Scriptability:
A Bare-Bones Introduction

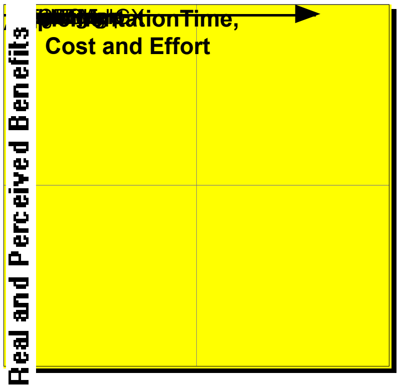
Kevin C. Killion
Stone House Systems, Inc.
kevin@shsmedia.com

Set aside your IM, AE Registry, and tech notes for the moment. Starting fresh, let's review the bare essentials you need t start making your app scriptable. Simple things should be simple, complex things should be possible, so let's ignore the rampant excesses of the AppleScript spec for now. This paper will introduce scripting with a simplified explanation of t concepts, and follow this up with implementation of a minimal, but very useful initial set of scripting functions. As an added bonus, we'll consider how to design your implementation for accessibility from BASIC as well as AppleScript.

An unprecedented number of new sets of APIs have been added to MacOS in the last few years. As of this writing, some 35+ different add-on technologies are provided by Apple in the MacOS SDK. Some of these technologies give developers the ability to implement exciting new features with a minimum of new coding effort. The Thread Manager, for example, enables an entirely different approach for handling long processing times without delaying the user, and it does so with only a few simple API calls. QuickTime has been another unqualified hit: simple to implement, and yet an incredible crowd-pleaser with customers.

Other new technologies released by Apple have not been so successful. Take AOCE: Many of the features it made possible were hard to explain, or passed minimal real benefits through to the end user. Nonetheless, documentation was vast and dense. It was hardly a surprise when Apple would say (in the January 5, 1996 issue of MacWeek) that it was "confusing to use" and that the entire API would be thrown out.

We might make an attempt to categorize MacOS add-on APIs with a cost/benefit approach. Every developer will have a different view, of course, depending on his or her own interests and target markets. For myself, I'd view the MacOS APIs this way:



This leads to a kind of technological triage. Some technologies are brainers: if an application can make any use at all of QuickTime, it should. Other toolkits are easy to dismiss: the documentation and methods for QuickDraw GX, for example, are so turgid and involve that only a high-end graphics program is likely to benefit.

For many developers, AppleScript belongs firmly in the upper right quadrant: high benefit at a high cost. AppleScript offers tantalizing power and flexibility, and could lead an application into a collaborative world in which the whole is much greater than the sum of the parts. On the other hand, the task of making an app scriptable appears daunting and formidable. Dave Mark says, "Many people (myself included) are intimidated the first time they try to take on the

Apple Event Manager. In many ways, this task is as complex as when you first learned to work with the Mac Toolbox."

Why Implement Scripting?

To start taking AppleScript seriously, we must first reinvigorate ourselves with the potential that AppleScript offers.

1) At the very least, AppleScript serves as a macro language that is far more powerful than most any scratch-baking scripting facility that a developer is likely to dream up on his or her own.

2) The principal language offered by Apple's scripting facilities leaves a powerful first impression: Who would not be charmed by a programming language that looks so much like English?

3) AppleScript crosses application borders. It goes where no application-based macro language can go: into another application.

4) It adds panache: scriptability is a recognizable marketing advantage. Even if a customer or prospect has no clue about what AppleScript looks like or what it's for, they probably perceive that having it is a good thing.

Beyond these, AppleScript provides the developer with additional benefits, which may not be as obvious:

5) Scriptability invests the user with a sense of personal participation, and that can only be good for long-term commitment to a product. Just ask your six-year-old: the favorite toys are the ones that offer lots of play value.

6) That sense of empowerment and entitlement not only adds your customer to your development team, but to your sales team as well. An "enlisted" customer can generate powerful positive word-of-mouth advertising.

7) AppleScript is an avenue for addressing outlier requests. I have often had key customers request some unusual feature as their "gotta have" top priority, even though other customers would see such a feature as worthless menu clutter. AppleScript provides the perfect solution.

8) Scripting reduces conflicting pressures on development, by giving users the power to implement new features themselves.

9) Ofttimes, merely *having* that power is satisfying to customers, even if it isn't *used* immediately! One of the great strengths of Microsoft Excel is the perception that one can do anything with it, no matter how convoluted; whether someone would ever actually undertake that effort is a question that is easily overlooked.

10) Scripting can be a profit center. A customers may place high val on having a custom report or process developed, made possible and practical by AppleScript.

So, Why Not Implement Scripting?

It isn't too hard to understand why developers have ignored AppleSc in droves, despite the inherent appeal of what it could deliver.

Technical Obstacles to Scripting

1) The sheer vastness of the documentation: "Inside Macintosh: Interapplication Communication" is a thousand pages of intense and cross-dependent text. The "registry", without which we are warned to proceed, adds more desk ballast.

2) No high-level interface: One might argue that the paucity of simplified, high-level API calls may be the single greatest technical deficiency of the Mac toolkit *in general* versus its modern competit AppleScript is no exception. In its simplest description, a scripting facility should provide a developer with the ability to expose the "objects" of an application and make them available for manipulatio by an external process. This kind of basic implementation should b possible with a reduced set of simplified calls. (It is true that some l of scripting support has been built into some class libraries. Howeve since each such class library is developed independently, there is lit portability between them. Of course, other class libraries and procedural projects are left out in the cold.)

3) Lack of a "start here" subset: Implementing the four "required" AppleEvent handlers gets developers on a roll. At that point, development comes to a screeching stop in the face of the massive documentation. There is no sense given that there is any valid subset of calls and features with which to get started on implementing the juicier parts of scripting. (As we will see, it *is* possible to start with a decent, manageable subset.)

4) Minimal sample code: As of this writing, the developer CDs offer only three examples of implementing scripting in a program.

5) Spaghetti sample code: "Quill" was an early piece of sample code that was not only unfinished (pretty scary) but which suggested unbelievable complexity nonetheless.

6) Aren't-I-wonderful: In technical writing, awe of the AppleScript architecture too often gets in the way of clear exposition of what a developer needs to do. Rather than identifying specific development steps, writing has often dwelled on terminology and theory.

7) The hidden power of the AppleScript language: The syntax of the language does not appear (on first blush, at least) to have much of a one-for-one correspondence with the code that must be built in the app. Thus, the *purpose* of each code element remains elusive. ("Uh, you mean I have to 'install' an 'object accessor'? What for?")

8) Disinterest in the AppleScript language: Some developers think AppleScript is too cutesy to take seriously. Other developers (myself included) feel that AppleScript has more serious problems as well.

9) The 'aete' resource: This messy and convoluted resource is the linchpin of scripting, and dealing with it cannot be avoided.

Marketing Obstacles to Scripting

1) The promise of collaborative applications only works if there is someone to talk to. With the conspicuous and embarrassing exception of Microsoft applications, a comparatively small portion of MacOS apps offer serious scriptability.

2) The promise of a common scripting language is often not borne out. The president of one well-known Mac publisher says that "There isn't just one AppleScript. There is AppleScript for Excel, AppleScript for ClarisWorks, and so on. Each variation sports its own terminology and set of available verbs. Even products within a category, such as word processors, can have markedly differing scripting dictionaries."

3) There is no guarantee that the effort involved in adding scriptability will pay out in terms of customer satisfaction. Part of this is because…

4) The portion of Mac users who do any kind of AppleScripting tod is vanishingly small. In my own survey of my major customers, wh turn have thousands of end users and a number of technical support groups, I failed to find a lead on even a *single person* who had *any* experience at all with AppleScript.

5) The challenge of mastering AppleScript: With few books for supp and no similar prior experience to build upon (with the exception of HyperTalk for a few folks), learning AppleScript is not simple. Although many users already have experience with BASIC, which i many respects is a far more powerful and flexible language, this experience doesn't help much when learning AppleScript's rather unique approach. Moreover, the allure of an "English" syntax soon dissolves into the reality of stilted, run-on sentences with peculiar structures.

6) Dictionary shock: From my own experience, I have seen how customers react with a glazed expression when first seeing scripting dictionaries.

7) Support costs: While scriptability may eventually deliver benefit you and your customers, the initial learning hump might impact you short-range support costs. When users run into trouble writing AppleScript statements, who else do they have to call but you? With few sources for AppleScript information or expertise (contrasted wi BASIC, for example), there is simply nowhere else to turn.

Does This Sound Like You?

The growth of AppleScript and scripting has languished over that co benefit issue: The benefits are irrefutable, but the cost of developm is often seen as insurmountable. But if implementation can be made manageable and affordable, the scriptability becomes a much more viable option.

AppleScript divides Mac developers into two camps: the mystical and the mystified. Some developers have bought into scripting completely, and are aglow with fervor for the Object Model, the OSL, the terminology resources and the natural language interface. *The rest of this article is not for this enlightened group.*

This article is intended for those who remain perplexed by scripting. I intend to provide a "start here" approach for adding scripting to your application. If your perspective of scripting matches the outline presented in the text up to now, keep reading.

"Scripting" versus "AppleScript"

A key step is to understand the difference between "scripting" and "AppleScript". When you make your app scriptable, *you are doing nothing that has anything inherently to do with AppleScript!* AppleScript is merely a language provided to access the scripting facilities of the system and applications.

The features and strengths of AppleScript often bear little relation to the code you write to support scripting. Moreover, there is nothing to prevent other languages from being developed to drive scripting. In fact, provision for such languages is built into the Mac scripting toolkits. Some folks love AppleScript, but not everyone. If you're in the latter group, then consider scripting anyway and hope that alternative scripting languages emerge.

Objects and Properties

Every application deals with a set of "things". An accounting system has clients, accounts, transactions, reports and many other elements. A model of some industrial process may have inputs, processing stages and outputs. A transportation management system may have parcels, containerized shipments, sources, destinations, warehouses and vehicles. (I'm bored to tears with Apple's fixation on text and pictures, so I like to use other examples.)

Of course, there is nothing novel about the concept of objects and properties. As programmers, we are used to the distinction between methods and instance variables in OOP. Even users have some sense of the distinction; for 12 years now they have dealt with screen "objects" and manipulated their "properties" through dialogs and commands.

(The original Xerox workstations crystallized this philosophy with a specific "Props" button on the keyboard. The user selected a screen object, and pressed Props to access a dialog to change the properties of an object. As time goes on, the brilliance of the Xerox design becomes even more evident. Much of what we admire in OpenDoc already was fundamental on the Xerox workstations of 1981!)

It's important to note that a crucial way of identifying the objects in application is to tell whether they can be created or removed. A shipment can be removed from our transportation system, but we cannot "remove" its color. The shipment is an object, color is one of that object's properties.

Of the tasks we envision enabling through scripting, an enormous portion involve *reading* and *setting* the values of these properties. W just these two actions, the user can create custom reports, transfer selected information to other applications, tweak settings to improv some measure of performance, and do many other things.

The Role of Context

Six-year-old on phone: "Do you want to come and play at my house
Classmate: "Where's your house?"
Six-year-old (puzzled): "Here!"

User: "Tell me the weight of the shipment."
Computer: "Which shipment?"
User: "This one!"

In an interactive, live program, direct pointing can identify the "thin to be discussed. But that obviously isn't sufficient for a batch-style, hands-off operation. Moreover, there are plenty of situations in whic script may wish to talk to elements that aren't even part of the visibl interface.

Consequently, "this one" is not a sufficient direction to a target. To identify this target, a descriptive address is needed, and it's natural to do that in terms of the context within a larger entity.

Parent of classmate: "Where's your house?"
Parent of six-year-old: "We're the third house on the block, on the block nearest the school, on the east side of the town."

Computer: "Which shipment?"
User: "The one stamped '47', in Bin C, on the upper shelf, of the truck that left at 10 am"

These addresses identify the object of interest with a series of successively larger contexts. Note that each context is an object in itself: house, block, side of town, the town, bins, shelves, trucks these are the objects, the things, of their respective applications. And each step in establishing context describes how one object is referenced *with respect to* another.

Context vs. Containment

Mac scripting documentation describes this relationship as *containment,* which is a very useful and descriptive name. However, we must quickly make note of what containment is *not:*

1) "Containment" is not exclusive: a cell in a spreadsheet is contained by a row but it's also contained by a column.

2) "Containment" does not necessarily mean physical enclosure: In our transportation program, a roadside weigh station might be used as a context within which to identify the tenth truck weighed today. The weigh station does not "contain" the truck in any English sense of the word, but it does provide a context for identification.

3) "Containment" does not necessarily involve visible entities on the screen. More generally, scripting certainly should be available even if there are *no visible entities* on screen at all; we might desire to "operate" a faceless application.

4) "Containment" does not necessarily correspond to subclassing in a class library: "box" and "letter" may be subclasses of "parcel" within our app, but we could conceivable desire to identify and handle boxes and letters in entirely different ways in our user scripting facility.

5) "Containment" does not necessarily correspond to supervisory relationships in a class library: Clicking on a client name may involve a class library's handling of a large pane as supervising the subpane containing the client name, and a document supervising the large pane. For our scripting purposes, "containment" may instead refer to the client's position on a list of clients, which isn't involved in the class bureaucracy at all.

So, while "containment" is a handy term in our scripting efforts, be aware that, more than anything else, it is merely a way to establish *context for identification of objects.*

Are Properties "Contained"?

A parcel does not "contain" a weight, a spreadsheet cell does not "contain" a font size, and a client does not "contain" a name. Nonetheless, when we understand that the role of "containment" references in scripting has more to do with *context* and *identificatio* than it does with enclosure, certain similarities do emerge.

"Change the color." "Of what?" "Of the third row."

In this example, reference to the object identifies, in a very real sens the context of the value to be changed. Understanding this subtlety make it easier to understand the syntactic similarity of property references and object references in both AppleScript and other scrip languages. Later on, this will also help to explain why the same use routine winds up being useful for identification ("resolution") of bot objects and properties.

A Make-Believe Scripting Language

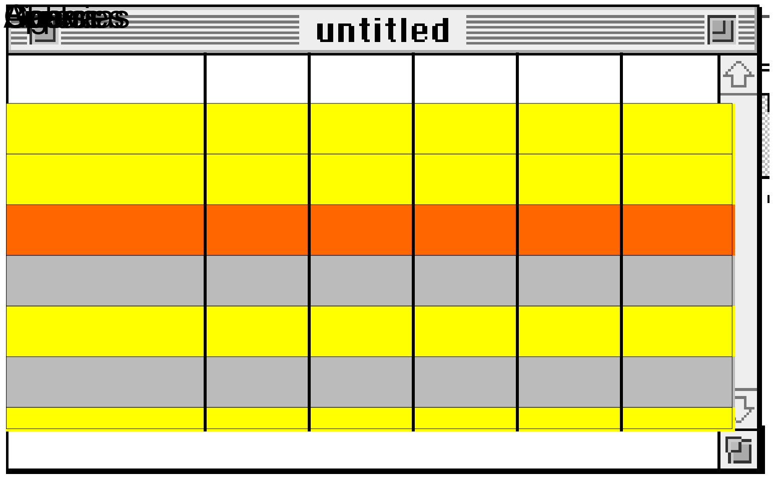
To get our heads around where scripting is positioned, and why AppleScript sentences look like they do, let's do a thought experime

Let's imagine that AppleScript doesn't exist. Then imagine that we were given the task of designing a scripting language from scratch. Further suppose that we buy into the concepts of:

objects as the entities within an application
objects have properties to be examined and set
objects are identified within contexts of other objects

Now, what structures might we place into our new language to implement these concepts?

To illustrate the task, let's suppose we have a simple tabular document, with a number of table rows that can be colored individually.



First, it seems natural to allow language variables to refer to properties of the objects in our application. A first try at devising such a scheme in BASIC, for example, might involve calling some subroutine that retrieves the needed value.

```
mycolor = GetValue("color")
  ' "color" is a pre-arranged keyword
```

We immediately see that we can't talk about *color* without talking about *of what*, so we also need some way to denote context. Let's next imagine another available subroutine called GetObject that provides a reference to an *object*, when we ask for a specific entry on a list. We then could expand our GetValue routine to use this reference as a context. We then have:

```
thisrow = GetObject("row", 4)
  ' get the fourth row ("row" is a pre-arranged
keyword)
mycolor = GetValue(thisrow, "color")
  ' get its color
```

We're making progress, but to get that row, we need to reference *its* context. A row only has meaning within the context of a document. In turn, the document is only one of several documents that may be open, but the context there isn't as clear. We might decide to denote this "top level" or "global" context with "NIL". Then:

```
thisdoc = GetObject(NIL, "doc", 1)
   ' get the first document
thisrow = GetObject(thisdoc, "row", 4)
   ' from the doc, get the fourth row
mycolor = GetValue(thisrow, "color")
   ' from this row, get its color
```

In a very simplistic form, note that all of our references constitute a *series* of *pairwise* operations:

- within the "global" context, identify a document
- within the document context, identify a row
- within the row context, reference the color property

Now let's play language designer. Rather than clutter our programm with lots and lots of subroutine calls, let's devise some new syntax elements to streamline the coding.

Since everything we've done involves these pairs, we might quickly devise a syntax that combines the two elements of *context* and a "contained" *object or property.* A simple dot may serve this purpose

```
{context}.{contained item}
```

For convenience, we might allow skipping any top-level "NIL" reference. With these changes, the previous listing could then be simplified as:

```
thisdoc = doc(1)
  ' get the first document
thisrow = thisdoc.row(4)
  ' from the doc, get the fourth row
mycolor = thisrow.color
  ' from this row, get its color
```

Again, the pairwise nature of these references is evident. But for th user's convenience, we could allow these statements to be strung together, so that the intermediate variables aren't required:

```
mycolor = doc(1).row(4).color
  ' get color of 4th row of doc #1
```

While this new, crisp syntax looks more complex, it's easy to imagi that processing of it still relies on those good ol' pairwise steps. (An lo and behold, what we have just devised is *not* an imaginary langua With small variations, this is the syntax used in Microsoft's Visual Basic family of languages.)

Now let's take a look at another language, namely AppleScript:

```
get the color of row 4 of document 1
```

Again, it's a long statement, but the fundamental notion is that processing this will involve a series of ever-finer context resolutions.

Prerequisites for Coding

Although this is intended to be a "bare bones" introduction, we will require a few prerequisites before getting into the coding.

First, we will assume that you have already implemented the four "required" AppleEvents, namely, Open Application, Open Documents, Print Documents, and Quit Application. Supporting these events are already well-documented, and are comparatively straightforward. If by some chance you have *not* implemented the required events, start there. They are very well described, complete with code examples, in "Inside Macintosh: Interapplication Communication", pages 4-11 through 4-20.

As a by-product of implementing the required events, we'll also assume you have some notion of the idea of AppleEvents acting as containers to transport "clumps" of data. Since you needed to "install" your "handlers" for these required events, we'll also assume that you're familiar with those buzzwords.

To initialize the scripting facilities for your program, you must call the `AEObjectInit` routine in your application's initialization:

```
err := AEObjectInit;
```

Of course, you should always check the result of this and all other calls that return an error parameter. In this text, we will skip that for clarity. However, error handling is implemented in the listing accompanying this article.

Set Data and Get Data

We are now ready to implement the first two crucial "handlers" to enable scripting. The "Set Data" and "Get Data" AppleEvents are the messengers that allow a user to retrieve and change properties of application objects. They are the heart of scripting.

In concept, their function is very simple. Let's start with the Set Data event (see figure, below). After you have created and installed your handler routine for Set Data, your application is prepared to receive these events. Each Set Data event contains two parameters: First, the parameter identified as "keyDirectObject" contains a reference to the property that is to be changed. Second, a parameter labelled "keyAEData" contains the new value for the property.

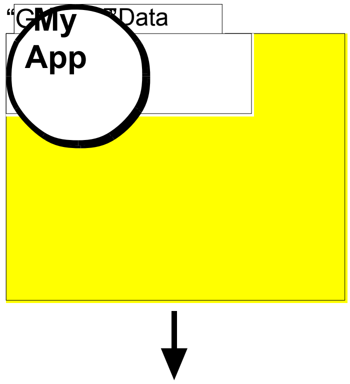


Figure 1: The Set Data event

Our code for this event involves retrieving these two parameters, figuring out what internal variable in our program corresponds to th property that has been specified, and then changing that variable's value. (For a complete implementation example of code used in this paper, see the listing, "Scripting.p". This sample listing is partially based on Apple's "Quill" example.)

Handling the Get Data event is similar (see figure, below). In this ca the AppleEvent that has been received has only one parameter, a reference to the object and property of interest. This is exactly the sa as for the Set Data event. The big difference is that when we access corresponding variable in our program, we retrieve its existing valu and send it back to the calling application. We do this by packaging this value as a parameter and attaching it to the reply AppleEvent.

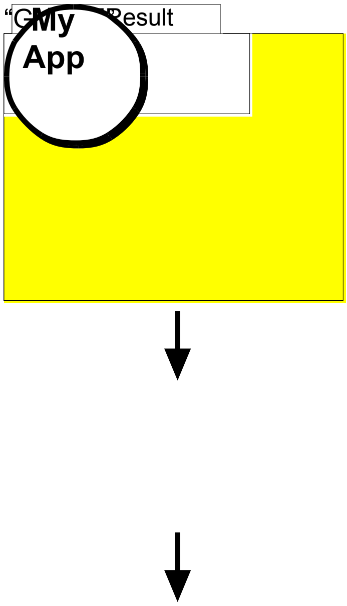


Figure 2: The Get Data event

So far, it hardly sounds difficult at all, does it?

We tell the system that we have Set Data and Get Data handlers by installing them as follows in our application's initialization area:

```
err := AEInstallEventHandler(kAECoreSuite,
kAESetData, @HandleSetData, 0, FALSE)
err := AEInstallEventHandler(kAECoreSuite,
kAEGetData, @HandleGetData, 0, FALSE)
```

Like all AppleEvent handlers, the calling sequence for our routines consists of an AppleEvent being received, a possible reply event, and a reference constant:

```
FUNCTION HandleSetData (theAppleEvent: AppleEvent;
reply: AppleEvent; refCon: LongInt): OSErr;
```

The reference constant makes it possible to handle several different kinds of events with the same handler. For example, you could conceivably have a single routine that handles both Set Data and Ge Data by installing the same routine for both functions, and telling th apart with that constant. However, for code clarity, I'd recommend ignoring the refCon and coding each handler separately.

Within each of these handlers, we retrieve the direct object (the obje and property to be accessed) as follows:

```
err := AEGetParamDesc(theAppleEvent, keyDirectObje
typeWildCard, myDirObj);
```

We now have the reference to the property to be retrieved or set, contained in the variable `myDirObj`. This variable is declared as t `AEDesc`, which has mystical trappings to it but which really is noth more than a handle with a 4-character label (a `DescType`) on it:

```
TYPE AEDesc = RECORD
   descriptorType:  DescType;
   dataHandle:  Handle;
END;
```

Ah, but what's in that handle? For initial scripting efforts, it is suffic to state that this handle contains an encoded representation of the en specification. For our figures, that specification is "font of row 1 of document 1", in a compact binary form. (No, it is *not* the literal text this specification.)

This is where the fun starts. It might be possible to parse the conten this handle ourselves, determining the structure and successive "containments" implied. Mull that over for a while, and think about how you would do that. Besides the effort involved in decoding and parsing the message, we would have to do a great deal of bookkeep along the way.

Now recall our earlier lengthy discussion about context and containment. We stressed that every step of the resolution involved pairing of a *context* and an *object or property* to be isolated from wi that context. What if we could reduce the problem of identifying tha direct object into a *series of simpler requests* of the context/object form?

HARD: decipher "font of row 1 of document 1"
SIMPLER: Do each of the following:
 - find document 1
 - find row 1 within it
 - find the font of that

This process of breaking down a complex specifier into a reference of a specific object or property is called "resolution". The Mac scripting architecture makes resolution much easier by handling the bookkeeping chore for us. To do this, we are given a function called `AEResolve`. We call `AEResolve`, and hand it the complex specification. The system will make additional calls back to our application, consisting of a series of simpler resolution questions, like those listed immediately above. When `AEResolve` returns, we are given a reference (of our own design and in terms of the application's internals) that uniquely identifies the target object or property.

The sheer power of this approach is that AppleScript and the scripting engines within the system contain a great deal of leverage to manipulate complex expressions on their own, requiring our application to deal with only these simple context/object pairing questions.

If you've been paying close attention, you've noticed two leaps of faith in the above discussion. First, we said that the system calls back to the application to ask these pairing questions, but we have not yet said how. Second, we said that `AEResolve` returns a reference to some internal variable that is to be retrieved or set, but we have not yet said where that comes from. We will cover those omissions in the next sections.

For now, let's continue to operate on faith. We call `AEResolve`, and we retrieve a spec for the application internal to be accessed. This is another `AEDesc`, called `newDesc` in this sample. `newDesc`'s handle contains a structure that is our own device for referring to goodies in the application. We extract this with a simple BlockMove, placing the result in a structured variable we call `myToken`:

```
err := AEResolve(myDirObj, kAEIDoMinimum, newDesc);
BlockMove(newDesc.dataHandle^, @myToken,
myTokenSize);
```

The few statements we've examined so far are identical in both Set Data and Get Data. Now, these two handlers will diverge. Consider what each must do:

Set Data:
1. Retrieve the value specified for the application internals
2. Attach it to the outgoing "reply" AppleEvent

Get Data:
1. Extract the desired new value from the incoming AppleEvent
2. Set the correct application internal to this value

There is an obvious symmetry to setting and getting data elements. Wherever possible, it's desirable to handle such similar functions wi the same code, to avoid them getting out of sync and to avoid largel redundant code. For example, many programmers like to have a sin I/O routine for both reading and saving documents.

We will do the same here, bottlenecking both our Set Data and Get Data procedures through one routine, called DoTransferProperty. In each case, this routine is passed our "token" reference, and the relev AppleEvent (either the incoming or outgoing one).

For our Set Data handler, we give our handler the token and the incoming AppleEvent:

```
VAR direction: (doSet, doGet);
...
direction := doSet;
err := DoTransferProperty(direction, myToken,
theAppleEvent);
```

The Get Data handler is passed the token and the outgoing reply AppleEvent:

```
VAR direction: (doSet, doGet);
...
direction := doGet;
err := DoTransferProperty(direction, myToken, repl
```

We are now very close to having our application take its first steps towards scriptability. All we need to do now in our code is fill in the steps we've taken on faith. These missing steps are:

1) Devise a means to uniquely identify variables in our system, usin "tokens".

2) Provide the callback routine that `AEResolve` will use to ask the context/object questions, returning a specific token.
3) Implement the `DoTransferProperty` routine.

Besides these code changes, there is one more step: we must create an 'aete' resource.

Defining "Tokens"

Much of the existing document and sample code for scripting correctly points out that the contents and usage of tokens are totally up to you. Unfortunately, they often then go on to define very convoluted token strategies for sake of illustration.

I have found that to get started with scripting, a very straightforward token design is really quite sufficient.

The tokens passed between the system and your application consist of `AEDesc`'s which, as we've said, are nothing more than handles with four-character labels. What you put in that label, and what you put in that handle are up to you.

It is conceivable that a very simple application may be able to define all of its objects and their properties with a coding that uses the label alone, and leaves the handle at NIL. Suppose we have only the app itself, and a single tabular document with no more than 99 rows and columns (identify them as "A", "R", and "C"). Further suppose that no object has more than 9 unique properties, and we will let "0" represent the object itself. Then, we could use codings like this:
R021 = the first property of row 2
R020 = row 2 itself
A001 = property #1 of the application (some global setting)

That just shows how simple a token strategy may be. A more realistic token design will create a structured record identifying the object or property. In the accompanying listing, this is the token definition used:

```
TYPE
MyTokenType = RECORD
  myTokenCode: DescType;
  theObject: CObject;
  subReference: longint;
  isAProperty: Boolean;
  propertyCode: DescType;
 END;
```

Here is how I use each of these fields in this record:

`myTokenCode`: this is just a simple code to identify the kind of object being discussed. The code used here is only meaningful to the application itself, of course, so it can be anything.

`theObject`: this is a handy place to store the handle containing th most relevant object. For example, from TCL, this field could be se the actual object of concern.
`subReference`: I defined this field for convenient referral of application elements which are not "objects" in the class hierarchy themselves. For example, I may have an object "client" which possesses an array of values representing billing dates. With theOb referring to Smith Company and subReference 3, I can resolve dow the third billing date for this client. If subReference=0, the token ref to the object itself.
`isAProperty`: This same token structure is used for both objects their properties. For an object, isAProperty=FALSE, for a property TRUE.
`propertyCode`: another four-character code, of my own device, identifying the needed property, if isAProperty=TRUE.

I have found that this simple token design satisfies all of the needs I have encountered for basic scripting capability. Remember -- the specifics of the design are totally up to you.

Answering Context Questions:
Object Accessor Routines

As we have said, the system's `AEResolve` procedure can be given complex specification of a series of containments and a final object property, and give back to us a reference to a specific application variable or element. Clearly, this has a nice magical feeling to it.

`AEResolve` works by breaking the resolution task into a series of simpler questions. Each of these questions takes the form, "Within *context*, identify a particular element". We accept and handle these questions by registering one or more *object accessor* routines.

The Apple documentation starts off with the assumption that you'll want to have several object accessor routines to handle different cases. I'll simplify things by using only a single accessor function. (Quite frankly, I think an application can survive very nicely, and keep its code more readable, by using only one.)

We install an object accessor with a simple call at initialization time:

```
err := AEInstallObjectAccessor(typeWildCard,
typeWildCard, @MyObjectAccessor, 0, FALSE)
```

The two "typeWildCard" parameters tell the system that the single MyObjectAccessor routine should be called for *all* resolution questions.

The accessor routine has this calling sequence:

```
FUNCTION MyObjectAccessor (desiredClass: DescType;
containerToken: AEDesc; containerClass: DescType;
keyForm: DescType; keyData: AEDesc; VAR theToken:
AEDesc; theRefCon: longint): OSErr;
```

This calling sequence includes:
- a reference to the container, that is, the context (`containerToken`)
- the kind of element to be identified within the context (`desiredClass`). For example, for an accounting system, a context of a "client" may include both "account" and "transaction" elements; we need to know which is being referenced)
- a reference (`keyForm` and `keyData`) for how to find the correct element of the desired class
- an AEDesc into which we are to place our "token" referring to the desired application element.

If you do use the single accessor routine strategy, then you'll dispatch different cases of the question yourself. One clear differentiation is between context/property questions and context/object questions. If the system is looking to find a specific property for an object, your accessor is called with `keyForm = formPropertyID`. In the code listing accompanying this article, we have separate routines to handle accessing properties, and identifying objects contained within other objects:

```
VAR
 err: integer;
BEGIN
 IF keyForm = formPropertyID THEN
  err := PropertyAccessor(desiredClass,
containerToken, keyData, aTokenBody)
 ELSE IF containerClass = typeNull THEN
  err := AppObjectAccessor(desiredClass,
containerToken, keyForm, keyData, aTokenBody)
 ELSE IF containerClass = 'docu' THEN
  err := DocObjectAccessor(desiredClass,
containerToken, keyForm, keyData, aTokenBody)
 ELSE
  err := errAECantHandleClass;

 IF err = noErr THEN
  err := AECreateDesc(desiredClass, @aTokenBody,
myTokenSize, theToken);
 MyObjectAccessor := err;
END;
```

Note that if the desired element is contained by the application itsel the `containerClass` is `typeNull`; this is the "top" of the containment hierarchy. Also note that if we cannot provide the need resolution, we must return an appropriate error code, such as errAECantHandleClass. This will let the system give an appropriat friendly message to the user.

Let us now look at the resolution of properties of objects, and then objects within objects.

Resolving Properties

Our resolution handler, `MyObjectAccessor`, calls another routi `PropertyAccessor`, to resolve a specifier down to a specific property of a specific object. The routine is given a reference to the object, we must return a reference to the property. The question her of the form, "Tell me how you'd like me to refer to the weight of thi package".

Fortunately, given the way we defined tokens, this is extremely eas We extract the token representing an object. Properties of this objec will be referenced with essentially the same token, just by setting `isAProperty` to TRUE, and setting the `propertyCode` field. T only complication is when we are asked about properties of the application itself (typically these are global settings or values). In th case, the containing token is '`typeNull`', and we must fill in the fi to note that the "object" under discussion is the application.

```
FUNCTION PropertyAccessor (desiredClass: DescType;
containerToken: AEDesc; keyData: AEDesc; VAR
theTokenBody: MyTokenType): OSErr;
VAR
 propertyCode: DescType;
```

```
PROCEDURE Bail (bailErr: integer);
 BEGIN
  PropertyAccessor := bailErr;
  EXIT(PropertyAccessor);
 END;
BEGIN
 IF containerToken.descriptorType = typeNull THEN
  BEGIN   {container is the app (which doesn't have a
token of its own), so make a token for the property}
   theTokenBody.myTokenCode := typeNull;
   theTokenBody.theObject := NIL;
   theTokenBody.subReference := 0;
  END
 ELSE IF NOT GetTokenFromAEDesc(containerToken,
theTokenBody) THEN
  Bail(eContainerDoesNotHaveValidToken);

 BlockMove(keyData.dataHandle^, @propertyCode, 4);

 theTokenBody.isAProperty := TRUE;
 theTokenBody.propertyCode := propertyCode;
 PropertyAccessor := noErr;
END;
```

Note that the object is specified with an `AEDesc` called `containerToken`. We call a little utility routine `GetTokenFromAEDesc`, which typically simply extracts one of our tokens from the handle of the `AEDesc`.

We have been given a code for the desired property in the `keyData` parameter. We use that value to set our `propertyCode` field.

Resolving Objects

Finding an object "contained" within another object requires a bit more effort. However, once you get past the basic requirements of this step, you'll quickly find some wondrous abilities.

Resolving object containment issues involves questions of the form, "On *this* truck, give a way to refer to the fifth package on your manifest", or, "For *this* list of customers, tell me how to refer to the client named 'Able & Baker'".

In the "Scripting.p" listing, we have a routine "`DocObjectAccessor`" which is used to find specific rows within a given document.

As in the property case, we start by extracting one of our tokens from the supplied `containerToken`. We also confirm that we are being asked for a contained row (coded as '`crow`'), which is the only contained element we know about.

```
 IF NOT GetTokenFromAEDesc(containerToken, docToken)
THEN
  Bail(eContainerDoesNotHaveValidToken);

 IF desiredClass = 'crow' THEN
    {that's good}
 ELSE
  Bail(eContainerDoesNotContainRequestedClass);
```

In this example, we take the object reference contained in our token and convert it to a class reference meaningful within our TCL progr The fact that this example used TCL is of small consequence; the crucial notion is that you use whatever you defined as the contents o the token to now identify a particular "object" in your application.

```
 doc := KMyDoc(docToken.theObject);
```

Now that we have a reference to the document, we will look at a lis rows that happens to have been specified as part of our document. (TCL fans: In the actual application from which this is extracted, we have our own document class, descended from `CDocument`, and o of its variables, `rowlist`, is of class `CList`.)

```
 rowlist := doc.rowlist;
 nrow := rowlist.GetNumItems;
```

We now have an internal reference to the list of rows, and we know how many entries the list has. We now look to see how we are to identify the desired specific row. If `keyForm` = `formAbsolutePosition`, that means that the row is to be identified by a serial index. We then extract the desired index from keyData, and check that it is within the valid range.

```
IF keyForm = formAbsolutePosition THEN
  BEGIN
   wantedIndex := LongHandle(keyData.dataHandle)^^
   IF keyData.descriptorType = typeLongInteger THE
    BEGIN
     IF wantedIndex <= 0 THEN
      wantedIndex := nmedia + wantedIndex + 1;
    END
   ELSE
```

```
    Bail(errInvalidReference);

   IF (wantedIndex < 1) | (wantedIndex > nrow) THEN
    Bail(eIndexNumberOutOfRange);

   row := KRow(rowlist.NthItem(wantedIndex));
          {get desired row, by index}
   found := TRUE;
  END
 ELSE
  Bail(eOnlyNameIndexFirstOrLast);
```

If the desired sub-element was found, we set the fields of a token accordingly; this is passed back to the system, completing the resolution task.

```
 IF found THEN
  BEGIN
   theTokenBody.myTokenCode := rowTokenCode;
   theTokenBody.theObject := CObject(row);
   theTokenBody.subReference := 0;
   theTokenBody.isAProperty := FALSE;
   Bail(noErr);
  END
 ELSE
  Bail(errAENoSuchObject);
```

In this brief description, we have only shown how to resolve an element within a container by serial index. With only a little more work (none of it very complex) we add the ability to find elements by name or by keyword (such as "first" or "last"). This adds an exciting pizzazz to the scripting facility, and makes the user's AppleScripts simpler and more lucid.

We won't discuss such enhancements here, but a few samples of these improvements are included in the accompanying listing. Now that you know what "object resolution" is actually about, you'll also find it easier to understand the documentation on this topic in "Inside Macintosh: Interapplication Communication", pages 6-12 to 6-15.

Getting and Setting Properties: Preparation

We have now accomplished all of the codework needed to allow the user to examine and set values in the system, with the exception of the actual nitty-gritty: the retrieval or setting itself!

As you'll recall, we referred to a single procedure, DoTransferProperty, in both our Set Data and Get Data handlers. This routine will be used to both set and get property values. We have defined its calling sequence as follows:

```
FUNCTION DoTransferProperty (propAction:
propActionType; VAR myToken: MyTokenType; ae:
AppleEvent): OSErr;
```

The first parameter just sets a direction, either "doSet" or "doGet". course, you indicate this direction with just a Boolean, but I like the self-documenting quality of enumerations.)

The second parameter is the token identifying the property under discussion.

The third parameter is the AppleEvent involved, the incoming AppleEvent for Set Data or the reply AppleEvent for Get Data.

The calls to DoTransferProperty thus looked like this:

```
{in the Set Data handler}
  direction := doSet;
  err := DoTransferProperty(direction, myToken,
theAppleEvent);

{in the Get Data handler}
  direction := doGet;
  err := DoTransferProperty(direction, myToken,
reply);
```

For full details, consult the complete listings accompanying this arti For now, here are some highlights.

DoTransferProperty begins by pulling out the target object and its desired property from the supplied token. If the object is specified class library object or a data handle, it is a good idea to lock the obj since we'll be doing a fair amount of juggling. (We also restore the original lock value when this routine returns.)

```
 obj := myToken.theObject;
 prop := myToken.propertyCode;
 oldLock := obj.Lock(TRUE);
```

The main structure of DoTransferProperty consists of a series of branches to handle the different types of objects that can be handled the number of different object types becomes large, you may wish t break this into separate routines, or to methods of the objects' separa classes in a class library. For our simple example we have:

```
{APPLICATION}
 IF myToken.myTokenCode = typeNull THEN
  BEGIN
     - - -
  END

{WINDOW}
 ELSE IF myToken.myTokenCode=winTokenCode THEN
  BEGIN
     - - -
  END

{DOCUMENT}
 ELSE IF myToken.myTokenCode=docTokenCode THEN
  BEGIN
     - - -
  END

{ROWS}
 ELSE IF myToken.myTokenCode = rowTokenCode THEN
  BEGIN
     - - -
  END

 ELSE
  err := eCannotHandlePropertiesOfThisClass;
```

Within the BEGIN..END section for each class, we test for each property that we support. If found, we pass the address of the property itself and the relevant AppleEvent to another that does the actual transfer.

As an example, our "row" class has properties that include its name, its height, and various line and fill colors and patterns. We handle these row properties by first coercing our object reference for convenient future use. We then check for each property, calling the "TransferProperty" routine when we find the correct one. Here is an excerpt:

```
   row := KRow(obj);

   IF prop = '*mht' THEN
    err := TransferProperty(propAction, @row.height,
'I', SIZEOF(row.height), TRUE, ae)
   ELSE IF prop = 'flpt' THEN
    err := TransferProperty(propAction, @row.fillPat,
'I', SIZEOF(row.fillPat), TRUE, ae)
   ELSE IF prop = 'pppa' THEN
    err := TransferProperty(propAction, @row.linePat,
'I', SIZEOF(row.linePat), TRUE, ae)
   ELSE IF prop = 'flcl' THEN
    err := TransferProperty(propAction, @row.fillCol,
'I', SIZEOF(row.fillCol), TRUE, ae)
   ELSE IF prop = 'ppcl' THEN
    err := TransferProperty(propAction, @row.lineCol,
'I', SIZEOF(row.lineCol), TRUE, ae)
   ELSE IF prop = 'ppwd' THEN
    err := TransferProperty(propAction,
@row.lineThick, 'I', SIZEOF(row.lineThick), TRUE,
   ELSE IF prop = 'pnam' THEN
    err := TransferProperty(propAction,
PTR(row.title^), 'S', SIZEOF(row.title^^), TRUE, a
     - - -
   ELSE
    err := eThisPropertyUnderConstruction;
```

For example, if the specified property is 'flpt', this is our code for "f pattern". In our application, we store the fill pattern for a row withi the "fillPat" instance variable of an object of type KRow. The AppleEvent that contains the new value (Set Data) or the reply AppleEvent to receive the existing value (Get Data) is specified by The actual transfer may now be performed.

Getting and Setting Properties: The Real Thing

All actual transfers are conducted by a routine we call TransferProperty:

```
FUNCTION TransferProperty (propAction:
propActionType; propPtr: ptr; kind: char; lenProp:
integer; writeable: Boolean; ae: AppleEvent): OSEr
```

We start by choosing an AppleEvent value type that best fits the property:

```
 IF (kind = 'I') & (lenProp = 2) THEN
  descriptor := typeShortInteger
 ELSE IF (kind = 'I') & (lenProp = 4) THEN
  descriptor := typeLongInteger
 ELSE IF (kind = 'R') & (lenProp = 4) THEN
  descriptor := typeShortFloat
 ELSE IF (kind = 'R') & (lenProp = 8) THEN
  descriptor := typeLongFloat;
```

If the direction is "propGet", we must retrieve the value at the locati specified, and pack it into the supplied AppleEvent (which is the rep event).

```
 IF propAction = propGet THEN
  BEGIN
    {get the value of the specified property}
```

```
   IF lenProp <= SIZEOF(buffer) THEN
    BlockMove(propPtr, @buffer, lenProp)
     {copy the property into the buffer}
   ELSE
    BEGIN
     err := eBufferTooSmall;
     GOTO 99;
    END;

   {stuff the value into the AppleEvent}
   IF descriptor <> difficult THEN
    err := AEPutParamPtr(ae, keyDirectObject,
descriptor, @buffer, lenProp)
   ELSE IF kind = 'S' THEN
    BEGIN
     strlen := ORD(buffer[0]);
     err := AEPutParamPtr(ae, keyDirectObject,
typeChar, @buffer[1], strlen);
    END
   ELSE
    err := eCannotHandleAPropertyOfThisType;
  END
```

On the other hand, if the direction is "propSet", we extract the desired new value from the AppleEvent, and set the property to this new value:

```
 ELSE IF propAction = propSet THEN
  BEGIN
   IF NOT writeable THEN
    BEGIN
     err := errAENotModifiable;
     GOTO 99;
    END;

   {retrieve the new value from the AppleEvent}
   IF kind = 'S' THEN a string
    BEGIN
     err := AEGetParamPtr(ae, keyAEData, typeChar,
actualType, @buffer[1], SIZEOF(buffer) - 1,
actualSize);
     IF err = noErr THEN
      BEGIN
      strlen := actualSize;
      IF strlen > 255 THEN
      strlen := 255;
      buffer[0] := CHR(strlen);

      IF (strlen + 1) > lenProp THEN {too big to fit
in a string structure this size}
      BEGIN
      strlen := lenProp - 1;
      buffer[0] := CHR(strlen);
      END;

      BlockMove(@buffer, propPtr, strlen + 1);
      END;
    END
```

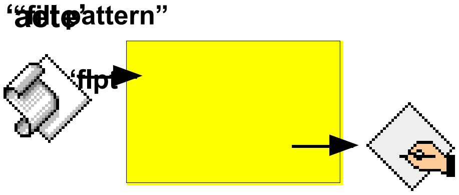
```
   ELSE IF descriptor <> difficult THEN
    BEGIN
     err := AEGetParamPtr(ae, keyAEData, descripto
actualType, @buffer, SIZEOF(buffer), actualSize);
     IF err = noErr THEN
      BEGIN
      IF descriptor <> actualType THEN      {we
didn't get what we wanted}
      err := ePropertyValueSpecifiedInIncorrectFor
      ELSE IF lenProp <> actualSize THEN
      err :=
ePropertyValueSpecifiedWithIncorrectSize;
      END;

     IF err = noErr THEN {everything looks good, s
revise the property itself!}
      BlockMove(@buffer, propPtr, lenProp);
    END
   ELSE
    err := eCannotHandleAPropertyOfThisType;
  END;
```

The beauty of the TransferProperty routine is that it handles all of th get/set needs of this sample within one place. It includes provision f variables of varying types and sizes, and provides a double-check so that "read-only" properties (such as creation date, or whether a wind has a title bar) can't be revised.

Believe it or not, we have now concluded all of the coding additions that are necessary to support the vital Set Data and Get Data events. There is only one obstacle remaining before we can say that our application is at least minimally scriptable.

The Dreaded 'aete' Resource

This ugly little beast serves as the translator between how the user t about the components of your app and how your application discuss those same elements with the scripting calls.



For example, the user may create a script that refers to a property by name of "fill pattern". Using the 'aete' resource contained in your application, the system translates this to a code of 'flpt'. When the system asks your application about this property, it will refer to the code.

Unfortunately, the organization of the 'aete' is remarkably convoluted. It organizes the scripting terminology first into "suites" and then lists events, objects and enumerations for each, with properties listed for each object. The complexity of the 'aete' is indicated by the fact that there are only two reasonable ways of creating and editing one, with a resource compiler such as MPW's Rez, or with Resorceror. Apple's own ResEdit tool is *not* capable of taking on 'aete'. (At the 1996 WWDC, Apple promised to deliver a more rational tool for aete editing.)

At this stage, if you've gotten this far in your development, you'll simply want to see scripting happening in your application. As a very simple first cut, you may wish to start with an existing simple 'aete' from another application. The 'aete' that is contained in the Scriptable Text Editor ("STE") sample application from Apple serves as a good foundation.

STE's 'aete' already includes the Get Data and Set Data events, and it includes references to the document object. You may wish to test your scripting features by implementing tests for these document properties. To do this, copy the 'aete' unchanged from the STE into your app. (Of course, it should never go on to users in this form.)

When you successfully have provided access to document properties in this manner, you can then add a few of your own objects. Using Rez or Resorceror, you will need to make these changes:
1) You will need to create entries for the new classes you define, and
2) You will need to identify these new classes as elements within the existing classes that contain them. For example, if you define a class "row" that is contained by the document, then the document class must be revised to show that it now has "row" as one of its sub-elements.

When you have completed a very rough 'aete' resource, you can now test and debug your newly-scriptable application!

At this point, you should pause and seriously review your objectives in scripting and how they would best be handled within your scripting facilities. When you finally get some form of scripting to work, the temptation is strong to plunge ahead and start coding up all kinds of objects and properties. Resist! This is the time to give deep thought to what you want your dictionary to look like to your users. Cal Simone's very fine articles in develop magazine are an excellent source for insights in this area.

Counting

There is one final element that pretty much must be included to qualify for a minimal level of scriptability. That is the ability for a script to determine the count of the number of objects there are of a given kind.

[Code to support the calls for counting is included in the sample code. Time permitting, this will be discussed in the live presentation.]

Scripting Alternatives

There are many arguments for AppleScript as an ideal scripting language. It would not be a stretch to say that many AppleScript us have crossed over into serious fandom about the language.

One of the great allures of AppleScript is that it looks like English. the very least, this can help make it easier for people to understand what an existing script is designed to do. While the meaning of "row(3).height = 20" can be learned with some basic programming training, the AppleScript equivalent, "set the height of row 3 to 20", takes no training at all.

In counterpoint to its apparent simplicity, AppleScript is also a real programming language that includes most of the essential construct one expects. The language includes loops, tests, branches, variables and mathematical and logical operators all the usual goodies.

Despite its allure, its strengths and its small but very dedicated fan following, AppleScript may not be the ideal solution for all users in situations. The language does have some drawbacks as well:

1) The English language syntax of AppleScript, which is a boon for easy *reading* of scripts, can be a barrier to *writing* of scripts. Use o natural language

implies a fluidity of meaning that AppleScript just simply does not support. I looked up the word "SET" in the Oxford dictionary, and it had 194 definitions; in AppleScript, "set" has but one meaning. If we promise the user that AS is natural to use, it becomes hard to explain why "get count of rows, copy it to rowcount" doesn't work. Similarly, all the keywords in "set me to true" are legal, but this sentence is completely wrong in AppleScript.

2) Involved statements in AppleScript strip away much of the English-like allure, by producing complex run-on sentences that really don't look much like English anymore.

3) AppleScript would appear to be a poor choice for long programs to do data processing, report generation or numerical calculations, if only due to the sheer verbosity that would be required.

4) A simple set of syntax rules in a traditional computer language may be easier to learn than a loose set of English-like structures. For example, BASIC makes no promise of fluidity. To express commands, a fairly simple set of rules do the job just fine.

5) Sources of information about AppleScript are limited, while there are hundreds of books and classes on how to learn BASIC.

6) Many people already know BASIC. BASIC is a thriving, immensely popular real world language. In the form of Visual Basic, it is arguably *the* most popular development environment for Windows. Even on the Mac, BASIC is rapidly gaining popularity as the preferred macro language for Microsoft's applications.

[The live presentation of this paper include some experimental approaches to scripting using tools other than AppleScript.]

Conclusions

Much of the benefit of making your application scriptable can be achieved in a manageable series of short tasks. Once accomplished this also serves as a sound basis for expanding your support for scripting.

Scriptability is not the same as AppleScript. Making an application scriptable opens the door to control from other scripting languages i the future.

Bibliography

Apple Computer, "Inside Macintosh: Interapplication Communicati Addison-Wesley, 1993. This is the great mother ship for AppleEv and scripting information. For completeness, relative clarity, and code samples, it is decidedly preferred to its predecessor, IM Volu VI.

Dave Mark, "Ultimate Mac Programming", IDG Books, 1994. This the only source, other than the IM, for in-depth coverage of implementing scriptability in one place. As usual, Mark provides cordial and painless introduction to his topic. This book also inclu (as appendixes) the Clark, Berdahl, and Simone (develop 21) artic listed below.

Richard Clark, "Apple Event Objects and You", develop, Issue 10, N 1992. This fine article is notable in particular for its task-oriented approach: it focuses on many of the issues that a scripting develop will encounter.

Eric Berdahl, "Better Apple Event Coding Through Objects", develo

Cal Simone, "Designing a Scripting Implementation", develop, Issu

Cal Simone, "Scripting Quandaries", develop, Issue 22, June 1995.

Cal Simone, "Thinking About Dictionaries", develop, Issue 23,

Greg Anderson, "Speeding Up whose Clause Resolution in Your

Cal Simone, "Steps to Scriptability", develop, Issue 24, December 1995.