script would look like this

```
On AppleEvent eventClass,eventID,sender
 if eventClass & eventID is WILDCOER then
   request appleEvent data with keyword "TO  "
   if it is "cRGB" then
     request appleEvent data with keyword "FROM"
     if it is "TEXT" then
       request appleEvent data
       reply AECreateDesc("cRGB","shor",item 1 of it,"shor",item 2 of it,"shor",item 3 of it)
     end if
 else pass appleEvent
end appleEvent
```

Function AEReadFromDesc theDesc, offset, dataSize, dataType

The XFCN will read **dataSize** number of bytes from the **theDesc** starting at the **offset**. The data will be intreprets as **dataType**. If the **dataType** is fixed length data types, then **dataSize** should match the length. There is a pseudo data type "byte". You can read 1 to 4 bytes from the descriptor and the data will be an unsigned number.

If we continue our cRGB example, we may want to do cRGB to Text coercion. We can expand our last example using the AEReadFromDesc call.

```
On AppleEvent eventClass,eventID,sender
 if eventClass & eventID is WILDCOER then
   request appleEvent data
   put it into dirParam
   request appleEvent data with keyword "TO  "
   if it is "cRGB" then
     request appleEvent data with keyword "FROM"
     if it is "TEXT" then
       reply AECreateDesc("cRGB","shor",item 1 of dirParam,,¬
        "shor",item 2 of dirParam,"shor",item 3 of dirParam)
     end if
   else if it is "TEXT" then
     request appleEvent data with keyword "FROM"
     if it is "cRGB" then
       put AEReadFromDesc(dirParam,0,2,"shor") into item 1 of x
       put AEReadFromDesc(dirParam,2,2,"shor") into item 2 of x
       put AEReadFromDesc(dirParam,4,2,"shor") into item 3 of x
       reply x
     end if
   end if
 else pass appleEvent
end appleEvent
```

On MoveDescToScrap theDesc

Take the descriptor and puts it into the clipboard.

When we create descriptors, they are passed along to other routines where they are eventually disposed of. This eliminates the need for you to worry about disposing of them yourself. What would happen if we ran out of memory and were unable to generate a descriptor? These XCMDs use the convention that whenever they receive a descriptor of type 'erro' as input, some error has occured and it will not execute. Instead it will free everything up and return a descriptor of type 'erro'. This way, any error will bubble up correctly.

Glue Routines.

With these support functions, you should be able to send a wide range of Apple events. However, it can be made more transparent to the user. We have given some examples of how to build object specifiers using the XFCNs AECompareSpec, AELogicSpec, and AEObjectSpec, but that is by no means an easy task. Hence, we need more glue routines to make it easier. For example, the following three HyperTalk functions can be used on top of LogicSpec:

AEAndTest [test1,test2,... testN]
AEOrTest [test1,test2,... testN]
AENOTTest [test1,test2,... testN]

They just take the test result test1 to testN and perform a logical AND/OR/NOT.

This helps a little, but more can still be done. If you have detailed knowledge of how Apple events are supported by an application, you can build up routines using these XCMDs as primitives. Fortunately, even if you do not know all the details, the application can provide an 'aete' resource which tells us a lot about the application. In fact, we can generate more glue routines specific to an application from its 'aete' resource. Here we will discuss the glue routines that are generated by this stack.

If **xxxx** is the name of the class of an object, then two glue routines, **xxxxObject** and **xxxxProperty**, are generated. For example, for the object class File, the following are generated:

FileObject(whose|index|named, data, container)

FileProperty(Class|Name|File Info|Alias|Stationery|Script System Number, whose|index|named, data, container)

FileObject is used to access a file. The parameters are similar to that of AEObjectSpec. FileProperty is used to access properties of the file object. The first parameter is the name of

the property, and the remaining parameters specify

the file (same as FileObject). To illustrate how this works, here is the 2nd word of window 1:

WordObject(index, 2, WindowObject(index,1))

If this is still too difficult to read and you don't need a whose clause, then there is a HyperTalk Function **AEChunk** that can be used on top of these functions.

Function AEChunk [objectName, form, data] [,objectName, form, data]...

Now, the 2nd word of window 1 becomes:

AEChunk(word, index, 2, window, index, 1)

Function AEPropertyOfChunk property, [objectName, form, data]
[,objectName, form, data]...

AEPropertyOfChunk is similar to AEChunk except that there is an additional parameter of the property name.

So the font of the 2nd word of window 1 is:

AEPropertyOfChunk(font, index, 2, window, index, 1)

We can also generate HyperTalk handlers from the 'aete' by asking another application to do some operations using Apple events. The user should not be asked to remember four byte keywords/enum/property/class, etc. Ideally, if we want application X to move its window, there should be a handler MoveWindow to do the job without requiring the user to know the details of Apple events. Again, the 'aete' resource provides a lot of information about the parameter so that such glue routine can be generated automatically.

Each glue routine will have the following conventions. It will either be a function or handler, depending on the result type in the 'aete'. It is then followed by all the required parameters. If the direct parameter is optional, it will be the first parameter after the required parameter. That will be the end of the positional parameters. The rest of the parameters are the optional parameters, which will be in (keyword, data) pairs. The keywords and enumerators can either be four letter codes or the full names used in the 'aeut' resources.

For example, here is the prototype for the glue routine for the ToolServer call Duplicate:

```
AEDuplicate directParam ,onto [,"Confirm",yes|no|Cancel]
[,"only",datafork|resourcefork]
```

To duplicate the data fork of file "My File" into file "My Copy" you just call

```
AEDuplicate "My File", "My Copy", "only", "datafork".
```

With this convention, there will be a HyperTalk handler for every single Apple event supported by the application. One problem, however, is that events that are inherited from the AEUT will all have the same name. Thus, there will be AESetData for many different applications. Because the glue currently hard codes the target application, there is potential for alot of conflicts. To avoid these conflicts, when we generate the glue routines, if we find an event that is also in the 'aeut', we shall set the target to be "*" so that it can be used with AEStartTargeting.