PowerPC Branch Prediction • Text Tracks in Movies • A Space-Saving PICT Trick

develop

The Apple Technical Journal

Issue 20 December 1994

Getting the Most out of AOCE Catalog Records

Exploring Advanced AOCE Templates

Make Your Own Sound Components

Scripting the Finder From Your Application

NetWare on PowerPC

Improving QuickDraw GX Printer Driver Performance

$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*
Managing Editor *Toni Moccia*
Technical Buckstopper *Dave Johnson*
Bookmark CD Leader *Alex Dosher*
Able Assistants *Meredith Best, Liz Hujsak*
Our Boss *Greg Joswiak*
His Boss *Dennis Matthews*
Review Board *Pete "Luke" Alexander, Ray Chiang, Dave Hersey, Dave Radcliffe, Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*
Contributing Editors *Lorraine Anderson, Steve Chernicoff, Toni Haskell, Judy Helfand, Jody Larson, Joe Williams*
Indexer *Marc Savage*

ART & PRODUCTION

Production/Art Director *Diane Wilcox*
Technical Illustration *Sandee Karr*
Formatting *Forbes Mill Press*
Photography *Sharon Beals, Jean-Eric Garnier, Marcie Griffin, Mark Maxham*
On-line Production *Cassi Carpenter*

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Press group, is published in March, June, September, and December. *develop* articles and code have been reviewed for robustness by Apple engineers.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark* CD. The Bookmark CD contains a subset of the materials on the monthly *Developer CD Series*, which is available from APDA. Included on the CD are this issue and all back issues of *develop* along with the code that the articles describe. The *develop* code is updated when necessary, so always use the most recent CD. The CD also contains Technical Notes, sample code, and other useful documentation and tools (these contents are subject to change). Software and documentation referred to as being on this issue's CD are located on either the Bookmark CD or the Reference Library or Tool Chest edition of the *Developer CD Series*.

The *develop* issues and code are also available on AppleLink and via anonymous ftp at ftp.apple.com.

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something like "(QT 4)," this indicates the category and number of the Note on this issue's CD. (QT is the QuickTime category.)

E-mail addresses. Most e-mail addresses mentioned in *develop* are AppleLink addresses; to convert one of these to an Internet address, append "@applelink.apple.com" to it. For example, DEVELOP on AppleLink becomes develop@applelink.apple.com on the Internet. To convert a NewtonMail address to an Internet address, append "@online.apple.com" to it.

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)974-6395. Or write to Caroline Rose at the above address.

Subscriptions. Subscribe to *develop* through APDA (see below) or use the subscription card in this issue. For subscription changes or queries, call 1-800-877-5548 in the U.S. or (815)734-1116 outside the U.S., or write to AppleLink DEV.SUBS, Internet dev.subs@applelink.apple.com, or *develop*, P.O. Box 531, Mount Morris, IL 61054-7858.

Back issues. Printed back issues are available for $13 each in the U.S. or $20 outside the U.S. To order, call 1-800-877-5548 in the U.S. or (815)734-1116 outside the U.S., or write to AppleLink DEV.SUBS, Internet dev.subs@applelink.apple.com, or *develop*, P.O. Box 531, Mount Morris, IL 61054-7858.

APDA. To order products from APDA or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. Order electronically at AppleLink APDA, Internet apda@applelink.apple.com, CompuServe 76666,2405, or America Online APDAorder. Or write APDA, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

Issue 20 December 1994

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

At the Worldwide Developers Conference in May, hundreds of developers volunteered comments on *develop*, the Developer CD, and related products, writing them either on a big board we set up for that purpose or in a comments box in our electronic survey. The remarks were collated and handed out to the people in Apple's Developer Press group who are responsible for those products. (Our Able Assistant Meredith Best, whom you can see hard at work with other *develop* team members on page 111, typed up all those comments that were scrawled on the big board — even the complaints about the conference food and the unique "At what elevation do deer turn into elk?")

So what results come out of this, if any? That's harder to answer. Of course it depends on the nature of the comment. The result of all the great feedback on *develop* is that I'll have the pleasure of doing this job for a while longer. (Thanks especially to the person who wrote "*develop* is brilliant! Give Caroline a raise now!"; it was hard convincing my boss I didn't enter that one myself.) The answers to some questions are really easy. For example, "How about a guest developer column?" is a no-brainer; we've always accepted columns from developers (and articles too!) and would be happy to send our author's guidelines to anyone who's interested.

Most of the comments from the conference are in the process of being assimilated, and what they'll yield is uncertain, but we do want to thank you for sharing, as we say in California. In turn we'll share with you some feedback that was among the strongest and that applies to many of you out there as well as to us at Apple: documentation should *not* be electronic-only. (OK, a couple of developers disagreed, and one said "Ehh, who cares," but we're talking overwhelming majority here.) Paper was praised.

But it's clear from other feedback that developers also hold up THINK Reference as a model for viewing documentation on-line. Does this contradict their praise of paper? I think not. They're recognizing the difference between reference and other types of documentation. They want paper for conceptual information but would like good (fast!) on-line access for reference materials. I for one am happy to see this distinction being made. Not only are the different types of materials too often lumped together when the big question of on-line versus off-line presentation arises, but this separation isn't being realized enough in printed books either. The first thing I polled developers on when planning NeXT's technical documentation (in a past life) was this very question, and as a result, the reference-type descriptions were placed in an entirely separate book. Read-once versus read-many — simple.

What I'd like to know is, if so many hundreds of you took the time to give us feedback at the conference, why don't more of you write to *develop*? You know where to find us . . .

C Rose

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) has been writing about software ever since the Internet was the ARPAnet. She was originally hired because they wanted the fresh approach of someone new to the industry, which in those days wasn't hard to find. Now she has to leave town to find someone new to the industry. Caroline loves her job but seems to enjoy leaving town most of all. Her big trip this year was to England, where she especially enjoyed the green rolling hills and quaint old towns of the Cotswolds. She was also crazy about British desserts, particularly the banoffi pie (she lacked the courage, however, to try the Spotted Dick).•

LETTERS

PENMODE NOT FOR TEXT

In the Macintosh Q & A section of *develop* Issue 18, I noticed the suggestion to use PenMode(srcBic) for drawing white text on a black background. This is wrong; the desired result will be achieved by TextMode(srcBic). Hope this helps.

— David Surovell

Thanks for the correction. Say, don't I recognize your name from an article in Issue 19?

— Caroline Rose

SLIME: IS IT SAFE?

As I was reading *develop* Issue 18, I came across the green slime question in Macintosh Q & A. A friend of mine has been looking for just this recipe, to help keep her preschool-aged daughter amused. However, after reading the recipe, I'm not going to forward it to my friend, because one of the ingredients, borax, is poisonous.

Now, I'm not a toxicologist, nor do I play one on TV, so I don't know exactly how toxic borax is. But I don't want to find out by letting anyone's children play with it (not even the nasty little feral children down the street). And what about people who screw up the recipe? Or decide to experiment with it? You didn't even tell them that straight borax is poisonous. And it's not like there's a big "Mr. Yuk" sticker on the box of Twenty-Mule Team, just those friendly-looking equines.

You and your staff do a fine job of technical presentation, and you have a process for technical review. You might consider a "toxicity review" before you publish any more recipes for nifty stuff that's not inherently edible. Ya never know.

— Greg Guerin

I volunteer at the Exploratorium, a science museum in San Francisco, and we've been handing out this slime for months to any kid who wants it. The recipe was acquired from a grade-school teacher and is used in schools all over the country.

A chemist at the Exploratorium told me that borax is toxic to about the same degree as soap: if you eat enough of it, you'll probably get a belly ache. Any kid who's old enough to have figured out that soap is yucky to eat is plenty old enough, in my opinion. It's my sincere hope and belief that most develop *readers have already reached that stage of maturity.*

To be absolutely sure, I called the Poison Control center, and they said that to have any toxic effect at all someone would have to eat an awful lot of it. They even told me I didn't need to bother to wash my hands after handling it. I asked if the soap analogy was a good one, and they said it was accurate.

You might recommend corn starch and cold water to your friend. In the right proportions (just enough water to get it all wet) it makes a very satisfying goop (usually called "oobleck" by teachers, after Dr. Seuss) that's probably more appropriate for very young children.

— Dave Johnson

DOGCOW IN THINK REFERENCE

I really enjoyed reading Mark Harlan's history of the dogcow in Issues 17 and

IT PAINS US WHEN YOU DON'T WRITE
We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014 (AppleLink CROSE or JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

18 of *develop*. He states that Technical Note #31 has not been available for quite a while. That may be so, but something like this Tech Note is hiding inside the THINK Reference databases. Just do a search for "DogCow" and you'll find it.

— Robert Grimm

Thanks for pointing out the dogcow lore in THINK Reference; I didn't know about that. I especially like the part about how, since the dogcow is two-dimensional, she can face a predator head-on to avoid being seen. Although that defensive maneuver sounds likely enough, keep in mind that the dogcow information in THINK Reference is unauthorized and has nothing to do with the Tech Note. The Tech Note gives lots more information, and we're glad it remains as mysterious as ever.

— Caroline Rose

PUZZLING OVER THE PUZZLE PAGE

After reading Issue 17 of *develop*, I have a question. I'm not sure what the purpose is in calling KON & BAL's page a Puzzle Page if it requires that you have access to certain obscure beta ROMs in order to solve the puzzle. Perhaps this is just sour grapes because I've never scored above 5? Sure, it still demonstrates various debugging techniques (although I'm not sure that iterative debugging with a reboot after each test is a very useful technique, and this seems quite common in the puzzle pages). But is it really a puzzle?

— Peter Lewis

KON & BAL chose the Puzzle Page format because they thought it was a fun way to give people debugging tips. They don't expect readers to take the puzzle aspect of it seriously; in fact, you're the first one we've ever heard from who has scored anything besides 0.

None of us were thrilled with Issue 17's puzzle, but, in KON's words, "There was a lot of great stuff about how the Resource Manager works, locking down handles, and other really useful advice. We try to demonstrate efficient and good debugging techniques."

Want a better Puzzle Page? Why not write one yourself? We now accept "guest puzzlers," as you may have noticed. If you've got a good idea for a Puzzle Page, please send it to us at AppleLink DEVELOP.

— Caroline Rose

SAVED BY THE PUZZLE PAGE

I've enjoyed reading *develop* since the first issue was published. I find that the articles contain a lot of useful technical information.

Normally I try to read each issue when it arrives, but when Issue 16 arrived, it sat for two months while I was finishing a product. About two weeks after the application shipped, a bug was reported where the application would randomly crash with a trashed stack on the PowerBook 180c. We spent a very frustrating Friday trying to reproduce and isolate the bug, but the behavior was inconsistent.

Over the weekend, in an effort to catch up on my reading, I picked up *develop* Issue 16 and read through it. About an hour after I finished, I thought back on KON & BAL's Puzzle Page, picked the magazine up again, and reread it. Something struck me about the problem they were puzzling over. Their result sounded similar to the problem we were encountering.

On Monday, we ran a series of tests in which we were able to prove that the bug was in Sound Manager 2.0 and disappeared under Sound Manager 3.0. If it hadn't been for the timely coincidence of reading *develop* and seeing a different manifestation of our bug described, we might have spent a lot longer tracking down the problem. *develop* saved us a lot of time and frustration.

Thanks for the magazine.

— Bruce D. Rosenblum

Getting the Most out of AOCE Catalog Records

Apple Open Collaboration Environment (AOCE) catalogs can contain any kind of data, which users can browse and edit using the Finder. Developers and knowledgeable end users can write AOCE templates, which add new record types to catalogs and tell the Finder how to display the data. This article describes AOCE templates and provides an example of using templates to extend PowerTalk's built-in User records to contain your own data.

CHRISTINE BUTTIN

Personal computers are great for storing large amounts of related data. Databases make it possible to organize data and to find individual items quickly, but it takes a long time to set up a database and enter data into it. If you're a system administrator or in-house developer and you want to provide a database for use by others, you also have to worry about providing and maintaining the database software for everyone involved. If you're using PowerTalk, however, you already have a way to store information: AOCE catalogs. What's more, the related Catalogs Extension (CE) to the Finder allows users to browse and edit catalog records with the Finder.

The Human Resources group at Apple Computer France wanted to create a directory of Apple personnel that included each person's name, title, department, address, and a list of keywords to identify areas of expertise. Furthermore, the group wanted to be able to list everyone with a particular area of expertise. Since Human Resources uses a PowerShare collaboration server, there was already a catalog with a User record containing all of the information for everyone with an account on the server, except for areas of expertise. What better way to create the directory than to add the new keyword data to the User records? To list people related to a specific keyword, the group could create keyword records by adding a new record type to the AOCE catalog in which the data was stored.

The User records that are defined as part of the PowerTalk system software store information such as a person's name, title, phone number, and electronic address. PowerTalk uses AOCE templates — resource files that go in the Extensions folder in the System Folder — to tell the CE how to store and display the data. This article shows you how to write AOCE templates that extend User records to hold additional

CHRISTINE BUTTIN has worked in Developer Technical Support for Apple since 1989, first for Apple Computer Europe and now for Apple France. At the office, they call her the talkative woman, not only because she enjoys chatting, but also because she mainly supports technologies that enable communication (such as AppleTalk, AppleScript, AOCE). When not talking, she spends most of her time practicing Aikido or traveling (she has a special love for the Sahara Desert, where she's sure to be far away from computers). An extra benefit of her job is that she can regularly visit the U.S., where she has made many good friends. Visiting her American friends might not be as restful as a trip to the Sahara Desert, but it certainly is a lot of fun.•

information (keywords identifying areas of expertise) and templates that define a new record type (keyword records). Although writing templates doesn't necessarily require writing any code resources at all, you can do more by adding code resources to your templates. This article goes on to demonstrate how to use a code resource to keep records synchronized that refer to each other's data.

You'll find all the example code on this issue's CD. Even if you're not using a PowerShare server, you can use the approach described in this article to store data in a personal catalog on your own Macintosh.

A BIT ABOUT AOCE CATALOGS

To get the most out of this article, you should have PowerTalk installed on your computer and have spent a little time playing with catalogs. A *catalog* is a hierarchically arranged store of data. The bottom level of the catalog hierarchy is the *record*, which is analogous to a file in the Macintosh hierarchical file system (HFS). Unlike files, however, when the user double-clicks a record, the application that opens the record is the Finder itself. How the contents of the record are stored in the catalog and displayed to the user is determined by sets of resources in files known as *AOCE templates*.

For the complete story on AOCE, see *Inside Macintosh: AOCE Application Interfaces.*•

A PowerShare collaboration server stores the name and account information of each entity (person, gateway, or whatever) that has an account on the AOCE server in a server-based catalog. It uses User records for this purpose. A personal catalog looks much like a server-based catalog but is, in fact, an HFS file on the local disk. There's practically no difference between a record in a server-based catalog and one in a personal catalog; AOCE templates work identically in both cases.

The data in records is organized into *attributes*. Each attribute has a type (for example, address or area of expertise) and any number of attribute values, which can contain any sort of data. Each attribute type is defined by a template that specifies the format for the data. You can write new templates to expand the types of attributes that existing record types can contain — that's precisely what this article does, in fact.

FROM RECORDS TO INFORMATION PAGES

When the user double-clicks an AOCE catalog record in the Finder, a window called an *information page window* opens. An information page window can contain a single information page, or several information pages, each with a pop-up menu listing the other pages. Each information page displays data stored in the record. The window in Figure 1 shows an information page that displays data stored in a User record.

The CE uses a two-step process to get from a record to an information page. Because there's not necessarily a one-to-one correspondence between attribute values and the data you want to display on an information page, the first step consists of parsing the data in the attribute values into discrete units of data known as *properties*. For example, an address attribute value may contain street, city, and zip code properties. The second step is to specify exactly where and how each property is displayed on the information page.

Two types of AOCE templates specify how the CE performs each of these steps:

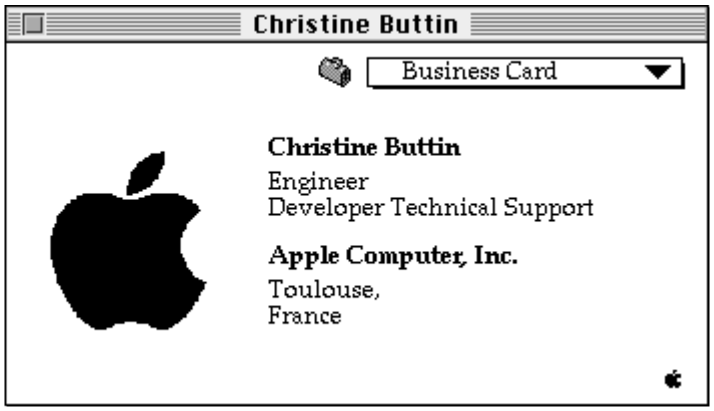
- An *aspect template* describes how the attribute values are to be parsed into properties.

- An *information page template* specifies how the properties are to be displayed on the information page.

These aspect and information page templates share a data structure in memory that contains the properties. This data structure is called an *aspect*. The relationships among records, aspect templates, aspects, information page templates, and information pages are illustrated in Figure 2.

There are a few important things to note about these relationships:

- An aspect template does not have to deal with every attribute type in a record. There can be any number of aspect templates that apply to a given record type, and each can describe the parsing of some subset of attribute types.
- An information page template does not have to use every property stored in an aspect.
- More than one information page template can use properties from the same aspect, but each information page template can use properties from only one aspect.



Christine Buttin
Engineer
Developer Technical Support
Apple Computer, Inc.
Toulouse,
France

Figure 1. An information page window

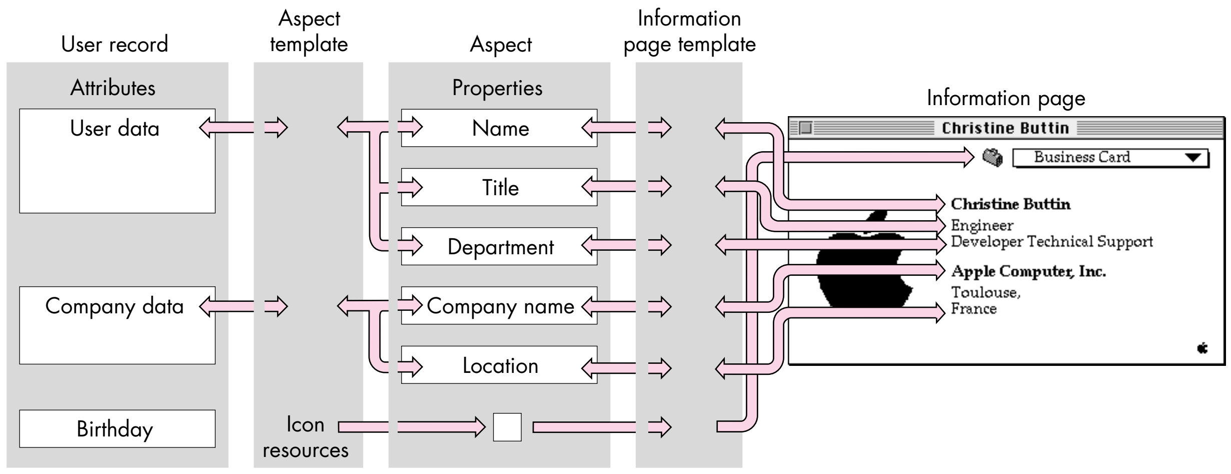


Figure 2. Getting from a record to an information page

- The process works in reverse as well. When the user enters data into an information page, the information page template defines which property that data belongs to and the aspect template describes how the data should be stored in an attribute value.

WHAT'S IN A TEMPLATE

Every template contains a *signature resource* that indicates the type of the template and specifies some other template characteristics. In addition, there are other resources that are required for every template (such as the template name resource), resources that are required for specific template types, and a variety of optional resources that you can include if needed.

The CE identifies each resource by its resource type and by the offset of its resource ID from the resource ID of the signature resource. For example, the template name resource and the record type resource (which specifies what type of record the template applies to) are both 'rstr' resources; the CE can distinguish between them because the template name resource's ID is equal to the signature resource's ID plus the constant value kDETTemplateName, while the record type resource has an ID offset of kDETRecordType.

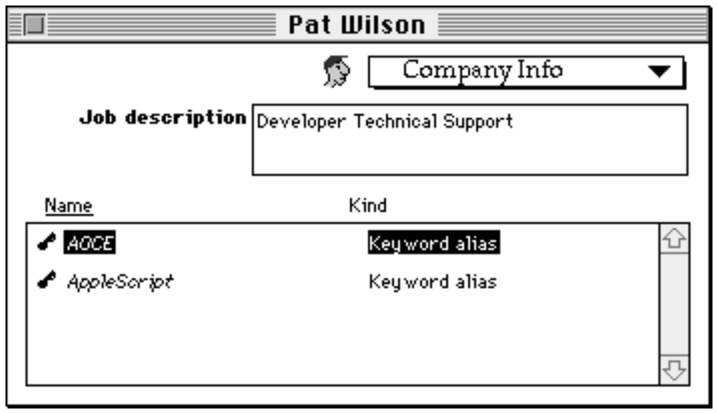
Aspect templates contain a resource called a *lookup table*. The lookup table contains the instructions to the CE for parsing attribute values into properties and properties into attribute values. If the CE needs a property that has a property number in the range 0 to 249, and it doesn't find a value for that property that the lookup table constructed from an attribute value, it looks for a resource with an ID offset equal to the property number. This means that in the aspect template itself you can provide property values to be used as default values, initial values, or constant values for properties.

Information page templates contain one or more resources called *view lists*, which specify the views that appear on the information page. A *view* is an item or a field on an information page displaying one or more property values (for example, a text field or a radio button).

Unlike the ID of other template resources, a view list's resource ID isn't related to the signature resource's ID. Instead, the information page template's signature resource includes references to all the view lists for that template. For each view list, the signature resource includes two property numbers that identify properties associated with that view list. The view list is active only if the values of its two associated properties are equal. You can use this feature to implement *conditional views*, that is, information-page features that the CE displays only under certain circumstances.

MAIN VIEWS, SUBLISTS, AND MAIN ASPECTS

Figure 3 shows another common feature of information pages — a *sublist*. A sublist is a portion of the information page that contains a list of attribute values or records. In Figure 3, the sublist holds two records (actually, it holds aliases to the records). Typically, when the user double-clicks an item in a sublist, the same two-step process as described earlier in the section "From Records to Information Pages" occurs; as a result, another information page opens and displays the information associated with the attribute or record represented by the selected item. For example, double-clicking the AOCE item in Figure 3 opens an information page displaying all people with that expertise (as shown in Figure 4 later in this article).


Pat Wilson

Job description: Developer Technical Support

Name	Kind
AOCE	Keyword alias
AppleScript	Keyword alias

Figure 3. Information page with a sublist

All the property values displayed on an information page outside the sublist come from a single aspect, called the *main view aspect*. This aspect also provides the list of items to be included in the sublist (if any).

Each item in a sublist has its own aspect, called a *main aspect*, which provides the property values necessary to display the item in the sublist (such as the name of a record or the kind of attribute value). A main aspect can contain other property values as well; in fact, a main aspect can also serve as the main view aspect for an information page.

If you want to create a new information page for an existing record, you must provide a main view aspect template for that information page. If you're defining a new record type to be displayed in a sublist, you need to provide a main aspect template for that record type. The example in this article demonstrates how to create a main view aspect template and an information page template that extend the User record to hold keywords. It then shows how to create the main aspect template needed in order to list keyword records in the new User record information page sublist.

DEFINING THE ASPECT TEMPLATE

Now we're ready to define the aspect template for the new information page, which is a main view aspect template. The main view aspect template serves as the aspect template for everything on the new User record information page except for the content of the sublist items. Aspect templates contain a signature resource, a name resource, a record type resource, resources that specify how to handle objects dropped on the information page, resources for handling the View menu and Balloon Help, and the all-important lookup table.

SIGNATURE, NAME, AND RECORD TYPE RESOURCES

The ID of the aspect template signature resource, of type 'deta', provides the base resource ID for the other aspect template resources. The signature resource also makes some settings related to drag and drop operations, as discussed in the next section, and specifies whether this template is a main aspect template.

```
resource 'deta' (kEInfoPageAspect, purgeable) {
   0,                  // Drop-operation order
   dropCheckAlways,    // Drop-check flag
   notMainAspect       // Not the main aspect template
};
```

In our example, the aspect template defines the properties for the new information page being added to an existing record type — "aoce User." Because User records already have a main aspect template provided as part of the PowerTalk system software, you don't have to provide one yourself. In fact, declaring a template to be the main aspect template for User records would cause a conflict and the User records would no longer work correctly.

To replace an existing main aspect template, you use a killer template, which is not covered in this article. See *Inside Macintosh: AOCE Application Interfaces* for more on killer templates.•

Aspect templates, like other templates, have a name that must be unique to be identified by other templates. To guarantee uniqueness, start the names of all your templates (as well as all the new record types and attribute types for which you provide main aspect templates) with your four-character application signature as registered with Apple's Developer Support Center. Here's our aspect template name resource:

```
resource 'rstr' (kEInfoPageAspect+kDETTemplateName, purgeable) {
   "ACFC InfoPage aspect"
};
```

The record type resource identifies the record that the aspect applies to, in this case the User record:

```
resource 'rstr' (kEInfoPageAspect+kDETRecordType, purgeable) {
   "aoce User"
};
```

Although we're going to be defining a new attribute type that goes in User records, we don't provide a resource (of type kDETAttributeType) to specify that attribute type for the aspect template. In fact, if an attribute type were specified for the aspect template, the CE could use the template only for that attribute type and could not use it to modify the User record itself. (If the new attribute type were in a sublist and we provided a main aspect template to describe how that attribute should appear in the sublist, that main aspect template would contain a kDETAttributeType resource, as shown in the sample code on this issue's CD.)•

DRAG AND DROP RESOURCES

The aspect template drag and drop resources in the sample code make it possible for users to drag keyword records and drop them either on a closed User record or directly on a sublist on an information page, thereby adding those records to the sublist.

The kDETAspectRecordDragIn resource specifies what types of records can be dropped on a sublist. Because the CE can't actually store a record in a record, it adds to the sublist an attribute containing an alias to the dropped record. For each type of record the user can drag in, you also need to specify the type of attribute to store the alias in. The following resource indicates that aliases to keyword records should be stored as attribute type "ACFC Alias keyword":

```
resource 'rst#' (kEInfoPageAspect+kDETAspectRecordDragIn, purgeable) {
   {
   "ACFC Keywords", "ACFC Alias keyword"
   }
};
```

The drop-operation order and drop-check flag in the aspect template signature resource (shown in the previous section) come into play during these drag and drop operations, as follows:

- The drop-check flag controls when the user will be prompted for confirmation upon performing the drag and drop. Setting this flag to dropCheckAlways indicates that the prompt message should always appear. Setting it to dropCheckConflicts limits its appearance to times when the user drops a record on a closed User record and more than one information page contains a sublist that accepts that record type. In this case, the CE would have to determine which information page should have the record added to its sublist. If you set the drop-check flag to dropCheckConflicts, the user is prompted for confirmation only if such a conflict arises.
- Setting the drop-operation order to 0 indicates that you want the highest possible priority in case of the conflict just described. If the conflicting template also set its drop-operation order to 0, the CE makes an arbitrary decision about which sublist to add the record to. The user can always avoid the conflict by opening the desired information page and dropping the object directly on the desired sublist.

You also need to define the prompt message, which can include parameters (token **^2** is the destination's name, and token **^3** is the dragged record's name).

```
resource 'rstr' (kEInfoPageAspect+kDETAspectDragInString, purgeable) {
   "Do you want to add %3%“^3”%the selected items% to the company info "
   "for “^2”?"
};
```

In this example, dragging the keyword record named "AppleScript" onto John's business card produces the message "Do you want to add AppleScript to the company info for John?" If the user drags several items at once, the CE substitutes "the selected items" for the name of the dragged item.

VIEW MENU COMMANDS AND BALLOON HELP

When a template contains a sublist, the user determines how data is sorted in the sublist by choosing from the Finder's View menu (for example, "by Name" or "by Kind") or by clicking the labels above the sublist ("Name" and "Kind" in Figure 3). For these features to work, you have to provide the items for the View menu and specify which properties are used for sorting in each case. Our example uses the metaproperties (properties that are provided by the CE and that don't correspond to specific attribute values): kDETPrName (the record's name) and kDETPrKind (the record's kind).

```
resource 'detm' (kEInfoPageAspect+kDETAspectViewMenu, purgeable) {
   kEInfoPageAspect+kDETAspectViewMenu,
   {
   kDETPrName, "by Name";
   kDETPrKind, "by Kind";
   }
};
```

You also need to add Balloon Help strings for properties. Each property has two strings: the first one is displayed if the property is editable, the second one if it's not. The first pair of text strings corresponds to the first property, the second pair to the second property, and so on. In our example, there's only one property (the employee's job description):

```
resource 'rst#' (kEInfoPageAspect+kDETAspectBalloons, purgeable) {
   {
   "Description of the employee's job", "Description : Uneditable "
   "because the record is locked or access controlled."
   }
};
```

THE LOOKUP TABLE

As mentioned earlier, the lookup table tells the CE how to parse attribute values into properties and how to convert property values into attributes. For each property that you want to use on an information page, the lookup table must contain an entry that describes how to process the property's associated attribute. An attribute is identified by an attribute type and an attribute value tag:

- The attribute type is a string that describes the contents of the attribute (such as "ACFC Alias keyword" or "aoce Member").
- The attribute value tag is a 4-byte code that specifies the data format of the attribute value (such as typePackedDSSpec or typeBinary).

A single lookup table entry can specify how to parse more than one attribute type, but only if they have the same attribute value tag. You can provide separate lookup table entries for input (that is, translating attribute values to properties) and output (translating properties to attribute values), but you must provide both.

Each lookup table entry contains a set of flags that indicate the following:

- whether the entry is used for translating attribute values to properties (useForInput)
- whether the entry is used for translating properties to attribute values (useForOutput)
- whether the attribute value is to be used in a sublist
- whether the resulting entry in the sublist is an alias

The entry also includes elements that specify what to do with the attribute. Each element consists of three parts: a format that drives the parsing process, a property number, and an extra parameter used in certain types of elements. The format can be a simple data type that specifies how many bytes of data to take from the attribute and how to store it in the property. For example, a format of type 'word' takes the next two bytes from the attribute value and puts it in a number-type property. In the other direction, it would take two bytes from a number-type property and store it in an attribute value.

Lookup tables are complex and can be used in more ways than described here. For more information, see *Inside Macintosh: AOCE Application Interfaces.*•

In our example, there are two attribute types: one contains the description of an employee's job (attribute type "ACFC Company Empext Function"); the second is multivalued and contains the list of keywords or users (contact people, not discussed in this article), which are records in the catalog. There are two entries in the lookup table, one for each attribute type: the first one maps the attribute value to a string property (kFunction, defined on the CD); the second is for sublist items, each of which is a record of type “ACFC Alias keyword.” The notForOutput and notForInput flags are set for the items in the sublist because sublist items are described in their own main aspects, not in the main view aspect that defines the sublist.

```
resource 'dett' (kEInfoPageAspect+kDETAspectLookup, purgeable) {
   {
   {"ACFC Company Empext Function"}, typeBinary,
      useForInput, useForOutput, notInSublist, isNotAlias, isNotRecordRef,
      {
      'rstr', kFunction, 0;           // An element
      };
   {"ACFC Alias keyword"}, typePackedDSSpec,
      notForInput, notForOutput, useInSublist, isAlias, isNotRecordRef,
      {};
   }
};
```

DEFINING THE INFORMATION PAGE TEMPLATE

As described earlier in “What’s in a Template,” an information page template specifies the contents and layout of an information page. It includes one or more view lists, which describe the different views on the page, such as text fields or radio buttons. When your information page contains a sublist, you must include a view list describing an entry in the sublist. The CE uses this view list to display appropriate information for each item of the sublist, such as an icon and the name of the item.

SIGNATURE, NAME, AND TYPE RESOURCES

The information page template signature resource provides the resource IDs of the view lists associated with the information page, as well as two property numbers for each view list. As discussed earlier, if the values of these two properties are not equal, the CE doesn’t display the views described by the view list.

The signature resource also specifies the sort-order number of the information page, the presence or absence of a sublist in the information page, and the rectangle that contains the sublist (if any). The CE displays the information pages in the sequence indicated by their sort-order numbers.

```
resource 'deti' (kEInfoPage, purgeable) {
   4000,               // Sort-order number
   {kSublistTop, kSublistLeft, kSublistBottom, kSublistRight},
   noSelectFirstText,  // Don't automatically select the first editable
                       // text field when the information page is opened.
   {
   kDETNoProperty, kDETNoProperty, kEInfoPage;
   },
   {
   kDETNoProperty, kDETNoProperty, kEInfoPage+1;
   }
};
```

As with the aspect template, you need to include resources that provide the template name and the type of record the template applies to:

```
resource 'rstr' (kEInfoPage+kDETTemplateName, purgeable) {
   "ACFC 1stInfoPage"
};
```

```
resource 'rstr' (kEInfoPage+kDETRecordType, purgeable) {
   "aoce User"
};
```

OTHER REQUIRED RESOURCES

Because there's more than one information page for the User record, you need to provide the string that gets displayed in the information page pop-up menu used for changing to a different page:

```
resource 'rstr' (kEInfoPage+kDETInfoPageName, purgeable) {
   "Company Info"
};
```

You also need to provide the name of the aspect template that defines the properties used by the information page:

```
resource 'rstr' (kEInfoPage+kDETInfoPageMainViewAspect, purgeable) {
   "ACFC InfoPage aspect"
};
```

VIEW LISTS

Our example has two view lists: the first describes the views (graphical elements) of the information page, and the second describes an entry in the sublist. To complete the information page template, you need to define these view lists (see Listing 1). A view list contains the following information for each view:

- the view's bounds
- the view's type (such as button or editable text field)
- the property associated with this view, if any
- information specific to the type of view

Listing 1. Defining the view lists

```
resource 'detv' (kEInfoPage, purgeable) {
   {
   // Icon
   {6, 156, 22, 172}, kDETNoFlags, kDETAspectMainBitmap, Bitmap {kDETSmallIcon};
   // Static text
   {kFunctionTop, kFunctionLeft, kFunctionBottom, kFunctionRight}, kDETNoFlags, kDETNoProperty,
      StaticTextFromView {kDETApplicationFont, kDETApplicationFontSize, kDETRight, kDETBold,
      "Job description"};
   // Editable text
   {kTEFunctionTop, kTEFunctionLeft, kTEFunctionBottom, kTEFunctionRight}, kDETMultiLine, kFunction,
      EditText {kDETApplicationFont, kDETApplicationFontSize, kDETLeft, kDETNormal};
   // Sublist label: "Name"
   {kSublistSeeAlsoTop, kSublistSeeAlsoLeft,kSublistSeeAlsoBottom, kSublistSeeAlsoRight},
      kDETNoFlags, kDETPrName,
      StaticCommandTextFromView {kDETDefaultFont, kDETDefaultFontSize, kDETLeft, kDETUnderline,
      "Name", kDETChangeViewCommand, -1};
```

(continued on next page)

Listing 1. Defining the view lists *(continued)*

```
   // Sublist label: "Kind"
   {kSublistKindTop, kSublistKindLeft, kSublistKindBottom, kSublistKindRight},
      kDETNoFlags, kDETPrKind,
      StaticCommandTextFromView {kDETDefaultFont, kDETDefaultFontSize, kDETLeft, kDETNormal,
      "Kind", kDETChangeViewCommand, -2};
   // Sublist box
   {kSublistTop-1, kSublistLeft-1, kSublistBottom+1, kSublistRight+1}, kDETNoFlags,
      kDETNoProperty, Box {kDETUnused};
    }
};

// Sublist view list -- description of an entry in the sublist
resource 'detv' (kEInfoPage+1, purgeable) {
   {
   // Icon
   {kDETSublistEntryTop, kDETSublistIconColumnLeft, kDETSublistEntryBottom,
      kDETSublistIconColumnRight}, kDETEnabled, kDETAspectMainBitmap, Bitmap {kDETMiniIcon};
   // Record's name
   {kDETSublistEntryTop, kSeeAlsoColumnLeft, kDETSublistEntryBottom, kSeeAlsoColumnRight},
      kDETEnabled+kDETDynamicSize, kDETPrName,
      EditText {kDETDefaultFont, kDETDefaultFontSize, kDETLeft, kDETItalic};
   // Record's type
   {kDETSublistEntryTop, kKindColumnLeft, kDETSublistEntryBottom, kKindColumnRight},
      kDETEnabled+kDETDynamicSize, kDETPrKind, EditText {kDETDefaultFont, kDETDefaultFontSize,
      kDETLeft, kDETNormal};
   }
};
```

DEFINING THE NEW KEYWORD RECORD TYPE

The aspect and information page templates are now defined, adding a new information page for User records. However, there's still work to do before the user can create keyword records — you need to provide aspect and information page templates for this new record type.

MAIN ASPECT TEMPLATE FOR KEYWORD RECORDS

Since the new record type appears in a sublist, you need to provide a main aspect template (as shown in Listing 2).

A main aspect template is similar to the aspect template defined earlier, but it includes some additional resources:

- the menu item text for the Finder's Catalog menu for creating keyword records
- the name the CE should assign to newly created records of this type
- an icon family for the icon, the record kind, and the kind of an alias to a record, as they are to be displayed in a sublist
- one or more categories that this record type belongs to
- Balloon Help strings for the record and for aliases to the record

Listing 2. Main aspect template for keyword records

```
resource 'deta' (kKeywordAspect, purgeable) {
   0,                // Drop-operation order
   dropCheckAlways,  // Drop-check flag
   isMainAspect      // Is the main aspect
};

resource 'rstr' (kKeywordAspect+kDETTemplateName, purgeable) {
   "ACFC Keywords Aspect"
};

resource 'rstr' (kKeywordAspect+kDETRecordType, purgeable) {
   "ACFC Keywords"
};

resource 'rstr' (kKeywordAspect+kDETAspectKind, purgeable) {
   "Keyword"
};

resource 'rst#' (kKeywordAspect+kDETAspectCategory, purgeable) {
   {
   "Miscellaneous"
   }
};

// Icons
include "Keywords.rsrcs" 'ICN#'(128) as 'ICN#'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'icl4'(128) as 'icl4'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'icl8'(128) as 'icl8'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'ics#'(128) as 'ics#'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'ics4'(128) as 'ics4'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'ics8'(128) as 'ics8'(kKeywordAspect+kDETAspectMainBitmap, purgeable);
include "Keywords.rsrcs" 'SICN'(128) as 'SICN'(kKeywordAspect+kDETAspectMainBitmap, purgeable);

include "KeywordsInfoPageAspect.code" 'code'(1) as 'detc'(kKeywordAspect+kDETAspectCode, purgeable);

// Drag and drop resources
resource 'rstr' (kKeywordAspect+kDETAspectDragInString, purgeable) {
   "Add “^3” to “^2”?"
};

resource 'rst#' (kKeywordAspect+kDETAspectRecordDragIn, purgeable) {
   {
   "aoce User", kMemberAttrTypeBody
   }
};

// View menu and Balloon Help resources
resource 'rstr' (kKeywordAspect+kDETAspectNewMenuName, purgeable) {
   "New Keyword"
};
```

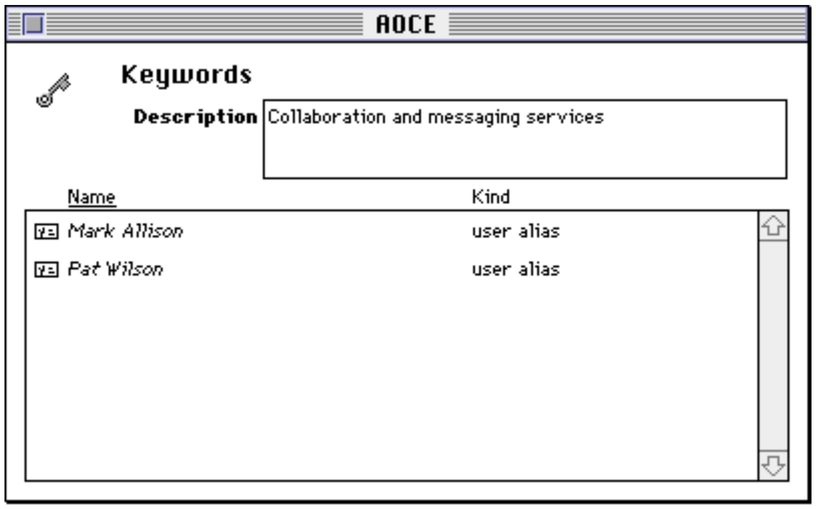
(continued on next page)

Listing 2. Main aspect template for keyword records *(continued)*

```
resource 'rstr' (kKeywordAspect+kDETAspectNewEntryName, purgeable) {
   "Untitled Keyword"
};

resource 'rst#' (kKeywordAspect+kDETAspectBalloons, purgeable) {
   {
   "Keyword description", "Keyword description : uneditable because the record is locked or "
   "access controlled."
   }
};

resource 'rst#' (kKeywordAspect+kDETWhatIs, purgeable) {
   {
   "Keyword \n \nA keyword record. Open this icon to see a description of this keyword and a list "
   "of people who have this job skill."
   }
};

resource 'rst#' (kKeywordAspect+kDETAliasWhatIs, purgeable) {
   {
   "Keyword alias \n \nAn alias to a keyword record. Open this alias to see a description of this "
   "keyword and a list of people who have this job skill."
   }
};

resource 'detm' (kKeywordAspect+kDETAspectViewMenu, purgeable) {
   kKeywordAspect+kDETAspectViewMenu,
   {
   kDETPrName, "by Name";
   kDETPrKind, "by Kind";
   }
};

// Lookup table
resource 'dett' (kKeywordAspect+kDETAspectLookup, purgeable) {
   {
   {"ACFC Keyword Description"}, typeRString,
      useForInput, useForOutput, notInSublist, isNotAlias, isNotRecordRef,
      {'rstr', prDescription, 0};
   {kMemberAttrTypeBody}, typePackedDSSpec,
      notForInput, notForOutput, useInSublist, isAlias, isNotRecordRef,
      {};
   }
};
```

INFORMATION PAGE TEMPLATE FOR KEYWORD RECORDS

Upon double-clicking a keyword record, the user expects an information page to appear, as shown in Figure 4. Defining the information page template for the keyword record is similar to defining the information page template for the User record, as described earlier. You include signature, name, type, and optional resources along with the view lists. The resource definitions are provided on this issue's CD.



Collaboration and messaging services

Name	Kind
Mark Allison	user alias
Pat Wilson	user alias

Figure 4. Keyword record information page

A CODE RESOURCE TO SYNCHRONIZE SUBLISTS

As illustrated earlier in Figure 3, the sublist in the new User record information page lists the areas of expertise of the employee described by that record. Each area of expertise is represented by an alias to a keyword record. The sublist in a keyword record lists all the employees who have expertise in the area described by that keyword record (as shown in Figure 4). To keep these two lists synchronized, both the aspect template for the new information page we added to the user template and the keyword record's aspect template include a code resource. Each time someone adds a User record to a keyword record's sublist, the keyword record's code resource adds that keyword to the User record's sublist; each time someone adds a keyword record to a User record's sublist, the User record's code resource adds that user to the keyword record's sublist.

The code resources of the two templates are exactly the same because the structure of the attributes in both the records is identical, as described in the lookup table entry of the two aspect templates.

The following line adds the code resource to the aspect template:

```
include "UserInfoPageAspect.code" 'code'(1) as
   'detc'(kEInfoPageAspect+kDETAspectCode, purgeable);
```

The CE calls code resources when certain events occur, such as a change in an attribute or a drag and drop action. If the code resource doesn't handle the event, it must return a kDETDidNotHandle result code; if it successfully handles the event, it returns noErr. The CE calls the code resource's main routine, passing it a pointer to a parameter block (see Listing 3). This call block indicates which event occurred and contains additional parameters specific to the event.

DODROPQUERY

The CE calls your code routine with a drop-query command (that is, with the kDETcmdDropQuery selector) when the user drops an object on the object that your aspect template applies to. If you want your code resource to handle the drop operation, return a value that's in the developer property-value range (that is, kDETFirstDevProperty through 249) in the commandID field of the call block.

Listing 3. The code resource's main routine

```
#define prChangeRec   kDETFirstDevProperty

/* Entry point called by the CE */
pascal OSErr KeywordsIP (DETCallBlock* callBlockPtr)
{
   OSErr err = noErr;

   if (callBlockPtr->protoCall.target.selector == kDETSelf ||
         callBlockPtr->protoCall.target.selector == kDETSublistItem)
      switch (callBlockPtr->protoCall.reqFunction) {
         case kDETcmdInit:
            /* Call-for masks */
            callBlockPtr->init.newCallFors = kDETCallForCommands +
               kDETCallForDrops;
            break;
         case kDETcmdDropQuery:
            err = DoDropQuery(callBlockPtr);
            break;
         case kDETcmdPropertyCommand:
            err = DoCommand(callBlockPtr);
            break;
         default:
            err = kDETDidNotHandle;
            break;
         }
   else err = kDETDidNotHandle;
   return err;
}
```

In our example (shown in Listing 4), we accept a record when the user drops it on a record that the template applies to by checking the commandID parameter provided by the CE and accepting drops only when the commandID value is kDETAlias. For other values, we inform the CE that we don't manage these cases by returning the constant kDETDidNotHandle. When we accept the drop operation, DoDropQuery returns the value prChangeRec in response to the drop-query command. When it receives a property number in response to the drop-query command, the CE calls the code resource again, this time with a property command (that is, with the selector kDETcmdPropertyCommand).

The property command's call block includes the property number specified in response to the drop-query command. The property-command code can use this property number as a routine selector. In our example, DoCommand checks to make sure the property number is prChangeRec. If it is, DoCommand calls DoAddRecord (more on this in a moment).

Most of the time, a resource doesn't handle all the CE events; therefore, to avoid the overhead resulting from frequent calls to the code resource, each template has a "call-for" mask that indicates which events to invoke it for. You must return the call-for mask when the CE calls the code resource with the kDETcmdInit selector. The kDETcmdInit case of the switch statement in Listing 3 returns a call-for mask that indicates that the code resource should be called only for property commands and drop queries.

Listing 4. DoDropQuery and DoCommand

```
/* Called when an object is dropped onto a record. Just set up a
   property number (prChangeRec) to be sent to the target aspect. */
OSErr DoDropQuery (DETCallBlock* callBlockPtr)
{
   DETTargetSelector theDET;
   /* What is the best guess of the CE regarding the dropped object? */
   if (((DETDropQueryBlock*)callBlockPtr)->commandID == kDETAlias) {
      /* Make an alias of the dropped object. */
      ((DETDropQueryBlock*)callBlockPtr)->copyToHFS = false;
      ((DETDropQueryBlock*)callBlockPtr)->commandID = prChangeRec;
      return noErr;
      }
   else
      return kDETDidNotHandle;
}

/* Called when the CE sends a property command. We check that it's
   the property we expect. */
OSErr DoCommand (DETCallBlock* callBlockPtr)
{
   OSErr err = noErr;

   /* Check whether called for the prChangeRec command. */
   switch (((DETPropertyCommandBlock*)callBlockPtr)->property) {
      case prChangeRec:
         err = DoAddRecord(callBlockPtr);
         break;
      default:
         err = kDETDidNotHandle;
         break;
      }
   return err;
}
```

DOADDRECORD
When the CE calls DoCommand with the property number prChangeRec, DoCommand calls DoAddRecord (Listing 5). DoAddRecord needs to add data to a record, so it must first identify which record is the target of the drop. To do so, it calls the CE's kDETcmdGetDSSpec callback routine, passing it the kDETSelf target selector. DoAddRecord then extracts the reference number of the personal catalog and the record ID from the record's DSSpec that was returned by the callback routine. To find out how many objects were dropped (that is, for how many dropped objects the drop-query routine returned the same property number), DoAddRecord calls the CE's kDETcmdGetCommandSelectionCount callback routine.

DoAddRecord calls the CE's kDETcmdGetCommandItemN callback routine for each object dropped. Dropped objects can be of different types, such as catalog items, files, and letters. In our example, the only kind of objects we want to add to our sublist are keyword records and User records, so DoAddRecord requests only information of type kDETDSType to get a packed DSSpec for each dropped record. If the targeted record is a User record, DoAddRecord checks to make sure the dropped record is a keyword record. If the targeted record is a keyword record,

Listing 5. DoAddRecord

```
/* When record A is dragged onto record B, this function updates record A to store
   an alias to record B and updates record B to store an alias to record A. Updates
   occur only if records are of the required type. */
OSErr DoAddRecord (DETCallBlock* callBlockPtr)
{
   DETCallBackBlock  cbb, cbb1, cbb2;
   short             PABrefNum;
   RecordID          targetRID, receivedRID;
   LocalIdentity     userLocalId;
   PackedDSSpec      *targetDSSpec, *droppedDSSpec;
   DSSpec            dsp, dsp1;
   OSErr             err;
   long              count, i;
   Str255            targetAttrType, droppedAttrType;

#ifdef USER
   Str255            theStr = "ACFC Keywords";
   RString           recType;
   OCECToRString(theStr, smRoman, &recType, kRStringMaxBytes);
#endif

   /* Find out target record DSSpec. */
   cbb.getDSSpec.target.selector = kDETSelf;
   cbb.getDSSpec.reqFunction = kDETcmdGetDSSpec;
   err = CallBackDET(callBlockPtr, &cbb);
   if (err != noErr)
      return err;

   HLock((Handle) cbb.getDSSpec.dsSpec);
   targetDSSpec = *(cbb.getDSSpec.dsSpec);
   /* Get record ID. */
   PABrefNum = cbb.getDSSpec.refNum;
   userLocalId = cbb.getDSSpec.identity;
   OCEUnpackDSSpec(targetDSSpec, &dsp, &targetRID);

   /* Find out how many records have been dropped. */
   cbb1.getCommandSelectionCount.reqFunction = kDETcmdGetCommandSelectionCount;
   err = CallBackDET(callBlockPtr, &cbb1);
   if (err != noErr) {
      DisposeHandle((Handle) cbb.getDSSpec.dsSpec);
      return err;
      }

   count = cbb1.getCommandSelectionCount.count;
   for (i = 1; i <= count; i++) {
      /* Get the DSSpec of dropped record. */
      cbb1.getCommandItemN.reqFunction = kDETcmdGetCommandItemN;
      cbb1.getCommandItemN.itemNumber = i;
      cbb1.getCommandItemN.itemType = kDETDSType;
      err = CallBackDET(callBlockPtr, &cbb1);
```

(continued on next page)

Listing 5. DoAddRecord *(continued)*

```
      if (err == noErr) {
         HLock((Handle) cbb1.getCommandItemN.item.ds.dsSpec);
         droppedDSSpec = *(cbb1.getCommandItemN.item.ds.dsSpec);
         OCEUnpackDSSpec(droppedDSSpec, &dsp1, &receivedRID);
         /* Check type of record. */
      #ifdef USER
         if (OCEEqualRString(receivedRID.local.recordType, &recType, kOCERecordType)) {
            strcpy(targetAttrType, kMemberAttrTypeBody);
            strcpy(droppedAttrType, "ACFC Alias keyword");
      #else
         if (OCEEqualRString(receivedRID.local.recordType, OCEGetIndRecordType(kUserRecTypeNum),
                          kOCERecordType)) {
            strcpy(targetAttrType, "ACFC Alias keyword");
            strcpy(droppedAttrType, kMemberAttrTypeBody);
      #endif
            /* Update target record to set up dropped record as an attribute of this record. */
            AddRecordAsAttribute(userLocalId, droppedDSSpec, &targetRID, PABrefNum, droppedAttrType);
            /* Update dropped record to set up target record as an attribute of the dropped record. */
            AddRecordAsAttribute(userLocalId, targetDSSpec, &receivedRID, PABrefNum, targetAttrType);
            }
         HUnlock((Handle) cbb1.getCommandItemN.item.ds.dsSpec);
         DisposeHandle((Handle) cbb1.getCommandItemN.item.ds.dsSpec);
         }
      else
         break;
      }

   if (err == noErr) {
      /* Ask for immediate update. */
      cbb2.requestSync.target = ((DETPropertyCommandBlock*)callBlockPtr)->target;
      cbb2.requestSync.reqFunction = kDETcmdRequestSync;
      err = CallBackDET(callBlockPtr, &cbb2);
      }
   DisposeHandle((Handle) cbb.getDSSpec.dsSpec);
   return err;
}
```

DoAddRecord checks to make sure the dropped record is a User record. It uses the AOCE utility routine OCEEqualRString to check the record type.

ADDRECORDASATTRIBUTE

If the dropped record is the right type, DoAddRecord calls AddRecordAsAttribute, which is shown in Listing 6. AddRecordAsAttribute calls the Catalog Manager's DirAddAttributeValue routine to add the dropped object to the record. The parameter block for DirAddAttributeValue includes parameters that identify the catalog containing the record to be modified, the record itself, and the attribute to be added. The attribute specification includes the attribute type, the attribute value tag, and the attribute value.

To maintain the synchronization between the User record and the keyword record, you also need to update the dropped record, adding the target record as an attribute

Listing 6. AddRecordAsAttribute

```
/* This routine adds an attribute as a DSSpec to a record. */
void AddRecordAsAttribute(LocalIdentity userLocalId, PackedDSSpec* theDSSpec,
                          RecordIDPtr updatedRecord, short refNum, Ptr attrType)
{
   OSErr          err;
   Attribute      theAttribute;
   DirParamBlock  dspb;
   AttributeType  kwRType;

   /* Prepare the attribute; set up its type and the data within the attribute. */
   OCECToRString(attrType, smRoman, (RString *) &kwRType, kAttributeTypeMaxBytes);
   theAttribute.attributeType = kwRType;
   theAttribute.value.tag = typePackedDSSpec;
   theAttribute.value.dataLength = theDSSpec->dataLength+sizeof(theDSSpec->dataLength);
   theAttribute.value.bytes = (Ptr) theDSSpec;

   /* Prepare the parameter block used by the Catalog Manager to add an attribute. */
   *(long *)&dspb.addAttributeValuePB.serverHint = nil;
   dspb.addAttributeValuePB.dsRefNum = refNum;         /* refNum of a personal catalog */
   dspb.addAttributeValuePB.identity = userLocalId;
   dspb.addAttributeValuePB.aRecord = updatedRecord;    /* Record to be modified */
   dspb.addAttributeValuePB.attr = &theAttribute;       /* Attribute to be added */
   dspb.addAttributeValuePB.clientData = 0;
   err = DirAddAttributeValue(&dspb, false);
}
```

of the dropped one. You can use the same routine — AddRecordAsAttribute — because the structure of the attributes in the two records is the same — the record to be updated is now the dropped record and the attribute value is the DSSpec of the target record.

SUMMARY

To summarize what happens when a drag and drop action occurs:

1. The user drops a bunch of icons on a closed record or an information page. If the drop is on an information page, that page is the only possible destination; if it's a closed record, all the information pages are potential destinations.
2. For each icon/possible-destination-aspect pair, the CE looks at the template resources and calls the code resource (if there is one).
3. Based on the information returned by the code resources and any drag and drop resources present, the CE decides what operation is desired for each pair.
4. Based on the drop-operation order number, the CE picks one destination and one operation for each icon.
5. For each group of icons with the same destination and operation, the CE performs that operation at that destination. If the operation is copy, move, or make alias, the CE handles it. If the operation is a property command (as in this article), the CE calls the code resource to perform the operation.

USING THE TEMPLATES

When you compile the templates, you get a PowerTalk extension that can be installed in the Extensions folder of the System Folder. Users who install these templates can display and edit new information about users in User records in their personal catalogs. The system administrator or anyone who has sufficient access privileges can use the new keyword records and the new User record information page to maintain this information for everyone who has a PowerShare account.

The benefit in extending User records is that all the information regarding a person is stored in the same place and therefore is very easy to retrieve. The work involved in developing this customized solution is much less than that typically involved in developing a database. What's more, AOCE catalogs are part of system software — why pay more for new database software and servers?

AND NOW LET'S DREAM

This article has shown a relatively easy way to extend the templates that come built into PowerTalk. As you become more familiar with AOCE templates, you'll undoubtedly think of more complex things you can do with templates and template code resources.

The next article, "Exploring Advanced AOCE Templates Through Celestial Mechanics," describes one imaginative use of templates. Another use might be to define different kinds of templates for different people, giving them access to different types of data in the same catalog.

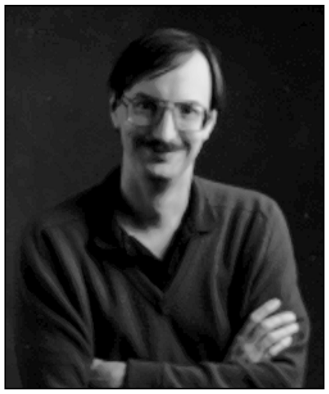
Imagine a traditional library using a catalog to store references to books: A specialized template could provide information on a book, including keywords related to the topics in the book. A keyword template would let users browse the catalog and find all the books available on a specific topic. And to dream a bit further, the catalog could be a very large database on a distant system that you access by dialing up. It would contain all the references in the world of existing books, and you could consult it from your home, just by browsing the catalog through the Finder. The book information page might even contain a button that could open the electronic version of the book.

So go ahead and take advantage of AOCE catalogs. Apply what you've learned about expanding records through AOCE templates and begin building the world of connectivity and information sharing that you dream about.

Thanks to our technical reviewers David Akhond, Paul Black, Harry Chesley, and Steve Falkenburg. Special thanks to Paul Black for writing the background material on AOCE catalogs and templates.•

Exploring Advanced AOCE Templates Through Celestial Mechanics

PowerTalk provides AOCE catalogs to store and edit collections of information. The Catalogs Extension to the Finder lets you use AOCE templates to extend the types of information stored and the means of editing it, which makes the catalogs open-ended rather than limited to the information types provided by Apple with the PowerTalk software. This article explores several advanced features of AOCE templates, showing how new types of entries can be added that store information about planets and calculate their current locations and orbits.


HARRY R. CHESLEY

The AOCE Catalogs Extension (CE) — an extension to the Finder and one of the PowerTalk components — was originally conceived as an open-ended means of providing addresses for PowerTalk mail and messaging; however, it goes well beyond that original goal. The CE allows third-party developers to extend the Finder in a variety of ways, including providing new catalog entry types, new views on the contents of entries, new means of editing those contents, runtime calculation of information to be displayed, and new actions to perform in the case of drag and drop and double-click operations. AOCE templates, which serve as the extension mechanism, provide resources and code that define the format, appearance, and functionality of catalog entries.

Because this article explores advanced features of the AOCE template mechanism, we assume some familiarity with AOCE catalogs and a basic understanding of AOCE templates and the terms used to describe them. The article "Getting the Most out of AOCE Catalog Records" in this issue gives an overview of AOCE catalogs and templates. For in-depth information, the definitive reference is *Inside Macintosh: AOCE Application Interfaces.*

In this article, we demonstrate how the template mechanism can be extended to plot the orbits of the planets. For those of you who aren't interested in celestial mechanics and could care less about the mathematics involved in calculating the position of a celestial body, don't worry — the article focuses on templates; you can skip the details on celestial mechanics without limiting your understanding. But if you are interested, see "Algorithms for Calculating Planetary Positions."

HARRY R. CHESLEY There are two mysteries that have always — well — mystified Harry: (1) Why do mirrors exchange left and right but not top and bottom? (2) What is consciousness? Harry recently worked out the answer to the first question. You reverse the scene yourself by turning around to look through the mirror rather than directly at it. If you'd turned head-over-heels instead of around, the scene would be top and bottom exchanged but not right and left exchanged. Given this resolution, Harry feels the answer to the second question can't be far behind. Meanwhile, Harry works in Apple Online Services, doing Newton programming.•

ALGORITHMS FOR CALCULATING PLANETARY POSITIONS

Here we discuss the parameters and algorithms used for calculating the positions of the planets. Orbits are three dimensional, but for our purpose — plotting the orbit from an overhead perspective — we need only two dimensions. Extending the templates to three dimensions is an excellent exercise for the reader.

The parameters needed for calculating a planet's orbit are as follows:

Symbol	**Meaning**
T_p	Period (tropical years)
ε	Longitude at epoch (degrees)
ϖ	Longitude at the perihelion (degrees)
e	Eccentricity of the orbit
a	Semi-major axis of the orbit

These parameters are for the epoch 1990 January 0.0. We use them to calculate a series of intermediate values, leading up to calculating the x and y coordinates used to plot the planet's position for the specified date and time. Table 1 below shows the actual values of the orbital parameters for each of the planets.

To begin the calculations, we need to know how many days (d) it has been since the start of the epoch. The epoch actually starts on midnight between December 30 and 31, 1989. This may seem confusing, but it simplifies some of the calculations. Thus, 6 A.M., January 5, 1990, is six days and six hours since the start of the epoch, or 6.25 days.

Next, we need to find the true anomaly (v), which is the angle the planet makes with the line between the sun and the perihelion (the point nearest the sun in the planet's orbit). To find it, we first calculate the mean anomaly (m), which would be the true anomaly if the planet's orbit were circular.

$$m = \frac{360}{365.242191} \times \frac{d}{T_p} + \varepsilon - \varpi \text{ degrees}$$

$$v = m + \frac{360}{\pi} e \sin m \text{ degrees}$$

The heliocentric longitude (l) is

$$l = v + \varpi$$

Now that we know the angle of the planet, all we need is the distance given by the radius vector (r).

$$r = \frac{a(1 - e^2)}{1 + e \cos v}$$

From here it's simple trigonometry to get the x and y coordinates:

$$x = r \cos l$$
$$y = r \sin l$$

You'll see these calculations later in the code.

Warning: While the above calculations are perfectly sufficient to tell you which window to look out of to see Mars, they may lack something if your object is to actually reach Mars. For this reason, readers with their own spacecraft should not count on these formulas, or the resulting templates, for purposes of celestial navigation.

Table 1. Orbital parameter values

Planet	T_p	ε	ϖ	e	a
Mercury	0.240852	60.750646	77.299833	0.205633	0.387099
Venus	0.615211	88.455855	131.430236	0.006778	0.723332
Earth	1.00004	99.403308	102.768413	0.016713	1.00000
Mars	1.880932	240.739474	335.874939	0.093396	1.523688
Jupiter	11.863075	90.638185	14.170747	0.048482	5.202561
Saturn	29.471362	287.690033	92.861407	0.055581	9.554747
Uranus	84.039492	271.063148	172.884833	0.046321	19.21814
Neptune	164.79246	282.349556	48.009758	0.009003	30.109570
Pluto	246.77027	221.4127	224.133	0.24624	39.3414

We begin by developing a set of templates that plot the positions and orbits of the planets at a specified time. A sublist on one of the record information pages lists the planets and their positions. We also develop templates to display information pages for each planet; these pages enable the user to enter the information needed to calculate a planet's orbit. The calculations and plotting are performed by code resources in the templates. Using the techniques described in the article, you could add other types of celestial bodies (such as comets, moons, and alien spacecraft) that would be defined by a different set of parameters and have a different algorithm for calculating position and orbit.

Although the templates are quite straightforward in general, the article focuses on the code resources that implement three advanced features of the template mechanism:

- type conversion between text (RStrings) and custom, internal data types — to display and edit floating-point numbers and date/time information
- automatic calculation of property values when other selected property values change — to update the planet's position when the time or orbital parameters change
- drawing in a custom view — to display the plotted object positions and orbits

DEFINING THE TEMPLATES

The templates we create define a record type of "hrc Orbits" to hold the list of planets we want to display. The record contains an attribute type "hrc Planet" with one attribute value per planet and an attribute value tag of 'plnt'. There's also a single-valued attribute of type "hrc Orbits info," which holds information pertinent to the orbits record.

Using an attribute value tag allows for future expansion to new types of objects — spacecraft, for instance. In the example, the aspect template for the attribute type "hrc Planet" is used only for attribute values with the attribute value tag 'plnt'. To add a new type of object, which may require different orbital parameters and a different algorithm to calculate the orbits, you would use a different tag. For example, an attribute value that describes a spacecraft might have an attribute value tag of 'crft'.

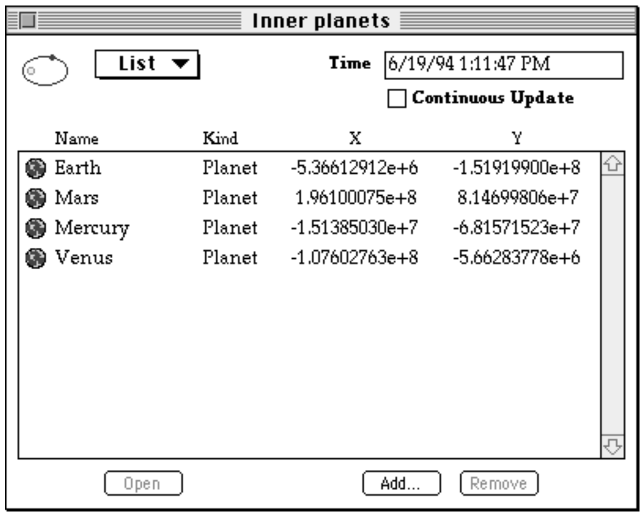
We need to define the following templates:

- information page templates for the orbits record (record type "hrc Orbits")
- information page templates for the attribute type "hrc Planets," which is the attribute type of the sublist entries
- an aspect template for the record type "hrc Orbits"
- an aspect template for the attribute type "hrc Planets"

These templates are included on this issue's CD. There's nothing remarkable about most of them. This article discusses only those portions of the templates that are more interesting and unusual.

ORBITS RECORD INFORMATION PAGE TEMPLATES

We use two information pages to display the information stored in an orbits record (Figure 1). The List information page contains a sublist of planets (attribute type "hrc Planet"), allowing the user to create new planets and drag existing ones into and out

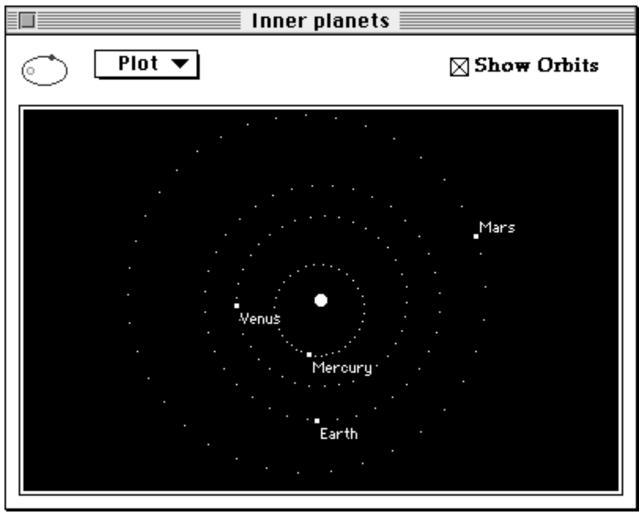


Time 6/19/94 1:11:47 PM

☐ Continuous Update

Name	Kind	X	Y
Earth	Planet	-5.36612912e+6	-1.51919900e+8
Mars	Planet	1.96100075e+8	8.14699806e+7
Mercury	Planet	-1.51385030e+7	-6.81571523e+7
Venus	Planet	-1.07602763e+8	-5.66283778e+6

List



Plot

Figure 1. Information pages for the orbits record

of the list. Besides an icon, name, and kind, the sublist on the List page displays x and y coordinates for each planet. This is the location at the time given in the field at the top of the page. The user can edit the time to see past and future positions. The Continuous Update checkbox, when checked, causes the Time field to be constantly updated to the current time. The state of this checkbox is kept in the "hrc Orbits info" attribute of the orbits record.

The Plot information page contains a plot of the position of each planet in the sublist on the List information page. When the Show Orbits checkbox is checked, the plot shows not only the position of each planet, but also the future track — the orbit — of the planet. Orbital calculations take a lot of time, especially on slower systems, so the user can choose whether or not to display this information.

Listing 1 shows resource definitions for the Plot information page template. Note that kOrbitsCustomViewProperty is used for the property number for both the Show Orbits checkbox and the custom view that plots the positions. Normally two views don't share the same property. Using the same one here causes an automatic redraw

Listing 1. Plot information page template

```
resource 'deti' (kOrbitsPlotPage, purgeable) {
   2000, kDETNoSublistRect, noSelectFirstText,
   {
   kDETNoProperty, kDETNoProperty, kOrbitsPlotPage;
   },
   {}
};

resource 'rstr' (kOrbitsPlotPage+kDETTemplateName, purgeable) {
   "hrc Orbits plot page"
};
```

(continued on next page)

Listing 1. Plot information page template *(continued)*

```
resource 'rstr' (kOrbitsPlotPage+kDETRecordType, purgeable) {
   kOrbitsRecordType
};

resource 'rstr' (kOrbitsPlotPage+kDETInfoPageName, purgeable) {
   "Plot"
};

resource 'rstr' (kOrbitsPlotPage+kDETInfoPageMainViewAspect,
   purgeable) {
   "hrc Orbits main aspect"
};

resource 'detv' (kOrbitsPlotPage, purgeable){
   {
   kDETSubpageIconRect, kDETNoFlags, kDETAspectMainBitmap,
   Bitmap {kDETLargeIcon};

   {12, kOrbitsPageWidth-120, 28, kOrbitsPageWidth-8},
      kDETNoFlags, kOrbitsCustomViewProperty,
      CheckBox {kPalatino, 12, kDETLeft, kDETBold,
                  "Show Orbits", kOrbitsCustomViewProperty};

   {44, 8, kOrbitsPageHeight-8, kOrbitsPageWidth-8}, kDETNoFlags,
      kDETNoProperty, Box {kDETUnused};

   {47, 11, kOrbitsPageHeight-11, kOrbitsPageWidth-11}, kDETNoFlags,
      kOrbitsCustomViewProperty, Custom {kDETUnused};
   }
};
```

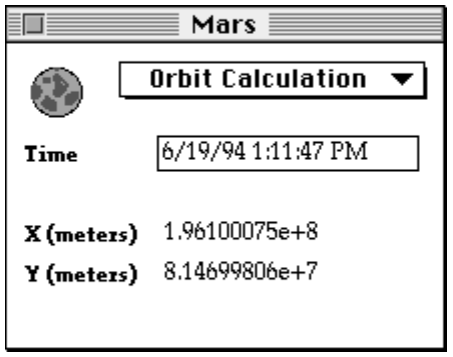
of the custom view when the checkbox changes. This is simpler than using the code resource to intercept the property-dirtied call resulting from the checkbox change and using a dirty-property callback to cause the custom view to be redrawn. (Whenever a property is changed, a kDETcmdPropertyDirtied call is made to the code resource.) The bulk of the work for the custom view occurs in the code resource, as described later in the section “Drawing in a Custom View.”

PLANET INFORMATION PAGE TEMPLATES

When the user double-clicks a planet in the sublist, a window opens with two more information pages (Figure 2). The Orbit Calculation information page displays the position of the planet at a user-specified time. The Orbit Parameters information page displays, and lets the user enter, the values for the orbital parameters (shown earlier in Table 1). These two pages could have been combined, but most users aren’t interested in seeing the orbital parameter values once they’ve been entered. They just clutter up the interesting information — the planet’s location at a given time.

ORBITS RECORD ASPECT TEMPLATE

We define one aspect template for the orbits record (record type “hrc Orbits”) — a main aspect that also serves as the main view aspect for the orbits record information pages. The aspect for the orbits record contains the properties listed in Table 2.


Mars

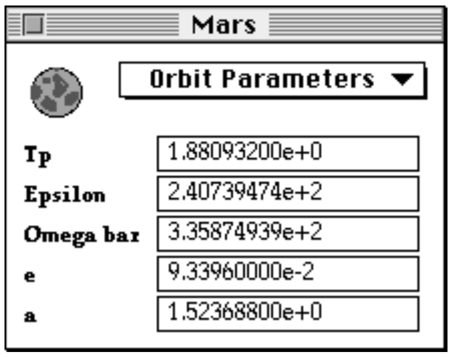
Orbit Calculation

Time 6/19/94 1:11:47 PM

X (meters) 1.96100075e+8

Y (meters) 8.14699806e+7

Orbit Calculation


Mars

Orbit Parameters

Tp 1.88093200e+0

Epsilon 2.40739474e+2

Omega bar 3.35874939e+2

e 9.33960000e-2

a 1.52368800e+0

Orbit Parameters

Figure 2. Information pages for a planet

Table 2. Properties in the orbits record's aspect

Property	**Meaning**
kOrbitsTimeProperty	The time to use when calculating positions, as entered by the user in the Time field on the List information page
kOrbitsNowProperty	The value of the Continuous Update checkbox
kOrbitsCustomViewProperty	The property number associated with the custom view that plots the planets' positions and with the value of the Show Orbits checkbox, both on the Plot information page

The kOrbitsNowProperty property and the entries in the sublist are stored in the record, as specified by the 'dett' lookup table resource (shown below). You'll find the full source code for the orbits record aspect template on this issue's CD.

```
resource 'dett' (kOrbitsMainAspect+kDETAspectLookup, purgeable) {
   {
   {kOrbitsAttributeType}, typeBinary,
      useForInput, useForOutput, notInSublist, isNotAlias, isNotRecordRef,
      {
      'long', kOrbitsNowProperty, 0;
      };
   {kPlanetAttributeType}, 'plnt',
      notForInput, notForOutput, useInSublist, isNotAlias, isNotRecordRef,
      {};
   }
};
```

ASPECT TEMPLATE FOR ATTRIBUTE TYPE "HRC PLANET"

The aspect template for attribute type "hrc Planet" is also a main aspect template. The properties defined by this aspect are shown in Table 3.

The orbital parameters, as well as the name of the attribute value (for example, "Mercury" or "Venus"), are stored in the attribute value, so they're included in the 'dett' resource:

```
#define kExtendedPropertyType 2
#define kExtendedPropertyTypeSize 10
...
```

```
resource 'dett' (kPlanetMainAspect+kDETAspectLookup, purgeable) {
   {
   {kPlanetAttributeType}, 'plnt',
      useForInput, useForOutput, notInSublist, isNotAlias, isNotRecordRef,
      {
      'rstr', kDETAspectName,    0;
      'btyp', kDETNoProperty,    kExtendedPropertyType;
      'blok', kTpProperty,       kExtendedPropertyTypeSize;
      'blok', kEpsilonProperty,  kExtendedPropertyTypeSize;
      'blok', kOmegaBarProperty, kExtendedPropertyTypeSize;
      'blok', keProperty,        kExtendedPropertyTypeSize;
      'blok', kaProperty,        kExtendedPropertyTypeSize;
      };
   }
};
```

Table 3. Properties in the "hrc Planet" attribute type's aspect

Property	Meaning
kTimeProperty	The time, as entered by the user in the Time field
kXProperty	The x coordinate at that time
kYProperty	The y coordinate at that time
kTpProperty	The T_p orbital parameter
kEpsilonProperty	The ε orbital parameter
kOmegaBarProperty	The $\bar{\omega}$ orbital parameter
keProperty	The e orbital parameter
kaProperty	The a orbital parameter

Each of the properties in the 'dett' resource except kDETAspectName has a template-defined custom property type of 2 (kExtendedPropertyType) and is 10 (kExtendedPropertyTypeSize) bytes in size. The actual format is that of the standard SANE floating-point extended type. The 'btyp' element specifies that all subsequent 'blok' elements should produce properties of the type given (kExtendedPropertyType). The 'blok' elements that follow specify a fixed-size block, kExtendedPropertyTypeSize bytes in size. The next section describes how these property types get used.

As with the main aspect template for the orbits, the rest of this template is quite simple and is included on the CD.

CUSTOM PROPERTY TYPE CONVERSION

The templates we're defining use two property types that aren't supported directly by the CE: SANE floating-point extended, for orbital parameters and positions, and date/time, for specifying the time for which the positions should be calculated. In addition to using these property types for internal calculations, we want to display them and let the user edit them. To do this, we display the items in text fields and supply a code resource that translates between the internal representation of the custom property types and text (RStrings). The code resource implements convertToRString and convertFromRString when called by the CE. The part of the Planet routine that figures out when to call the conversion functions is as follows:

```
#define kTimePropertyType 1
#define kTimePropertyTypeSize 8
```

```
pascal OSErr Planet(DETCallBlockPtr callBlockPtr)
{
   if (callBlockPtr->protoCall.target.selector == kDETSelf)
      switch (callBlockPtr->protoCall.reqFunction) {
         ...
         case kDETcmdConvertToRString:
            return convertToRString(callBlockPtr);
         case kDETcmdConvertFromRString:
            return convertFromRString(callBlockPtr);
         ...
         }
   return kDETDidNotHandle;
}
```

In each case, the conversion function in the code resource first gets the type of the property being converted — either kTimePropertyType or kExtendedPropertyType — and then performs the conversion appropriate to that property type. The code in Listing 2 is for the convertToRString case; code for convertFromRString performs the opposite conversion, taking an RString and turning it into a custom property type.

Listing 2. Converting custom property types to a text string

```
OSErr convertToRString(DETCallBlockPtr callBlockPtr)
{
   DETConvertToRStringBlock* ctrs;
   DETGetPropertyTypeBlock   gpt;

   ctrs = &(callBlockPtr->convertToRString);

   // Get the type of the property being converted.
   gpt.reqFunction = kDETcmdGetPropertyType;
   gpt.target = ctrs->target;
   gpt.property = ctrs->property;
   if (CallBackDET(callBlockPtr, (DETCallBackBlock*) &gpt) == noErr) {
      char          s[256];
      RStringHandle  h;

      // Convert time property types.
      if (gpt.propertyType == kTimePropertyType) {
         LongDateTime   ldt;
         char           tStr[256];

         // Get the current value.
         ldt = GetTimeProperty(callBlockPtr, ctrs->property);

         // Convert it to a string.
         iuldatestring(&ldt, shortDate, s, nil);
         tStr[0] = ' '; tStr[1] = 0;
         strcat(s, tStr);
         iultimestring(&ldt, true, tStr, nil);
         strcat(s, tStr);
         }
```

(continued on next page)

Listing 2. Converting custom property types to a text string *(continued)*

```
      // Convert floating-point extended property types.
      else if (gpt.propertyType == kExtendedPropertyType) {
         extended n;
         decform  df;
         decimal  d;

         // Get the current value.
         n = GetExtendedProperty(callBlockPtr, ctrs->property);

         // Convert it to a string.
         df.style = FLOATDECIMAL;
         df.digits = 9;
         num2dec(&df, n, &d);
         dec2str(&df, &d, &s);
         }

      // If we don't know the type, don't convert it.
      else return kDETDidNotHandle;

      // Return the string as an RString handle.
      h = (RStringHandle) NewHandle(strlen(s) + sizeof(ProtoRString));
      if (h) {
         HLock((Handle) h);
         OCECToRString(s, smRoman, *h, strlen(s));
         HUnlock((Handle) h);
         ctrs->theValue = h;
         return noErr;
         }
      else return MemError();
      }

   return kDETDidNotHandle;
}
```

Two utility functions retrieve properties of the new types — getTimeProperty and getExtendedProperty. Listing 3 shows getExtendedProperty (getTimeProperty is virtually identical).

The code shown in this section belongs to the aspect template for attribute type "hrc Planet." Similar code is used for the orbits record aspect template, but that template never needs to convert extended types — they're always converted by the "hrc Planet" attribute type template — so only the code for converting times is included.

The CE makes all the decisions about when to perform the conversions. When it needs to display a property in a text field, it calls the code resource to convert the property to text. When the user finishes editing a property and closes the field (by tabbing to the next field, pressing Enter, switching pages, or closing the window), the CE calls the code resource to convert the property from text to the internal type.

The CE knows what type a property is because the template tells it. In the case of properties stored in an attribute value, the 'dett' resource includes the type

Listing 3. getExtendedProperty

```
extended getExtendedProperty(DETCallBlockPtr callBlockPtr,
                             short property)
{
   DETGetPropertyBinaryBlock  gpb;
   extended                   n;

   gpb.reqFunction = kDETcmdGetPropertyBinary;
   gpb.target = callBlockPtr->protoCall.target;
   gpb.property = property;
   if (CallBackDET(callBlockPtr, (DETCallBackBlock*) &gpb) != noErr)
      return 0.0;
   BlockMove(*gpb.propertyValue, (char*) &n, sizeof(n));
   DisposeHandle(gpb.propertyValue);
   return n;
}
```

information, as discussed earlier in the section on the aspect template for attribute type “hrc Planet.”

In the case of temporary properties not stored in an attribute value, for which there is no 'dett' entry, the code resource sets the type, generally while setting the property. For example, in the aspect template for attribute type “hrc Planet” the code resource initializes the Time field to the current time as a part of the instanceInit routine, which is invoked when the code resource is called with the kDETcmdInstanceInit routine selector (Listing 4).

CALCULATING POSITIONS AUTOMATICALLY

The aspect template for attribute type “hrc Planet” calculates the position of the planet at a specified time. It takes the time from kTimeProperty and puts the resulting position in kXProperty and kYProperty. This calculation, which is performed whenever kTimeProperty changes, is used in three places: in the Orbit Calculation information page of each “hrc Planet” attribute value; in the sublist on the List information page of the orbits record; and in calculating where to draw the planets on the Plot information page of the orbits record.

If you want to create another template that implements a different type of celestial body — using a different attribute value tag — the same procedure would work, even though you may use an entirely different algorithm to calculate kXProperty and kYProperty from kTimeProperty. We’re using the template as an object-oriented database: Each object (aspect) is of a specific class (aspect template), which specifies how it should react to certain messages (setting the kTimeProperty property). Portions of the object (properties) are persistent (stored in AOCE catalogs).

To calculate kXProperty and kYProperty from kTimeProperty, we supply code that responds to a kDETcmdPropertyDirtied call, as shown in Listing 5. Note that the code resource also recalculates kXProperty and kYProperty when any of the orbital parameters changes. The functions degsin and degcos are versions of sin and cos that take their parameters in degrees rather than radians. The constant kAU is the size of one astronomical unit (149,600,000.0 meters).

Listing 4. Initializing the Time field in instanceInit

```
OSErr instanceInit(DETCallBlockPtr callBlockPtr)
   {
   DETSetPropertyTypeBlock   spt;
   DETSetPropertyBinaryBlock spb;
   unsigned long             l;
   LongDateCvt               ldt;

   // Set the time property type.
   spt.reqFunction = kDETcmdSetPropertyType;
   spt.target = callBlockPtr->protoCall.target;
   spt.property = kTimeProperty;
   spt.newType = kTimePropertyType;
   CallBackDET(callBlockPtr, (DETCallBackBlock*) &spt);

   // Set the time property to the current time.
   GetDateTime(&l);
   ldt.hl.lHigh = 0; ldt.hl.lLow = l;
   spb.reqFunction = kDETcmdSetPropertyBinary;
   spb.target = callBlockPtr->protoCall.target;
   spb.property = kTimeProperty;
   spb.newValue = (Ptr) &ldt;
   spb.newValueSize = sizeof(ldt);
   if (CallBackDET(callBlockPtr, (DETCallBackBlock*) &spb) == noErr) {
      // Dirty the time property.
      DETDirtyPropertyBlock   dp;

      dp.reqFunction = kDETcmdDirtyProperty;
      dp.target = callBlockPtr->protoCall.target;
      dp.property = kTimeProperty;
      CallBackDET(callBlockPtr, (DETCallBackBlock*) &dp);
      }
}
```

Listing 5. Calculating kXProperty and kYProperty from kTimeProperty

```
// Returns days (including fractions) since 1990.
extended daysSince1990(LongDateTime t)
{
   LongDateRec    ldr;
   LongDateTime   t1990;
   extended       et, et1990;

   et = t;
   ldr.ld.era = 0; ldr.ld.year = 1989; ldr.ld.month = 12;
   ldr.ld.day = 31; ldr.ld.hour = 0; ldr.ld.minute = 0;
   ldr.ld.pm = 0;
   LongDate2Secs(&ldr, &t1990);
   et1990 = t1990;
   return et/(24.0*60.0*60.0) - et1990/(24.0*60.0*60.0);
}
```

(continued on next page)

Listing 5. Calculating kXProperty and kYProperty from kTimeProperty *(continued)*

```
OSErr propertyDirtied(DETCallBlockPtr callBlockPtr)
{
   DETPropertyDirtiedBlock*   pd;

   pd = (DETPropertyDirtiedBlock*) &callBlockPtr->propertyDirtied;
   switch (pd->property) {
      // Recalculate only on selected properties.
      case kTimeProperty:
      case kTpProperty:
      case kEpsilonProperty:
      case kOmegaBarProperty:
      case keProperty:
      case kaProperty:
         {
         DETSetPropertyTypeBlock   spt;
         DETSetPropertyBinaryBlock spb;
         extended                  d, tp, epsilon, omegaBar, e, a;
         extended                  n, m, l, v, r, x, y;

         // Get the orbital parameters.
         d = daysSince1990(GetTimeProperty(callBlockPtr,
                           kTimeProperty));
         tp = GetExtendedProperty(callBlockPtr, kTpProperty);
         epsilon = GetExtendedProperty(callBlockPtr, kEpsilonProperty);
         omegaBar = GetExtendedProperty(callBlockPtr,
                                 kOmegaBarProperty);
         e = GetExtendedProperty(callBlockPtr, keProperty);
         a = GetExtendedProperty(callBlockPtr, kaProperty);

         // If the parameters are zero, return zero.
         if (tp == 0.0) {
            x = 0.0; y = 0.0;
            }
         // Otherwise, calculate the current position.
         else {
            n = fmod((360.0/365.242191)*(d/tp), 360.0);
            m = n+epsilon-omegaBar;
            l = fmod(n+(360.0/pi())*e*degsin(m)+epsilon, 360.0);
            v = l-omegaBar;
            r = kAU*(a*(1.0-e*e))/(1.0+e*degcos(v));
            x = degcos(l)*r;
            y = degsin(l)*r;
            }

         // Prepare to set the type and value of the x and y properties.
         spt.reqFunction = kDETcmdSetPropertyType;
         spt.target = pd->target;
         spb.reqFunction = kDETcmdSetPropertyBinary;
         spb.target = pd->target;
```

(continued on next page)

Listing 5. Calculating kXProperty and kYProperty from kTimeProperty *(continued)*

```
         // Set x's type.
         spt.property = kXProperty;
         spt.newType = kExtendedPropertyType;
         if (CallBackDET(callBlockPtr, (DETCallBackBlock*) &spt)
               == noErr) {
            // Set x's value.
            spb.property = kXProperty;
            spb.newValue = (Ptr) &x;
            spb.newValueSize = sizeof(x);
            CallBackDET(callBlockPtr, (DETCallBackBlock*) &spb);
            }

         // Set y's type.
         spt.property = kYProperty;
         spt.newType = kExtendedPropertyType;
         if (CallBackDET(callBlockPtr, (DETCallBackBlock*) &spt)
               == noErr) {
            // Set y's value.
            spb.property = kYProperty;
            spb.newValue = (Ptr) &y;
            spb.newValueSize = sizeof(y);
            CallBackDET(callBlockPtr, (DETCallBackBlock*) &spb);
            }

         return noErr:
         }
      }
   return kDETDidNotHandle;
}
```

The calculation in Listing 5 happens automatically when the user changes the Time field on the Orbit Calculation information page, or any of the orbital parameters on the Orbit Parameters page. But on the orbits record List information page, we need to do a little work to make each entry in the sublist change when the user changes the Time field on that page. The updateOrbitEntries routine sets the time for each item in the sublist by calling setSublistTimeProperty (Listing 6). The updateOrbitEntries routine iterates through the sublist until it gets an error return, which happens when it tries to reference an entry that doesn't exist — the one just past the end of the list.

Listing 6. updateOrbitEntries and setSublistTimeProperty

```
OSErr updateOrbitEntries(DETCallBlockPtr callBlockPtr)
{
   LongDateTime   ldt;
   long           i;

   // Get the time from the Time field.
   ldt = getTimeProperty(callBlockPtr, kOrbitsTimeProperty);
```

(continued on next page)

Listing 6. updateOrbitEntries and setSublistTimeProperty *(continued)*

```
   // Set the time in each sublist entry.
   for (i = 1;; i++)
      if (setSublistTimeProperty(callBlockPtr, kTimeProperty, i, ldt)
            != noErr)
         break;
   return noErr;
}

OSErr setSublistTimeProperty(DETCallBlockPtr callBlockPtr,
                     short property, long itemNumber, LongDateTime ldt)
{
   DETSetPropertyBinaryBlock  spb;
   OSErr                      retVal;

   spb.reqFunction = kDETcmdSetPropertyBinary;
   spb.target.selector = kDETSublistItem;
   spb.target.aspectName = nil;
   spb.target.itemNumber = itemNumber;
   spb.property = property;
   spb.newValue = (Ptr) &ldt; spb.newValueSize = sizeof(ldt);
   retVal = CallBackDET(callBlockPtr, (DETCallBackBlock*) &spb);
   if (retVal == noErr) {
      DETDirtyPropertyBlock   dp;

      dp.reqFunction = kDETcmdDirtyProperty;
      dp.target.selector = kDETSublistItem;
      dp.target.aspectName = nil;
      dp.target.itemNumber = itemNumber;
      dp.property = kOrbitsTimeProperty;
      retVal = CallBackDET(callBlockPtr, (DETCallBackBlock*) &dp);
      }
   return retVal;
}
```

DRAWING IN A CUSTOM VIEW

The CE calls the orbits record aspect template's code resource with the routine selector kDETcmdCustomViewDraw whenever the part of the Plot information page that contains the custom view needs redrawing. This can happen because the user has just flipped to that page, or because all or part of the page was uncovered — perhaps because another window was moved out from in front of the orbits record window.

```
pascal OSErr Orbits(DETCallBlockPtr callBlockPtr)
{
   if (callBlockPtr->protoCall.target.selector == kDETSelf)
      switch (callBlockPtr->protoCall.reqFunction) {
         ...
         case kDETcmdCustomViewDraw:
            return customViewDraw(callBlockPtr);
         }
   return kDETDidNotHandle;
}
```

Listing 7 shows the calculations we need to perform before we can draw the custom view. First, we determine the view bounds. Given the bounds of the view, the template can then calculate the center of the display, which is where it draws the sun. Finally, the template determines a scaling factor such that the largest orbit will just fill the display. (Actually, with the algorithm we use, it may overflow the display a bit if the orbit is very elliptical.) After these preparations, the template can go through each of the items in the sublist and plot their current positions, names, and (if the Show Orbits checkbox is checked) orbits (Listing 8). Being able to call on the aspect template for attribute type "hrc Planet" to do most of the work greatly simplifies this process.

Listing 7. Preparing to draw the custom view

```
DETGetCustomViewBoundsBlock  gcvb;
OSErr                        retVal:
short                        halfWidth, halfHeight, centerX, centerY;
LongDateTime                 ldt;
long                         i;
extended                     x, y, largestDistance, scaleFactor;

// 1. Determine the view bounds.
// If this isn't for our view, ignore it.
if (callBlockPtr->protoCall.property != kOrbitsCustomViewProperty)
   return kDETDidNotHandle;

// Get the bounds of the view.
gcvb.reqFunction = kDETcmdGetCustomViewBounds;
gcvb.target = callBlockPtr->protoCall.target;
gcvb.property = callBlockPtr->protoCall.property;
retVal = CallBackDET(callBlockPtr, (DETCallBackBlock*) &gcvb);
if (retVal != noErr) return retVal;

// 2. Calculate the center of the display.
halfWidth = (gcvb.bounds.right - gcvb.bounds.left) / 2;
halfHeight = (gcvb.bounds.bottom - gcvb.bounds.top) / 2;
centerX = gcvb.bounds.left + halfWidth;
centerY = gcvb.bounds.top + halfHeight;

// Draw space.
PaintRect(&gcvb.bounds);

// Draw the sun.
ForeColor(whiteColor);
r.top = centerY - 4; r.bottom = centerY + 4;
r.left = centerX - 4; r.right = centerX + 4;
PaintOval(&r);

// 3. Determine the proper scaling factor.
// Get the time.
ldt = getTimeProperty(callBlockPtr, kOrbitsTimeProperty);
```

(continued on next page)

Listing 7. Preparing to draw the custom view *(continued)*

```
// Guess the maximum size.
largestDistance = 0.0;
for (i = 1;; i++) {
   extended newDistance;

   if (getSublistPosition(callBlockPtr, i, ldt, &x, &y) != noErr)
      break;
   newDistance = sqrt(x*x + y*y);
   if (newDistance > largestDistance)
      largestDistance = newDistance;
   }
scaleFactor = (halfHeight - 15) / largestDistance;
```

Listing 8. Drawing the custom view

```
DETGetPropertyRStringBlock   gpr;
long                         showOrbits;
Rect                         r;

// Plot each planet.
showOrbits = getNumberProperty(callBlockPtr, kOrbitsCustomViewProperty);
TextFont(kDETApplicationFont);
TextSize(9);
gpr.reqFunction = kDETcmdGetPropertyRString;
gpr.target.selector = kDETSublistItem;
gpr.target.aspectName = nil;
gpr.property = kDETPrName;

for (i = 1;; i++) {
   // Draw the body.
   if (getSublistPosition(callBlockPtr, i, ldt, &x, &y) != noErr)
      break;
   r.top = centerY - ((short) rint(scaleFactor*y)) - 1;
   r.bottom = r.top + 3;
   r.left = centerX + ((short) rint(scaleFactor*x)) - 1;
   r.right = r.left + 3;
   PaintOval(&r);

   // Draw the name.
   gpr.target.itemNumber = i;
   if ((CallBackDET(callBlockPtr, (DETCallBackBlock*) &gpr) == noErr) &&
         ((*gpr.propertyValue)->dataLength < 256)) {
      HLock((Handle) gpr.propertyValue);
      MoveTo(r.right + 1, r.top < centerY ? r.top - 1 : r.bottom + 10);
      DrawString(((char*) &(*gpr.propertyValue)->dataLength) + 1);
      DisposeHandle((Handle) gpr.propertyValue);
      }
```

(continued on next page)

Listing 8. Drawing the custom view *(continued)*

```
   // Show the orbit (if requested).
   if (showOrbits) {
      LongDateTime   ldtInc;
      extended       orbitInc;
      short          j;

      if (getSublistExtendedProperty(callBlockPtr, i, kTpProperty,
            &orbitInc) != noErr)
         break;
      // orbitInc is calculated such that 36 of them produce a complete
      // one-year orbit.
      orbitInc *= (10.0*24.0*60.0*60.0);
      for (j = 36, ldtInc = ldt + orbitInc; j--; ldtInc += orbitInc) {
         if (getSublistPosition(callBlockPtr, i, ldtInc, &x, &y)
               != noErr)
            break;
         r.left = centerX + ((short) rint(scaleFactor*x));
         r.right = r.left + 1;
         r.top = centerY - ((short) rint(scaleFactor*y));
         r.bottom = r.top + 1;
         PaintRect(&r);
         }
      }
   }

// Return things to normal.
ForeColor(blackColor);
updateOrbitEntries(callBlockPtr);
```

BEYOND PLUTO

AOCE templates are extraordinarily elastic. You can use them to do all of the following:

- show information such as users, addresses, file servers, and planets contained in local and remote catalogs
- easily display text and integer information and, with a little work, display and let the user edit floating-point numbers, times, and virtually any other data type
- display information as text, pictures, or any developer-defined custom view

In this article, we developed a set of templates to hold information about planets, to calculate the positions of the planets, and to plot the positions and orbits of those planets. This issue's CD contains records with entries for all nine known planets. More entries can be added as more planets are discovered in our solar system — or in some other solar system. The planets supplied are divided into two records: inner planets and outer planets. If they're all placed in one record, the scaling of the orbit plots, forced by the size of the outer planet orbits, is such that the inner planets are squished too close together — try it.

Some readers may wonder why we used AOCE templates for our planetary explorations rather than HyperCard, a desk accessory, or a full Macintosh application. Templates provide a lightweight solution, which doesn't require the support of a large application like HyperCard. Indeed, templates run within the Finder itself and leverage off its existing user interface code. Desk accessories are also lightweight, but we wanted permanent storage of the data, for which the AOCE catalog system is perfect.

There's plenty of room for extending these templates. Here are a few ideas:

- Add the z coordinate — see *Practical Astronomy With Your Calculator* for the appropriate formulas.
- Add new types of celestial objects — moons and comets for starters.
- Add spacecraft as a type. Allow the user to set the acceleration vector of the ship.
- Add a page to the orbits record that plots the planet's positions in the sky from a given location on Earth.
- Add options to the existing Plot information page to allow the user to choose one of the planets as the center of the plot, rather than the sun.
- Add a pop-up menu to one of the two information pages for the planet attribute values that selects the color to use when plotting that planet.

We hope you're inspired by this article to write templates for many other uses besides celestial ones. As you can see, AOCE templates provide capabilities well beyond supplying electronic mail addresses or browsing network devices.

RECOMMENDED READING

- *Practical Astronomy With Your Calculator*, by Peter Duffett-Smith (Cambridge University Press, 1988). All the algorithms for the celestial mechanics used in the templates come from this excellent book.
- *Inside Macintosh: AOCE Application Interfaces* (Addison-Wesley, 1994).

Thanks to our technical reviewers Paul Black, Dave Evans, and Bruce Gaya.•

NICK THOMPSON

SOMEWHERE IN QUICKTIME

Supporting Text Tracks in Your Application

Text media tracks were introduced with QuickTime version 1.5 and have been further improved in QuickTime 2.0 with the addition of calls for more powerful searching and a new facility called *burnt text*. The big news is that text tracks are now supported in QuickTime for Windows version 2.0, which makes text tracks a cross-platform solution. If you're developing content-based products that need to be cross-platform, you'll want to take a look at text tracks.

Text tracks give you the ability to embed text in a QuickTime movie, which is particularly useful if you're aiming your product at international markets or at people with hearing impairments (you can subtitle your movie) or if you want to enable your users to perform searches (by including the script of a play or movie in a text track, for instance, you can make it easy for users to find a particular scene by searching for a key piece of dialog). The possibilities for adding value to your QuickTime application and content with text tracks are limited only by your imagination.

This column shows you how to add text track support to your QuickTime application, including support for searching and editing. The sample program QTTextSample on this issue's CD demonstrates what's involved. This small application plays a movie controller–based movie in a window and offers users the ability to search for a particular sequence of characters, or edit the text, in the text track.

Text tracks are handled by the text media handler, which is documented in the "Movie Toolbox" chapter of *Inside Macintosh: QuickTime*. The text media handler's functionality includes the following:

- Searching for text, using FindNextText. With QuickTime 2.0, a new routine for text searching, MovieSearchText, can be used. This call is also available with QuickTime 2.0 for Windows.
- Adding plain or styled text, using AddTextSample or AddTESample. Both of these calls allow you to define additional text properties, such as scrolling and highlighting.

Searching is something that all applications that support text tracks will want to offer the user, while editing text is likely to be something that only a few specialized applications will want to provide. Editing of text can be accomplished with Movie Toolbox routines, as discussed later in this column.

FIRST THINGS FIRST

Your QuickTime application needs to do a few basic things at the outset, as QTTextSample demonstrates. These include checking for the QuickTime version number, growing your heap as large as possible, and checking return codes.

QuickTime version number. A Macintosh application that supports text tracks requires QuickTime version 1.5 or later; a Windows application that supports text tracks requires QuickTime 2.0 for Windows. On Macintosh platforms you can use the Gestalt selector gestaltQuickTime to check that the version number returned in the high-order word is greater than or equal to the required version (0x0150 for QuickTime 1.5; 0x0200 for QuickTime 2.0).

Our sample program bails out if the system isn't running QuickTime version 1.5 or later. If your application uses calls provided by later releases of QuickTime but you also want it to run on earlier versions, you should check for the version number and selectively enable your application's functionality accordingly.

Heap size. As discussed in the Somewhere in QuickTime column in *develop* Issue 13 ("Top 10 QuickTime Tips" by John Wang), you need to ensure that you grow your heap as large as possible in your initialization code by calling the MaxApplZone routine. QuickTime needs to use a lot of handle-based data structures, so you also need to ensure that there are enough master pointers allocated in the base of your

NICK THOMPSON (AppleLink NICKT) found his first job in a surfboard factory, gluing wetsuits together. Then he scammed a job finishing custom surfboards. Somewhere along the way he learned how to program, and he's been riding that wave ever since. Last summer he snagged a job at Apple in Developer Technical Support and moved from London to California. Now he dresses up in a neoprene seal suit on weekends and goes shark fishing in the cold Pacific, armed only with a surfboard. (The glue from his first job must have affected his brain.)•

heap. To do that, you should call MoreMasters a number of times (you can tell how many times by examining your heap while the application is running). If you don't do this, your heap may become fragmented, which in turn may cause certain QuickTime routines to fail due to lack of memory. This is a general Macintosh programming issue that you should be aware of for all applications.

Return codes. Finally, always check those return codes, and handle errors as required. Most QuickTime routines return a status code of type OSErr; if this isn't equal to noErr, you have problems. QuickTime is always trying to tell you something — listen and your life will be more complete!

THE USER INTERFACE

The user interface for the sample application is pretty basic. Figure 1 shows the application window. As you can see, the text for the movie is repeated below the movie so that it can be edited. Buttons offer the user the options of finding specific text or updating the text with editing changes. (The Update Text button is dimmed unless the text in the text box has been modified.)

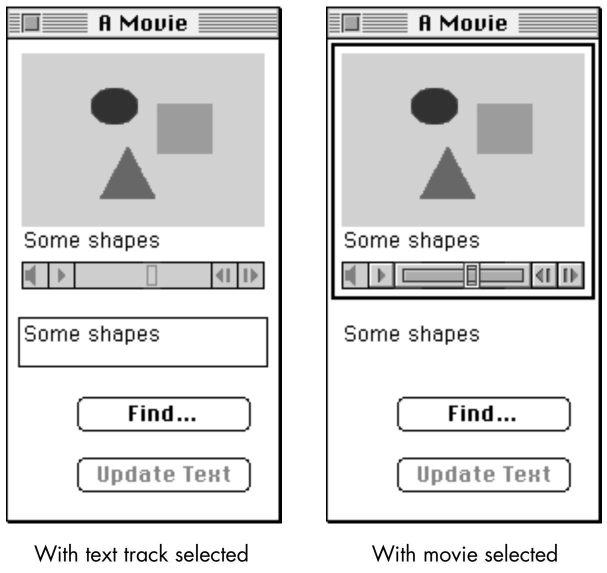


Figure 1. The application window for QTTextSample

Because the user can potentially edit two items — the movie's text track or the movie itself — the application needs to keep track of what the user selected last (either text or the movie) and highlight it in some way. As shown in Figure 1, when the text track is selected in our sample application, the text below the movie has a black box around it; when the movie is selected, the movie frame has a black box around it. When the window is inactive (for example, when you switch applications), the box surrounding the active item is rendered in gray.

TRICKS FOR EASY ACCESS

In the sample code, the movie controller is stored in a record referenced by a handle stored in the window's refCon, along with a few other bits and pieces related to the movie and the text for a window. This gives us easy access to both the movie controller and its associated movie:

```
aDocHdl = (DocHandle)GetWRefCon(theMovieWindow);
aController = (**aDocHdl).myController;
aMovie = MCGetMovie(aController);
```

When we need to locate the first track of a particular type (in our case a text track) in a movie, we can use the following handy utility routine:

```
Track GetFirstTrackOfType (Movie aMovie,
                           OSType trackType)
{
   Track    theTrack = nil;
   OSType   mediaType;
   short    trackCount, index;

   trackCount = GetMovieTrackCount(aMovie);
   for (index=1; index <= trackCount; index++) {
      Track t = GetMovieIndTrack(aMovie, index);
      GetMediaHandlerDescription(GetTrackMedia(t),
                   &mediaType, nil, nil);
      if (mediaType == trackType) {
         theTrack = t;
         break;
      }
   }
   return theTrack;
}
```

A new function, GetMovieIndTrackType, was introduced with QuickTime 2.0 for both Macintosh and Windows. GetMovieIndTrackType provides an easy way to iterate through all tracks in a movie that are either of a given media type or that support a particular media characteristic. Documentation for this, and the other new QuickTime calls, can be found in the QuickTime 2.0 Developer's Kit (which is available from ADPA).

HANDLING TEXT TRACKS

By default, QuickTime displays the text for an enabled text track. We want to be able to exercise more control over the format and display of the text track and to edit the text embedded in the movie. Thus, we need to extract the text from the track and stuff it into a TextEdit record.

Our application needs to be able to access the text for a particular frame as it's displayed. We do this by defining a text-handling procedure (or textProc for short) with the following format:

```
pascal OSErr MyTextProc (Handle thisText,
         Movie thisMovie, short *displayFlag,
         long refCon)
```

The text is passed in the handle. To access the text, we need to determine the length of the text and store it somewhere.

```
// This yields the actual size of the text.
textSize = *(short*)(*thisText);
// This yields a pointer to the text.
textSamplePtr =
            (char*)(*thisText + sizeof(short));
```

The style data for a text track is stored in one of two places. Information about the default text style, together with other items of interest (such as the background color), is stored in a text description handle (see page 2-291 of *Inside Macintosh: QuickTime*). Additional information may be supplied at the end of the handle passed to the textProc, in the form of 'styl' atoms (see page 2-290 of *Inside Macintosh: QuickTime*). Our sample code demonstrates how to access the style information. To get the text description, you need to call GetMediaSample in the textProc, as shown in Listing 1. You need to parse the handle passed into your textProc to see if additional information is supplied; the sample code illustrates how to do this.

We can control whether QuickTime also displays the text, by returning a value in the displayFlag parameter. For example, if we want the default display, we set it in the following way:

```
*displayFlag = txtProcDefaultDisplay;
```

Other flags are available either to suppress output or to ensure that QuickTime always displays the text track, regardless of the settings saved in the movie. Check page 2-364 of *Inside Macintosh: QuickTime* for more details.

In order for the textProc we've defined to get called, we need to tell QuickTime about it. This is easily accomplished with the SetMovieTextHandler routine (shown in Listing 2), which uses the utility routine described earlier to get the first text track.

SEARCHING FOR TEXT

One feature that should be provided in movies that have embedded text is the ability to search for words. Consider a scenario where you're providing an interactive learning experience. The video track of your

Listing 1. Getting the text description

```
theErr = GetMediaSample(aMedia, myData, nil, nil, mediaCurrentTime,
         nil, sampleDescriptionH, nil, nil, nil, nil);
...
if (sampleDescriptionH) {
   scrapHdl = (StScrpHandle)NewHandle(sizeof(StScrpRec));
   if (scrapHdl == nil)
      CheckError(MemError(), "\pCouldn't allocate memory for the style table");
   (**scrapHdl).scrpNStyles = 1;
   (**scrapHdl).scrpStyleTab[0] = (**((TextDescriptionHandle)sampleDescriptionH)).defaultStyle;

   // Delete the previous contents of the TextEditHandle.
   TESetSelect(0, (**myDocTEH).teLength, myDocTEH);
   TEDelete(myDocTEH);

   // Insert the new text.
   TEStylInsert(textPtr, textSize, scrapHdl, myDocTEH);

   DisposeHandle((Handle)scrapHdl);
}
else TESetText(textPtr, textSize, myDocTEH);
```

QuickTime movie contains a play, and the text track contains its script. Students can search for a particular scene just by searching for a few words. If the play were Shakespeare's *Julius Caesar*, for example, searching for "Lend me your ears" would find Marc Antony's speech at Caesar's funeral.

Searching for text in a movie is a straightforward operation using QuickTime's FindNextText routine (which is described on page 2-298 of *Inside Macintosh: QuickTime*). You can control the way this routine works by passing in the following flags:

- findTextWrapAround — wraps the search around at the end or start of the movie and continues searching for the text
- findTextReverseSearch — searches backward
- findTextCaseSensitive — makes the search case sensitive

Under QuickTime 1.5 or 1.6, however, you shouldn't use all three of these flags together in the same call; if you do, a bug will cause a bus error. You can work around this by manually implementing a wrapped search. This was fixed in QuickTime 2.0.

The sample code illustrates the use of FindNextText in the DoSearchForStringInMovieWindow routine. This routine gets the movie controller and its associated movie from the movie window. We pass in the text to search for, the direction to search in, and whether to wrap the search. The sample code shows one way of doing this with a simple dialog.

The new routine MovieSearchText, which was added to the Movie Toolbox in QuickTime 2.0 and in QuickTime 2.0 for Windows, also aids in searching for text in a movie. It can search any track that supports the text characteristic. (For a track to support the text characteristic, it must implement the FindNextText and HiliteTextSample calls as defined in the Movies.h file.) MovieSearchText is defined like this:

```
pascal OSErr MovieSearchText(Movie theMovie,
   Ptr text, long size, long searchFlags,
   Track *searchTrack, TimeValue *searchTime,
   long *searchOffset);
```

In this definition, theMovie is the movie to search, text is a pointer to a block of text that contains the search string, and size is the length of the search string in bytes. The other parameters are as follows:

- searchFlags is a combination of findText flags as defined for media handlers that support text and searchText flags that manage the higher-level searching operation.

Listing 2. The SetMovieTextHandler routine

```
OSErr SetMovieTextHandler (WindowPtr aWindow)
{
   MediaHandler       aMediaHandler;
   MovieController    aController;
   Movie              aMovie;
   Track              aTrack;
   DocumentHandle     aDocHdl;

   aDocHdl = (DocumentHandle)GetWRefCon(aWindow);
   aController = (**aDocHdl).myController;
   if (aController != nil) {
      aMovie = MCGetMovie(aController);

      // If there's a text track in the movie, set the textProc.
      if (aMovie != nil && (aTrack = GetFirstTrackOfType(aMovie, TextMediaType)) != nil) {
         aMediaHandler = GetMediaHandler(GetTrackMedia(aTrack));
         if (aMediaHandler != nil)
            SetTextProc(aMediaHandler, NewTextMediaProc(MyTextProc), (long)aWindow);
      }
   }
   return GetMoviesError();
}
```

- searchTrack is a pointer to the first track to search (or the only track, if searchTextOneTrackOnly is set in searchFlags). If the text is found, searchTrack will be updated to point to the track in which the text was found. If nil is passed in for searchTrack or if it points to nil, the search will start from the first track in the movie.
- searchTime is a pointer to the movie time at which to start the current search. If the text is found, searchTime will be updated to reflect the movie time at which the text was found. If nil is passed in for searchTime or if it points to -1, the current movie time will be used.
- searchOffset is a pointer to the offset within the text sample (as defined by searchTrack and searchTime) in which to start the search. If the text is found, searchOffset will be updated to reflect the offset into the text sample where the text was found. If nil is passed in for this parameter, an offset value of 0 will be used.

If MovieSearchText doesn't succeed in finding the search string because either there were no text tracks in the movie or the text simply wasn't found, an error value is returned.

EDITING TEXT

While the text media handler provides routines to add and delete text track segments, it doesn't provide routines to edit the text contained in a text track. Most applications won't need to edit text, but in case you're interested, this section looks at the Movie Toolbox routines involved.

The DoUpdateText routine in the sample code shows how the user can edit the text contained in a movie's text track. The steps involved in this process are listed below, and the code to accomplish these steps is shown in Listing 3. Note that error checking isn't included in this simplified version of the sample code; you'll find the complete code on this issue's CD.

1. Determine which text track to edit.
2. Determine the segment of the track to be edited. To do this we need to find the start time and duration of the sample we want to edit.

Listing 3. The DoUpdateText routine, simplified version

```
// Step 1:

// Get the text track; remember to check that it's not nil.
aTrack = GetFirstTrackOfType(aMovie, TextMediaType);
...

// Step 2:

// Get the media time of the current sample.
mediaCurrentTime = TrackTimeToMediaTime(currentTime, aTrack);
...
// Get detailed information on start and duration of the current sample (this is used later).
MediaTimeToSampleNum(aMedia, mediaCurrentTime, &mediaSampleIndex, &mediaSampleStartTime,
   &mediaSampleDuration);
...
// Look back and find where this text starts.
theErr = GetTrackNextInterestingTime(aTrack, nextTimeMediaSample | nextTimeEdgeOK, currentTime,
         -kFix1, &interestingTime, nil);

currentTime = interestingTime;

// Determine the duration of this sample.
theErr = GetTrackNextInterestingTime(aTrack, nextTimeMediaSample | nextTimeEdgeOK, currentTime,
         kFix1, &interestingTime, &theDuration);
...
```

(continued on next page)

Listing 3. The DoUpdateText routine, simplified version *(continued)*

```
// Step 3:

// Tell the media that we're about to edit stuff.
theErr = BeginMediaEdits(aMedia);
...
// Delete whatever was there before.
theErr = DeleteTrackSegment(aTrack, interestingTime, theDuration);
...

// Step 4:

// Write out the new data to the media.
theErr = AddTESample(aMediaHandler, aTEH, (RGBColor *)&theTextColor, teFlushDefault, nil,
         dfClipToTextBox, 0, 0, 0, nil, mediaSampleDuration, &sampleTime);
...
// Insert the new media into the track.
theErr = InsertMediaIntoTrack(aTrack, interestingTime, sampleTime, mediaSampleDuration, kFix1);
...
theErr = EndMediaEdits(aMedia);
...
```

3. Delete the existing text using the start time and duration we've determined.

4. Add the text from the TextEdit handle into the media, using the QuickTime routine AddTESample (described on page 2-295 of *Inside Macintosh: QuickTime*). Then call the InsertMediaIntoTrack routine to insert the media we just created into the track.

Like all movie editing operations, editing text will cause the movie to become fragmented. You should ensure that the final production version of your movie is flattened; this will remove any fragmentation introduced by editing.

NEW IN QUICKTIME 2.0: BURNT TEXT

QuickTime 1.6 introduced the capability for applications to apply special effects to the text in text tracks, notably antialiased text and drop shadows. Antialiased text is generally easier to read and looks more attractive than fonts rendered in the normal manner. However, antialiasing text takes time, and the performance penalty that's incurred playing movies with antialiased text tracks limits their usefulness.

QuickTime 2.0 allows applications to prerender text tracks, with a new facility called *burnt text.* Burnt text not only incurs less of a performance penalty than antialiased text but also has the advantage that a font doesn't need to be installed on the target machine in order to be rendered correctly. Applications that want to take advantage of this facility need to write data to the movie file in the form of a number of new atoms; for information on these additional atoms, see the file Text Imaging in QuickTime 2.0, accompanying this column on the CD.

THAT'S ALL THERE IS TO IT

Adding text track support to QuickTime applications really makes sense. With just a few lines of code, you can add a great deal of functionality. Most applications that use text tracks won't need to support editing the text, but it's a good idea to support searching because it provides an easy and powerful way of indexing into a movie containing text tracks.

Take a look at the sample code on the CD. It will help you get started with adding basic searching to your applications and (if required) with more advanced text track features, such as editing. Have fun!

Thanks to Ken Doyle, C. K. Haun, Peter Hoddie, Don Johnson, and John Wang for reviewing this column.•

Make Your Own Sound Components

Sound Manager 3.0, Apple's current audio software release, has an extensible architecture based on sound components that makes it easy for developers to add support for third-party audio hardware and any compressed audio format. Inside Macintosh: Sound gives the theory of how to do this; this article illustrates how to implement the theory, with examples of a sound output component and a sound decompression component.

KIP OLSON

Since its release as a system extension in June 1993, Sound Manager 3.0 has offered developers the possibility of obtaining high-quality digital audio output from the Macintosh using third-party audio hardware and any compressed audio format. Sound Manager 3.0 is now built into all shipping Macintosh computers and is fully integrated into System 7.5. QuickTime 2.0 also takes advantage of new Sound Manager features to provide an even higher level of audio support for multimedia applications. "Somewhere in QuickTime: What's New With Sound Manager 3.0" in *develop* Issue 16 gives a brief sketch of Sound Manager 3.0 and the new vistas it opens.

The Sound Manager architecture and the sound component programming interface are described in detail in *Inside Macintosh: Sound*. What you won't find there is an illustration of how Sound Manager features are implemented in practice. This article fills that gap by offering two examples of sound components. On this issue's CD you'll find NoiseMaker, a sound output component, and MewLaw, a sound decompression component. After briefly describing how Sound Manager 3.0 works, I'll explain how to make a sound output component and a sound decompression component, with reference to these examples.

HOW SOUND MANAGER 3.0 WORKS

Sound Manager 3.0 uses an architecture based on sound components to process audio samples for playback. A sound component is a software module that performs a specific task, usually some kind of audio processing like decompression, rate conversion, sample format conversion, or mixing. Sound components use the Component Manager for registration, loading, and execution.

Figure 1 diagrams a typical sound playback scenario and the sound components that are used. Basically, the sequence is as follows: When an application wants to produce

KIP OLSON received a watch for Christmas last year that records vertical feet skied. To test it out, he had to abandon his post working on QuickTime at Apple while he went on a two-month ski odyssey with his shredder-pal KON. When they weren't arguing about unfair scoring on the Puzzle Page, they managed to rack up 537,460 vertical feet in 340 runs over 28 days, a feat akin to skiing down Mt. Everest 18 times.•

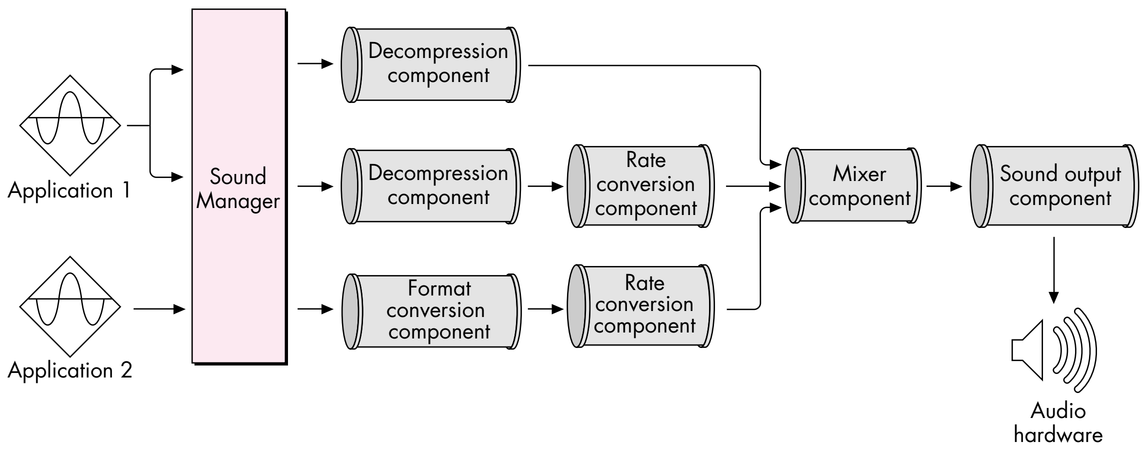


Figure 1. A typical playback scenario

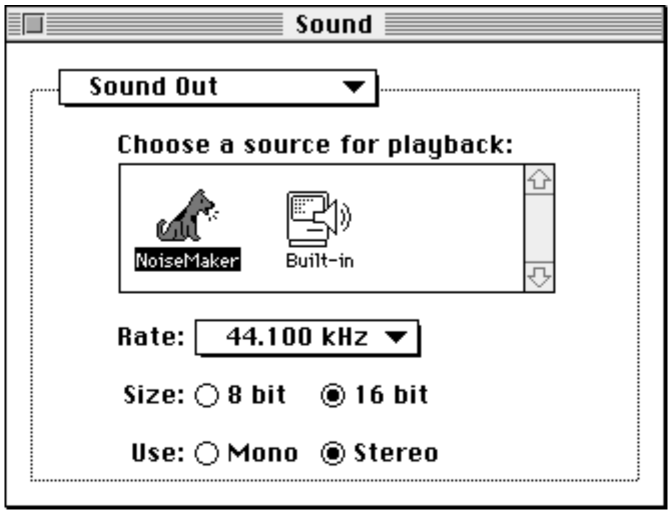
a sound, it calls the Sound Manager to open a sound channel and play the sound. In response the Sound Manager creates a chain of sound components for this channel, where each component performs a specific operation on the audio data. The Sound Manager passes the audio data to the first component in the chain, which can be a decompression component if the data is compressed, a format conversion component if the sample size needs to be changed, or a rate conversion component if the sample rate needs to be adjusted. These components can be applied in series to completely process the audio data into the required format. The mixer component then sums all these audio streams together and provides a single audio source for the sound output component, which uses hardware to convert it to an audible sound.

The sound output component (sometimes called the sound output device component or the output device component) is a software module that identifies, controls, and plays audio samples on some audio hardware device. This device can be a plug-in audio board, a telecommunications pod, or just about anything else that can play sound. Apple provides a sound output component for the built-in audio hardware on every Macintosh except the Macintosh Plus, SE, and Classic machines.

All the sound output components installed on a Macintosh have icons displayed in the Sound control panel that ships with Sound Manager 3.0. The user selects which sound output component to play sounds with by clicking an icon. Figure 2 shows a situation where two sound output components are available: the standard built-in Macintosh sound output component and our example sound output component, NoiseMaker.

When the Sound Manager wants to play a sound, it opens the selected sound output component and sends it commands to start playing the sound. The component is closed when sound playback is completed. The sound output component is responsible for opening a mixer component, which handles the complicated work of allocating chains of sound components and processing the audio data. Our example sound output component, NoiseMaker, shows how this works.

Apple provides sound components for mixing, rate conversion, sample format conversion, and decompression. In addition, sound components can be defined to expand compressed audio from any other format into a format that can be used by other components and played by the hardware. Our example decompression component, MewLaw, illustrates this.


Sound

Sound Out

Choose a source for playback:

NoiseMaker Built-in

Rate: 44.100 kHz

Size: 8 bit [✓ 16 bit]

Use: Mono [✓ Stereo]

Figure 2. The Sound control panel

MAKING A SOUND OUTPUT COMPONENT

Now I'll explain how to make your own sound output component, using NoiseMaker as an example. NoiseMaker doesn't actually control an audio hardware device but rather plays sounds using normal Sound Manager routines. It can be installed on any Macintosh running Sound Manager 3.0 and is meant to be used as a template to manage your own audio hardware.

Here's how NoiseMaker works: The NoiseMaker component is loaded at system startup if the Register method called at that time indicates that the corresponding hardware is available. When the Sound Manager plays a sound using NoiseMaker, it first calls the Open method to open the component, followed by a call to the InitOutputDevice method to have NoiseMaker do any hardware initializations and open a mixer component. It then calls the PlaySourceBuffer method to start the sound playing. When the sound has finished playing, the Sound Manager calls the Close method to have NoiseMaker release the audio hardware and dispose of any memory it created.

I'll refer to NoiseMaker as I describe how to register a sound output component, the structure of a sound output component, the dispatcher, and the methods and interrupt routine that must be implemented.

REGISTRATION AND LOADING

In order for a sound component to be recognized by the Sound Manager, it must be registered with the Component Manager. This is most easily done by creating a file of type 'thng' containing a 'thng' resource that describes your sound component. When you place this file in the Extensions folder, the Component Manager will automatically load the sound component every time the Macintosh starts up. Listing 1 is a 'thng' resource describing our example sound output component.

For full details on the 'thng' resource, see the Component Manager documentation in *Inside Macintosh: More Macintosh Toolbox.*•

The component type and subtype in the 'thng' resource identify the component so that the Sound Manager can find it. The subtype must be unique for each hardware device connected and must contain at least one uppercase character (Apple has dibs on all-lowercase types); it's usually advisable to use an application creator type that

Listing 1. The 'thng' resource for NoiseMaker

```
#define cmpWantsRegisterMessage (1 << 31)
#define componentDoAutoVersion  (1 << 0)
#define kNoiseMakerVersion      0x00010000
#define kNoiseMakerComponentID  128

resource 'thng' (kNoiseMakerComponentID, purgeable) {
   'sdev',                          // sound output component type
   'NOIS',                          // subtype of this component
   'appl',                          // manufacturer
   cmpWantsRegisterMessage, 0,      // component flags
   'proc', kNoiseMakerComponentID,  // code resource
   'STR ', kNoiseMakerComponentID,  // component name
   'STR ', kNoiseMakerComponentID+1,  // component description
   'ICON', kNoiseMakerComponentID,  // component icon
   kNoiseMakerVersion,              // component version
   componentDoAutoVersion,          // registration flags
   0, 0                             // icon family ID, platform
};
```

has been registered with Apple's Developer Support Center to avoid conflicts with other companies. Similarly, the manufacturer name should identify your company and must contain at least one uppercase character; in our example, Apple is the manufacturer so we can get away with using all lowercase letters.

Setting the cmpWantsRegisterMessage bit in the component flags causes the Component Manager to call the sound component with the Register method during the startup process so that the component can determine whether its hardware is available (more on this later). The code resource is the resource type and ID of the code that implements your component. The component description is a string that describes the function of the component to the user. The component name and component icon are used in the Sound control panel, as shown in Figure 2.

The component version and the registration flags are used by the Component Manager during loading to determine whether this component should replace an existing one. If the componentDoAutoVersion bit is set in the registration flags, the Component Manager will install this component only if the version given here is greater than for any other existing component.

The icon family ID and platform fields aren't used by our component.

THE COMPONENT'S STRUCTURE

Sound output components use the standard format required by the Component Manager. The main entry point is a dispatcher, which uses a selector to call the appropriate subroutines (methods). The standard Component Manager methods must be supported, along with a number of additional methods defined for sound output components. The sound output component also contains an interrupt routine that functions as its heartbeat.

Sound output components can create globals that are passed to each method. In addition, there can be one set of global variables that the Component Manager maintains even when the sound output component is closed, which is useful for storing state information. (More about this later when I describe the InitOutputDevice method.)

THE DISPATCHER

The first routine in the component is the dispatcher, which uses a given selector to call the appropriate method. Selectors used by the dispatcher have three ranges, described in Table 1.

Table 1. The ranges of selectors used by the dispatcher

Selector Range	Description
–5 to –1	These are standard Component Manager selectors that all sound components must support. The selectors –6 and –7 are optional and can be supported if you wish. Refer to the Component Manager documentation for more information.
0 to 255	These selectors can't be delegated. If the component receives one of these but doesn't implement it, the component should return the badComponentSelector error.
256 to infinity	These selectors should be delegated. If the component receives one of these but doesn't implement it, the component should use DelegateComponentCall to pass this selector on up the sound component chain.

A number of methods are defined for sound components that don't need to be implemented by every type of sound component. For example, the method SoundComponentAddSource is used only by mixer components and shouldn't be implemented by sound output components. When a component receives a selector that it doesn't support but that can be delegated, it should delegate that selector to its source component and let that component take care of it.

Listing 2 is the dispatcher from our example sound output component. This dispatcher calls an internal utility routine called GetComponentRoutine that returns the address of the routine to call based on the selector. If the sound output component doesn't implement a method, it returns kDelegateComponentCall (-1) as the routine address, which is a flag that this method should be delegated. If the routine returns nil, this is a nondelegatable selector not supported by this component, and an error should be returned. Otherwise, this is a valid method address and the method should be called.

STANDARD COMPONENT MANAGER METHODS

The Component Manager requires that every sound component implement five standard methods, as listed in Table 2. I'll describe each of these methods here; look at the NoiseMaker code to see them in practice.

The Open method is the first method called when a component is opened. This method must create the component globals and store them with the Component Manager, which will then pass these globals to all subsequent methods. While this sounds fairly simple to implement, there are a number of nuances that frequently escape the attention of component writers and wreak havoc; read "Pitfalls of the Open Method" and be forewarned!

The CanDo method is used to determine whether a selector is implemented by this component. In our example code, the CanDo method calls the internal utility routine GetComponentRoutine to determine whether a selector is implemented.

Listing 2. The dispatcher from NoiseMaker

```
#define kDelegateComponentCall     -1

pascal ComponentResult NoiseMaker(ComponentParameters *params,
                                   GlobalsPtr globals)
{
   ComponentRoutine  theRtn;
   ComponentResult   result;

   // Get address of component routine.
   theRtn = GetComponentRoutine(params->what);

   if (theRtn == nil)
      // Selector isn't implemented.
      result = badComponentSelector;
   else if (theRtn == kDelegateComponentCall)
      // Selector should be delegated.
      result = DelegateComponentCall(params, globals->sourceComponent);
   else
      // Call appropriate method.
      result = CallComponentFunctionWithStorage((Handle) globals,
                     params, (ComponentFunctionUPP) theRtn);
   return (result);
}
```

Table 2. The standard Component Manager methods

Method	Selector	Result
Open	kComponentOpenSelect	Opens the component
CanDo	kComponentCanDoSelect	Determines if a given selector is implemented by the component
Version	kComponentVersionSelect	Returns the version of the component
Register	kComponentRegisterSelect	Determines if hardware is installed
Close	kComponentCloseSelect	Closes the component

The Version method returns the version of the component, specified as a fixed-point number. If you're using the auto-version feature of the 'thng' resource, this version must agree with the one specified there.

If the cmpWantsRegisterMessage bit is set in the component flags of the 'thng' resource, the Register method is called by the Component Manager at startup time so that you can see if your hardware is installed and determine whether your component should be loaded. Typically, the Register method should just try to find your hardware. If it's successful, it should return 0; if not, it should return 1, in which case the component won't be registered and won't show up in the Sound control panel. Note that the Open method is always called before the Register method.

The Close method is called to release all memory allocated and all hardware set up by the component. If the Open method fails for some reason and returns an error, the

PITFALLS OF THE OPEN METHOD

Implementing the Open method can be straightforward if you watch out for some common pitfalls. Most important, *do not access or even look for your hardware in the Open method!* There is a separate method (InitOutputDevice) for initializing hardware. The Open method should only allocate instance globals and return. There are two reasons for this:

- If you access hardware in the Open method that isn't available, you might crash or make bad assumptions. The possibility that you might try to access hardware that isn't available is a real one because the Component Manager calls the Open method before it calls the method that checks to see whether your hardware is installed (the Register method).
- Sometimes the component is opened when sound is already playing, to get status information like sample rates and sizes. For instance, the Sound control panel does this to display hardware settings to the user. Resetting or changing hardware in any way during this kind of Open component operation would obviously be disruptive to the sound.

Another tricky interaction with the Component Manager comes into play when you're trying to decide where to create the component globals. Because the sound output component is shared by all applications that are playing sound, the Component Manager will attempt to load the component in the system heap. In this case, your component should create its globals in the system heap as well, so you aren't dependent on any application heaps.

However, if the Component Manager can't find enough space in the system heap to load the component, it will load it in the application heap. In this case, you'll want to create your globals in the application heap as well.

The Component Manager gives you a way to determine where you should create your globals. The call GetComponentInstanceA5 returns the A5 world for the component. If it returns 0, the component was loaded in the system heap and the globals should go there as well; otherwise, the component is in the application heap and the globals should also be created there. The NoiseMaker code shows how this works.

Component Manager calls the Close method. This means the Close method must always check to see if there are valid globals before using or disposing of them. The Close method also must not access the hardware unless the InitOutputDevice method has been called, for the same reasons described in "Pitfalls of the Open Method."

OUTPUT COMPONENT METHODS

In addition to the standard Component Manager methods just described, the sound output component must support the methods listed in Table 3. Again, I'll describe these methods here and leave it up to you to take a look at how they're implemented in the NoiseMaker code.

The InitOutputDevice method is called by the Sound Manager after it opens the component to set up the hardware. This method should initialize the output hardware to a known state and then set the hardware to the default settings stored in

Table 3. Additional methods the output component must support

Method	Selector	Result
InitOutputDevice	kSoundComponentInitOutputDeviceSelect	Sets up the hardware and opens a mixer
GetInfo	kSoundComponentGetInfoSelect	Returns hardware information
SetInfo	kSoundComponentSetInfoSelect	Changes hardware settings
PlaySourceBuffer	kSoundComponentPlaySourceBufferSelect	Begins sound
StartSource	kSoundComponentStartSourceSelect	Begins sound after pause

MANAGING SOUND COMPONENT PREFERENCES

Sound Manager 3.0 provides an easy way to save and recall preferences for your sound component. It maintains a file called Sound Preferences in the Preferences folder and provides two routines to manage this file: SetSoundPreference and GetSoundPreference.

```
pascal OSErr SetSoundPreference(OSType type,
   Str255 name, Handle settings);
```

The SetSoundPreference routine saves a handle of data in the Sound Preferences file tagged with the OSType and name you provide. Typically, the name and type will be the same as your sound component's.

```
pascal OSErr GetSoundPreference(OSType type,
   Str255 name, Handle settings);
```

The GetSoundPreference routine retrieves a handle from the Sound Preferences file based on the OSType and name provided. Typically, the name and type will be the same as your sound component's, and you'll store the handle in the refCon of your component, as shown below.

```
OSErr GetPreferences(ComponentInstance self, Handle prefsHandle)
{
   Handle                 componentName;
   ComponentDescription   componentDesc;
   OSErr                  err;

   componentName = NewHandle(0);   // Space for name
   if (componentName == nil)
      return (MemError());

   // Get name and subtype of sound component.
   err = GetComponentInfo(self, &componentDesc, componentName, nil, nil);
   if (err != noErr)
      return (err);

   // Get preferences for this component from file.
   HLock(componentName);
   err = GetSoundPreference(componentDesc.componentSubType, *componentName, prefsHandle);
   DisposeHandle(componentName);

   // Keep preferences in component's refCon.
   if (err == noErr)
      SetComponentRefcon(self, (long) prefsHandle);
   return (err);
}
```

the component's permanent globals. "Managing Sound Component Preferences" describes an easy way to manage permanent globals.

The InitOutputDevice method must also open a sound mixer component that will be its source for all further sound operations. The mixer does all the hard work of maintaining separate chains of sound components and calling back to the Sound Manager to get more data, while mixing down all the sounds into the single stream of audio data required by your sound output component. Your component simply has to specify the type of sound it needs in the SoundComponentData structure, and the mixer will take care of the rest.

Listing 3 shows how NoiseMaker opens a mixer that will produce a 16-bit, stereo, 44.1 kHz sample stream. For 8-bit data, the format field must be kOffsetBinary, while for 16-bit data, the format must be kTwosComplement. The sampleRate field contains an unsigned fixed-point sampling rate in samples per second. The sampleSize and numChannels fields specify sample size and the mono/stereo setting. The sampleCount field specifies the size of the mixer's output buffer in samples. Every time it's called, the mixer returns a buffer of this size that can then be copied directly to the hardware buffers.

The GetInfo method returns information about the hardware settings. The information returned is based on a four-character selector and is different for each

Listing 3. NoiseMaker's InitOutputDevice method

```
pascal ComponentResult __InitOutputDevice(GlobalsPtr globals,
                                          long actions)
{
#pragma unused (actions)

   ComponentResult   result;
   PreferencesPtr    prefsPtr;

   // Open the mixer and tell it the type of data it should produce.
   // The description includes sample format, sample rate, sample size,
   // number of channels, and the size of your optimal interrupt buffer.
   // If a mixer can't be found that will produce this type of data,
   // an error is returned.

   // Get settings from preferences.
   prefsPtr = *globals->prefsHandle;

   // Set to hardware defaults.
   globals->hwSettings.flags = 0;
   globals->hwSettings.format = (prefsPtr->sampleSize == 8) ?
      kOffsetBinary : kTwosComplement;
   globals->hwSettings.sampleRate = prefsPtr->sampleRate;
   globals->hwSettings.sampleSize = prefsPtr->sampleSize;
   globals->hwSettings.numChannels = prefsPtr->numChannels;
   globals->hwSettings.sampleCount = kInterruptBufferSamples * 2;

   // Open mixer that will produce this format.
   result = OpenMixerSoundComponent(&globals->hwSettings, 0,
               &globals->sourceComponent);
   if (result != noErr)
      return (result);

   // Set the hardware to these settings.
   result = SetupHardware(globals);
   if (result == noErr) {
      // Hardware is ready to go.
      globals->hwInitialized = true;
      // Lock prefs so that we can use them at interrupt time.
      HLock((Handle) globals->prefsHandle);
   }

   return (result);
}
```

selector; see *Inside Macintosh: Sound* for the details. The SetInfo method changes hardware settings. All sound output components must support the GetInfo and SetInfo selectors listed in Table 4. All other selectors must be delegated to the mixer component.

Table 4. GetInfo and SetInfo selectors

Selector	GetInfo Result	SetInfo Result
siSampleSize	Gets current sample size	Sets current sample size
siSampleSizeAvailable	Gets available sample sizes	
siSampleRate	Gets current sample rate	Sets current sample rate
siSampleRateAvailable	Gets available sample rates	
siNumberChannels	Gets current number of channels	Sets current number of channels
siChannelAvailable	Gets available number of channels	
siHardwareVolume	Gets current volume setting	Sets current volume setting

Note: The "...Available" selectors apply only to GetInfo.

The PlaySourceBuffer method is used to begin playing a sound. This method has to first delegate the call to the mixer to start it playing the sound and then make sure the hardware is turned on before returning, by checking the kSourcePaused bit in the actions parameter.

Similarly, the StartSource method is used to begin playing a sound that has previously been paused. In this case, the mixer isn't returning any data for this sound, and the sound output component may have turned off the hardware if no other sounds were playing. Like PlaySourceBuffer, this method has to first delegate the call to the mixer to start it playing the sound again and then make sure the hardware is turned on before returning.

THE INTERRUPT ROUTINE

The interrupt routine is the heartbeat of a sound output component. It's called whenever the audio hardware needs to play another buffer of audio data. It's entirely defined by the component, so it doesn't have a selector or a programming interface. Still, all interrupt routines share the same common features.

First, the interrupt routine should check to see whether any requests have been made to change the hardware settings as a result of the SetInfo method. If so, the hardware should be reset to the new settings. It's important to do this at interrupt time so that the mixer can be synchronized to the new settings without a glitch in the sound.

Second, the interrupt routine must make sure the mixer has provided a load of data. It does this by checking the state of the SoundComponentData structure last returned by the mixer and asking the mixer for more data if needed with the GetSourceData method, described in the next section. Remember, the mixer always returns a buffer of the same size, so it's a simple matter to copy the data to the hardware in fixed sizes. The only exception occurs at the end of the sound, when it might not fill up an entire mixer buffer. In this case, you should copy only as much data as the mixer returned.

Finally, if the mixer has no more data, all sounds have completed playing and the interrupt routine can turn off the hardware.

MAKING A SOUND DECOMPRESSION COMPONENT

Sound components can also be defined to expand compressed audio into a form that can be used by other components and played by the available hardware. The Sound Manager tells a decompression component what format it should use for the data it produces so that when a sound is played it will decompress to the correct sample size, sample format, and number of channels. Our example sound decompression component, MewLaw, decompresses audio data encoded in µ-law format.

When your sound decompression component is installed, the Sound Manager will be able to automatically play sounds compressed in your format, so most applications will be able to play your compressed sounds without any modification. To determine which decompressor to call, the Sound Manager matches the compression format types stored in AIFF files and 'snd ' resources (described in detail in *Inside Macintosh: Sound*) against the subtype you specify in the 'thng' resource.

The structure of a decompression component is similar to that of a sound output component, with a few exceptions and some additional methods, described in the following sections.

EXCEPTIONS TO THE OUTPUT COMPONENT MODEL

Because there is no hardware associated with a decompression component, no initialization is needed beyond the Open call, and only one selector has to be implemented for the GetInfo method. The 'thng' resource should have 'sdec' for a type and your compression OSType for a subtype. The component flags must also be set to describe your format, as documented in *Inside Macintosh: Sound.* The flags should specify the data formats supported by your component in terms of the sample sizes, sample formats, and number of channels that your component can handle.

In the case of a sound decompression component, the GetInfo method returns information about your compression algorithm. Your component needs to support only the siCompressionFactor selector. The infoPtr parameter passed in will point to a CompressionInfo data structure, which must be filled out with information describing your compression algorithm. On entry, the format field of the CompressionInfo record will contain the OSType of the compression format. The fields that you must fill out are as follows:

compressionID	Must be set to fixedCompression
samplesPerPacket	Number of samples in a compressed packet
bytesPerPacket	Number of bytes in a compressed packet
bytesPerSample	Number of bytes in an uncompressed sample

Just like in the output component, the PlaySourceBuffer method is called when the Sound Manager needs to play a new sound. In the case of a sound decompression component, though, your routine should clear out any pointers to the source data that you're keeping. It should *not* reset your compression state variables, as this new buffer is probably a continuation of a previous sound. Be sure to delegate this call to your source component (the component immediately preceding yours in the chain) before you return.

ADDITIONAL METHODS

Besides the methods supported by the sound output component, the sound decompression component must support the methods listed in Table 5. MewLaw demonstrates the use of these methods.

Table 5. Additional methods the decompression component must support

Method	Selector	Result
SetSource	kSoundComponentSetSourceSelect	Specifies who to call for data
SetOutput	kSoundComponentSetOutputSelect	Specifies type of output
GetSourceData	kSoundComponentGetSourceDataSelect	Gets compressed data and decompresses it
StopSource	kSoundComponentStopSourceSelect	Stops sound

The SetSource method is used by the Sound Manager to tell your component who to call to get more data. The source field should be stored for later use; then the SetOutput method should be called on this source to tell it the type of input needed by the decompressor.

The SetOutput method is used by the Sound Manager to tell your component what kind of output to produce. A SoundComponentData record is passed in specifying the output the Sound Manager is requesting. If your component can't produce data using this format, it should set the actual field to the kind of output it can produce and return paramErr. The Sound Manager will then try to convert your output to the format it needs.

The GetSourceData method does the work of getting compressed data from the source, decompressing the data into an internal buffer, and returning this buffer to the component that called it. A couple of subtle features must be supported:

- If the source returns data that's already in the required output format, no decompression needs to take place, and the source data pointer should be passed back.
- If the source buffer is nil, that means that the Sound Manager is requesting that the mixer produce silence for this channel, and your component should just pass the data pointer through.

If neither of these is the case, there is valid compressed source data that your routine should decompress. First you have to decide whether the sound is playing in reverse (that is, backward). Many applications (such as QuickTime applications) need to play sounds backward. The Sound Manager supports this by setting a bit in the flags field of the SoundComponentData record telling the components to play the sound backward. Decompression can't go backward, but the decompression component can decompress chunks of the sound starting at the end of the source buffer and working toward the beginning. (Because the internal buffer is typically fairly small, decompression components often need to decompress source data in chunks.) It passes these decompressed chunks to the format conversion component, which takes care of actually reversing the samples. The example code on the CD shows how this is done.

Once you've determined the right source data to decompress, you simply call your decompression routine, update the information in your SoundComponentData record, and return a pointer to this record.

The StopSource method is called when the Sound Manager needs to stop a sound from playing. Your routine should clear out any pointers to the source data and reset all compression state information. Be sure to delegate this call to your source component before you return.

PLAY ON

With Sound Manager 3.0 so widely available, you'll want to take advantage of the support it offers for third-party audio hardware and for the full range of compressed audio formats by making your own sound components. Look to *Inside Macintosh: Sound* for the technical details of how to proceed, and examine the code for NoiseMaker and MewLaw to see examples of how it's done. Then start making some noises of your own!

REFERENCES

- *Inside Macintosh: More Macintosh Toolbox* (Addison-Wesley, 1993), Chapter 6, "Component Manager."
- *Inside Macintosh: Sound* (Addison-Wesley, 1994), Chapter 5, "Sound Components."
- "Somewhere in QuickTime: What's New With Sound Manager 3.0" by Jim Reekes, *develop* Issue 16.

Thanks to our technical reviewers Bob Aron, Ray Chiang, and Jim Reekes.•

Do you yearn for the adulation of your colleagues?

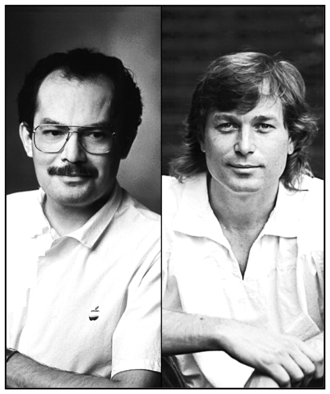
YOUR NAME HERE

Yearn no more: write for *develop*. We're always looking for people who might be interested in submitting an article or a column. If you'd like to spotlight and distribute your code to thousands of developers of Apple products, here's your opportunity.

If you're a lot better at writing code than writing articles, don't worry. An editor will work with you. The result will be something you'll be proud to show your colleagues (and your Mom).

So don't just sit on those great ideas; feel the thrill of seeing them published in *develop*!

For Author's Guidelines, editorial schedule, and information on our incentive program, send a message to DEVELOP on AppleLink, develop@applelink.apple.com on the Internet, or Caroline Rose, Apple Computer, Inc., One Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.



GRAPHICAL TRUFFLES

A Space-Saving PICT Trick

GUILLERMO A. ORTIZ AND DAVE JOHNSON

On the Macintosh, the standard file format for storing images is the PICT format. When pixel maps are stored in PICTs, the color table is always included. If you have a large number of PICTs that all use the same color table, however, it's redundant to have a separate copy of the color table in each PICT. It would be better to strip the color tables out of the PICTs themselves, store only one copy of the color table on disk, and use that color table for all the PICTs. This column describes a simple way to do that.

OF COLOR TABLES AND PICTS

One of the really neat features of the Macintosh and QuickDraw is that the process to store a pixel image in a picture is really simple. The code can be as simple as opening a picture to start PICT recording, doing a CopyBits of a PixMap onto itself, and closing the picture; QuickDraw does the rest.

```
newPict = OpenPicture(&offRect);
CopyBits(srcPix, srcPix, &offRect, &offRect,
   srcCopy, nil);
ClosePicture();
```

Here I use OpenPicture for simplicity, but as pointed out in *Inside Macintosh: Imaging With QuickDraw*, you should really use OpenCPicture, especially if you want to specify a resolution other than 72 dpi.•

This code works great, but there's a catch: every time you do this, the whole PixMap — including its color table — is recorded into the picture. For a PixMap that's 8 bits deep (256 colors), the color table takes a little over 2K. This may not be that big a deal for one or two pictures, but what if you're pressing a CD with *thousands* of pictures that use the same color table? Suddenly all those embedded color tables add up to a significant chunk of space. For pictures delivered on floppy disks, space may be at even more of a premium, and those extra few kilobytes might matter. Also note that if you create PICTs with multiple calls to CopyBits, a color table is stored with *each* PixMap in the picture, making the problem even worse.

LEARNING TO SHARE

More often than not, a series of PICT files will all use the same color table (usually the default color table). In these cases it would be much more economical if we could somehow store only one copy of the color table, and then share it among *all* the pictures. As it turns out, doing that is pretty easy. The solution has two parts: first, we need to be able to create PICTs that don't include a color table, and later we need to be able to draw those same PICTs using the appropriate color table, stored separately from the picture.

JUST SAY NIL

The first part of the solution is easy. In a PixMap, the pmTable field contains a handle to a color table. When recording a picture, QuickDraw simply stores the color table specified by the pmTable field into the PICT. But if pmTable is nil, there's no color table to put in the picture, so its size ends up smaller. Only one extra line of code is needed to do this:

```
// Set PixMap color table handle to nil.
(*srcPix)->pmTable = nil;
```

Naturally, you'll need to save the color table handle beforehand, and afterward restore it before disposing of the PixMap, so that QuickDraw can dispose of all the pieces correctly. But even more important, you'll use the color table handle to create a 'clut' resource that you can save and that can later be used to draw the PICT with its correct colors. The sample program named CLUTLess, included on this issue's CD, contains the complete code to do this.

In reality, QuickDraw adds more than just a nil color table handle to the picture when the code described above is executed. The reason for this is that some applications assume that PixMaps in a picture always have a color table, and they would die horrible deaths if

GUILLERMO A. ORTIZ of Apple's Developer Technical Support group left on sabbatical just after completing the first draft of this column, but before he had a chance to write a bio. He left his office chanting his favorite mind-soothing mantra, "Nothing compensates like cash."•

DAVE JOHNSON watched in astonishment recently as a large hawk or falcon of some kind (he's not much of an ornithologist) devoured a freshly killed dove just outside his living room window in San Francisco. Whoa.•

QuickDraw didn't somehow protect them from themselves. To avoid this problem, QuickDraw stores a color table "stub" into the PICT that it recognizes as such when playing back the picture.

If you simply call DrawPicture to display the "clutless" picture you've made, QuickDraw will see the color table stub and create a color table for the picture on the fly. This color table is a color ramp between the current port's foreground color and its background color — in most cases this means that you get a grayscale image. (If the foreground color and background color are not black and white, respectively, you'll get a "colorized" image.) Figure 1, which appears along with Figure 2 on the inside back cover of this issue of *develop*, shows an example of a once-colorful picture drawn this way.

GET IT TOGETHER

To display the image in its original colors, you need to somehow get the saved color table back into the PICT before drawing. You can do this by replacing the QuickDraw low-level routine that's called when a PixMap opcode is found in a picture. This replacement low-level routine will simply add the color table to the PixMap and then let QuickDraw continue on its merry way, doing all the real work of drawing, but with the correct color table in place. Replacing a low-level routine is a standard technique:

```
void SetNewBitsProc (WindowPtr aWindow,
                     CQDProcs *theProcs)
{
   // Load structure with the standard routines.
   SetStdCProcs(theProcs);
   // Change the one we want to override.
   theProcs->bitsProc = NewQDBitsProc(AddClutProc);
   // Set our window's port to use them.
   aWindow->grafProcs = theProcs;
}
```

Our replacement routine first saves the contents of the PixMap's pmTable field so that it can later restore this value before returning. It then puts a handle to the appropriate color table in the pmTable field and calls StdBits to let QuickDraw do the hard work of drawing. Finally, it restores the saved pmTable value in the PixMap. (Note that to be completely bulletproof you might want this routine to check to see if the image actually needs a color table — that is, check to make sure it's an indexed image, not direct. In our sample code we know this routine won't get called with a direct PixMap.)

```
pascal void AddClutProc (BitMap *src, Rect *srcR,
         Rect *dstR, short mode, RgnHandle msk)
{
   CTabHandle   saveCTH;

   // Save color table handle.
   saveCTH = ((PixMapPtr) src)->pmTable;
   // Put 'clut' resource.
   ((PixMapPtr) src)->pmTable = gSharedClut;
   // Let QuickDraw do the work.
   StdBits(src, srcR, dstR, mode, msk);
   // Restore saved handle and return.
   ((PixMapPtr) src)->pmTable = saveCTH;
}
```

Once the new QDProcs are in place, you can simply call DrawPicture and the correct color table will be inserted on the fly. Figure 2 shows the same PICT as in Figure 1, drawn with the correct color table this time.

BUT WHAT IF . . .

The sample code illustrates the case in which the pictures are being created from PixMaps directly and the color tables are being removed as it happens. But what if the images already exist, and they need to be "postprocessed" in order to strip out the color tables? Using techniques similar to those described above, it's not hard; see this issue's CD for an example.

Another common case might be this: you have a large number of pictures that among them share just a few color tables. To deal with this, you could use a PicComment to store a number in the PICT that will indicate which of the color tables to use for that PICT.

IS IT FOR YOU?

These techniques are probably useful only if you plan to deliver a lot of images that all use the same color table (most likely on a CD). Stripping out the color tables may help you cram more images onto your distribution media or fit a few more images in memory, perhaps allowing for faster redraw (for example, when a sequence of pictures is being used for animation).

Remember that the images thus created will be seen as grayscale (or colorized) images in any application that's not aware it needs to load a separate color table from a resource. But if you find yourself jumping through flaming hoops to try to cram just a few more PICTs on your disk, this may be just the trick you need.

Thanks to Cameron Esfahani, Josh Horwich, and Don Moccia for reviewing this column.•

What are these doing here?

See the Graphical Truffles column on page 63.

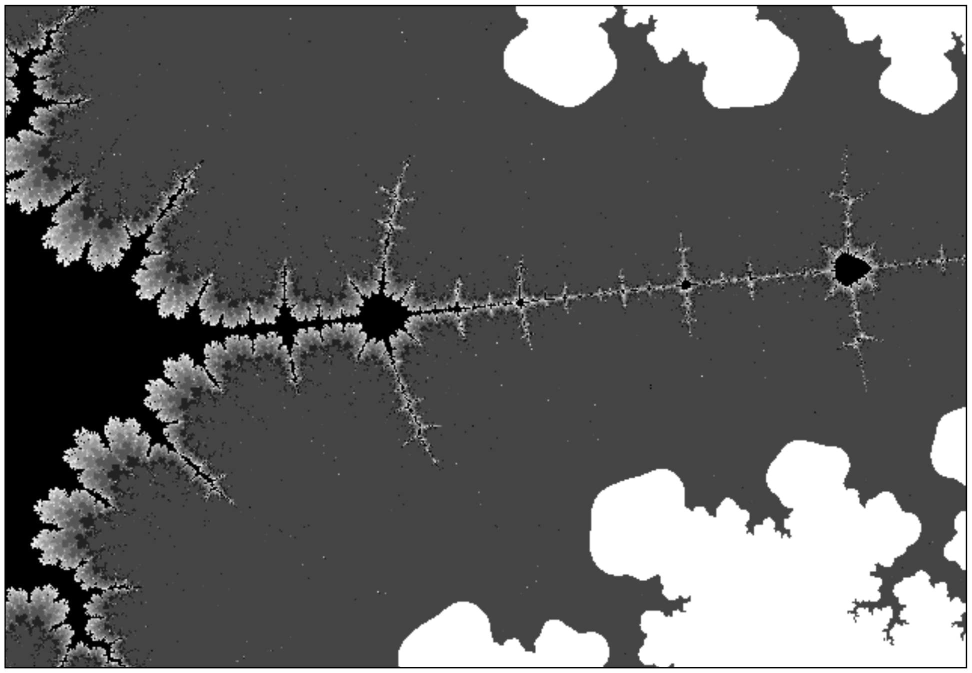


Figure 1. Fractal picture with no color table

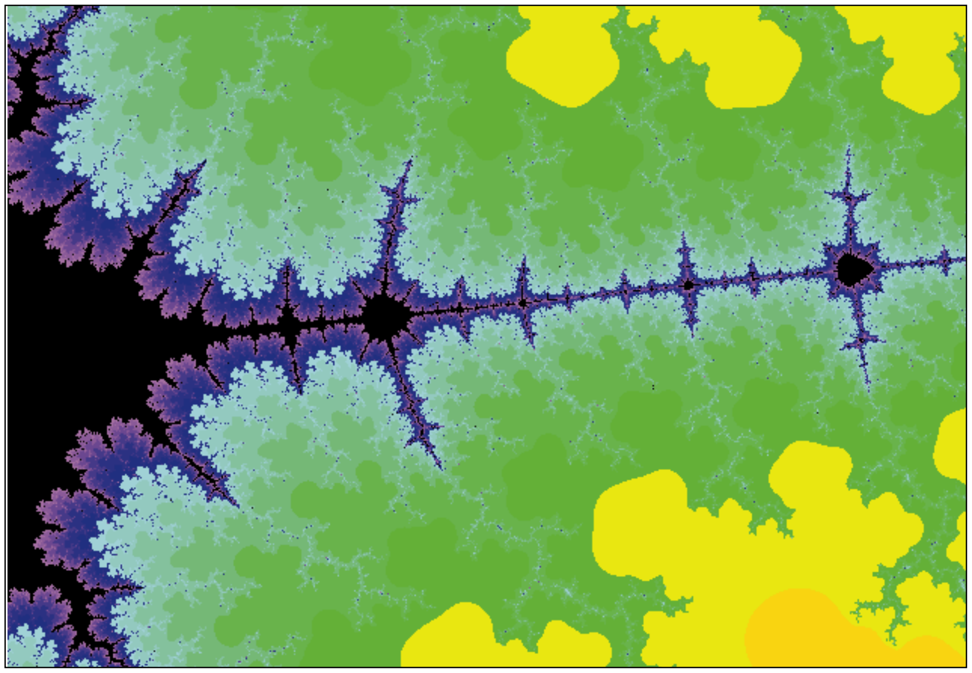


Figure 2. Correct color version of the same fractal

Scripting the Finder From Your Application

The Finder has long been a black box to users and developers — extending the Finder or even examining its state has been nearly impossible. With System 7.5, Apple has shipped a Finder that supports the Object Support Library; this Scriptable Finder opens a new world to developers by allowing applications to interact with the Finder through Apple events.

GREG ANDERSON

The System 7 Finder has always accepted a number of simple events that provide services such as duplicating files, making aliases, and emptying the Trash. But the System 7.0 and 7.1 Finder events are very limited and have strict requirements for the order of parameters and for parameter data types. The Finder that shipped with System 7.5 greatly expands the set of available events: it uses the Object Support Library (OSL) to provide full compatibility with AppleScript, and it provides a new set of events to do things such as examine the Finder's selection, change Finder preferences, and modify file sharing settings.

The term *Scriptable Finder* refers to any Finder that's OSL compliant. In System 7.5, this support is implemented by the Finder Scripting Extension in the Extensions folder; however, future Finders will have scriptability built into their core code base. Developers can count on the presence of the Scriptable Finder in all future versions of system software.

The OSL and the Open Scripting Architecture are critical additions to the Macintosh Toolbox. They mark the end of black-box applications and system software and pave the way for configurable, component-based systems. A Scriptable Finder is only the first step in providing a more unified, open system, but it's an important one.

This article shows you how to generate Finder events from your application. First we'll look at event addressing and the Apple Event Manager, and then we'll see how to specify Finder objects. Finally, the section "Making the Finder Do Tricks" provides a taste of the power and flexibility of the Scriptable Finder, showing some practical uses of this great new capability. On this issue's CD, you'll find the complete code for the article's examples along with sample applications that show how to control the Finder with Apple events. The header file FinderRegistry.h on the CD declares all of the event message IDs, class IDs, and property IDs that the Finder defines.

GREG ANDERSON (AppleLink G.ANDERSON) is currently the Technical Lead of the Finder Team at Apple and was the lead engineer on the Finder Scripting Extension. He's known to engage in a number of activities of questionable sanity, including running straight up hills that are much too steep and much too long, working at Apple for four solid years, making chain mail by hand (with pliers, actually), and putting on armor and hitting people with sticks. Don't worry, he never hits anyone in staff meetings or developer conferences.•

CREATING AND ADDRESSING FINDER EVENTS

Every feature of the Scriptable Finder is accessible via AppleScript. For example, the following script, if typed into the Script Editor and executed, would create a new folder on the desktop:

```
tell application "Finder"
   make folder at desktop
end tell
```

An application doesn't need to compile and execute scripts, however, to use the features of the Scriptable Finder; every command that a script can instruct the Finder to do has a corresponding representation as an Apple event. An application that controls the Finder may bypass AppleScript entirely and send Apple events to the Finder directly. That's the technique we'll use in this article.

There are a number of ways to address an Apple event, but for sending an event to the Finder on the local machine, the simplest and most straightforward technique is to address the event by process serial number (PSN). To determine the Finder's PSN, you walk the list of running processes and search for the Finder's file type and creator, 'FNDR' and 'MACS'.

Listing 1 shows one way to generate an address targeted at the Finder on the local machine. Notice that we've used TDescriptor, which is a C++ wrapper class that corresponds to the Apple Event Manager type AEDesc. (See "C++ Wrappers" for an explanation of wrappers used in this article.)

Should the Finder not be running, looking for processes with the signature 'MACS' will find other user interface shells, such as At Ease, and in some cases you might prefer your application to do that. However, no shells other than the Finder currently support the full Finder Event Suite, so the sample code provided here always requires the process type to be 'FNDR'.

Earlier Finders were not only unaware of the OSL, but they also didn't use the Apple Event Manager. That's right, the System 7.0 and 7.1 Finders never call AEProcessAppleEvent — they interpret and process high-level events in their own special way, without ever informing the Apple Event Manager of what's going on. This means that an application that sends any unrecognized high-level event to the System 7.0 or 7.1 Finder will never get a reply; the application will sit idle in AESend until the event times out (assuming that the send mode was kAEWaitReply).

C++ WRAPPERS

The sample code in this article makes extensive use of C++ wrappers. The file AppleEventUtilities.h, included on this issue's CD, defines the wrapper classes TDescriptor and TAEvent, which correspond to the Apple Event Manager types AEDesc and AppleEvent, respectively. The class TDescriptor contains methods for examining and extracting the contents of AEDesc, AEDescList, or AERecord structures. TAEvent inherits from this class (since an Apple event really is an AERecord) and adds methods for getting and setting attributes and addressing and sending events.

The use of the C++ wrappers makes the code easier to read, but it would be a simple matter to translate the code back into straight C or Pascal functions that call the Apple Event Manager directly. If you do this, don't forget that the C++ constructor of TDescriptor automatically initializes the fields of the AEDesc to a null descriptor (descriptor type = typeNull, data handle = nil). You must do this explicitly in your C or Pascal program, or you could cause problems for the OSL. For example, CreateObjSpecifier will crash if its second parameter is a pointer to an uninitialized AEDesc rather than a valid object specifier or a null descriptor.

Listing 1. Getting the address of the Finder

```
TDescriptor GetAddressOfFinder()
{
   ProcessSerialNumber  psn;
   ProcessInfoRec       theProc;
   TDescriptor          finderAddressDescriptor;

// Initialize the process serial number to specify no process.
   psn.highLongOfPSN = 0;
   psn.lowLongOfPSN = kNoProcess;

// Initialize the fields in the ProcessInfoRec, or we'll have memory
// hits in random locations.
   theProc.processInfoLength = sizeof(ProcessInfoRec);
   theProc.processName = nil;
   theProc.processAppSpec = nil;
   theProc.processLocation = nil;

// Loop through all processes, looking for the Finder.
   while (true)
   {
      FailErr(GetNextProcess(&psn));
      FailErr(GetProcessInformation(&psn, &theProc));
      if ((theProc.processType == 'FNDR') &&
          (theProc.processSignature == 'MACS'))
         break;
   }

   finderAddressDescriptor.MakeProcessSerialNumber(psn);
   return finderAddressDescriptor;
}
```

To determine whether the Finder on the local machine supports the Finder Event Suite, an application can call Gestalt with the selector gestaltFinderAttr and check the gestaltOSLCompliantFinder bit of the result. Before System 7 Pro, gestaltFinderAttr didn't exist, so Gestalt will return the error gestaltUndefSelectorErr on some machines.

Unfortunately, the only way to determine whether the Scriptable Finder is running on a remote machine is to send it an event and wait for it to time out. The best event to send is the Gestalt event from the Finder Event Suite (an event whose class is kAEFinderSuite and whose ID is kAEGestalt) with a direct parameter whose type is typeEnumeration and whose value is gestaltFinderAttr. If the Scriptable Finder is running, the result will have the gestaltOSLCompliantFinder bit set. Under System 7 Pro, the Finder will return an error (event not handled) if the Scriptable Finder isn't running, but the System 7.0 and 7.1 Finders will never return a result.

The Gestalt event can be used to ask for the value of any Gestalt selector. It's easier to call Gestalt directly on the local machine (and more reliable, since the Scriptable Finder might not be running), but some distributed computing applications may want to examine the result of Gestalt selectors on remote machines to determine which are suitable for use as remote hosts.

SPECIFYING FINDER OBJECTS

Most events operate on some Finder object, such as a file, a folder, or a window. These objects are always specified with an Apple event descriptor (AEDesc) placed in the direct object of the event. Some events require specification of more than one object; for example, the Copy event requires parameters for both the objects to be copied and the location to copy them to. In these cases, the direct object of the event is the object being operated on, and other parameters are defined for any other object it requires. The destination of the Copy event goes in the parameter keyAEDestination; other events may define other keywords for parameters they use.

Most scriptable applications require object specification parameters to be in a very specific format called an *object specifier*. The Finder is a little more flexible than that — it will accept a descriptor of type typeAlias (alias record) or typeFSS (FSSpec) in any parameter that requires an object specifier. All the same, understanding object specifiers is critical to sending events to the Finder, because many objects cannot be represented by an alias record or an FSSpec, and therefore must be referenced by object specifier.

Object specifiers are described in "Apple Event Objects and You" in *develop* Issue 10 and in "Better Apple Event Coding Through Objects" in *develop* Issue 12. See also Chapter 6, "Resolving and Creating Object Specifier Records," in *Inside Macintosh: Interapplication Communication.*•

An object specifier is a descriptor whose type is typeObjectSpecifier, but it's actually an Apple event record (AERecord), and can be accessed as such if coerced to type typeAERecord. To build an object specifier, it's most convenient to use the routine CreateObjSpecifier (MakeObjectSpecifier in AppleEventUtilities.cp), which takes four parameters: the desired class of the specified object, the key form, the key data, and the object container.

- The *desired class* indicates the kind of object. Some classes that the Finder recognizes are disks, windows, and folders. The desired class may also be set to typeWildCard to indicate any class of object.
- The *key form* specifies how the object is being addressed; the most common choices are by name and by index.
- The *key data* contains the specification for the object in a format compatible with the key form. For example, if the key form is formName, the key data will contain the name of the object being specified.
- The *object container* is either another object specifier or a null descriptor. Thus, object specifiers have a recursive definition that's always terminated with a null descriptor.

The null descriptor in the object specifier's container is a reference to a special container called the *null container*, which serves as the root container of every scriptable application. In most applications, the items accessible from the null container (called the *elements* of the container) include all the open documents and open windows. The Finder doesn't have any documents that it can open on its own; its null container contains all the open windows, plus all the objects on the desktop, including the mounted disks and the Trash.

You specify *properties*, such as the name of an object or the original item of an alias file, with an object specifier whose desired class is cProperty and whose key form is formProperty. The key data is always of type typeType, and it contains the four-character code identifying the property. The container of the property's object specifier is, as required, an object specifier or a null descriptor.

Usually, the property's container specifies an object — for example, "name of disk 1" would be represented as a property specifier for pName with a container specifier to disk 1. It's also possible to create property specifiers of property specifiers, such as "name of startup disk" (since the term "startup disk" is represented as a property specifier for pStartupDisk). Additionally, there are properties that refer to the Finder itself, or to the Macintosh that the Finder is running on — such as "file sharing," the property that indicates whether file sharing is on or off. These are called properties of the null container, and the container of these property specifiers is always a null descriptor.

Any four-character code that's recognized by FindFolder may be provided as a Finder property that refers to the folder returned by FindFolder. You'll find this useful when moving, copying, or setting properties of special folders.•

The Finder defines a special key form named formAlias. The key data of an object specifier whose key form is formAlias should be an alias record; the desired class should be typeWildCard; and the object container must always be a null descriptor. At first, the existence of formAlias may seem superfluous. The Finder will accept alias records in any object-specifier parameter, and there's no functional difference between a descriptor of type typeAlias and an object specifier of form formAlias. However, formAlias object specifiers are very useful in one regard, and that's to specify properties of files referenced by alias records. As mentioned earlier, the container parameter of an object specifier *must* be another object specifier. If an application already has an alias record, it may use it to build an object specifier of form formAlias for use in other object specifiers. Putting a descriptor of type typeAlias into the container parameter of an object specifier doesn't work, and can even cause the OSL to crash.

MAKING THE FINDER DO TRICKS

A Macintosh running the Scriptable Finder is capable of a variety of tricks that other Finders only dream about. This section demonstrates a number of these features, including events that examine and change the state of the Finder, that notify the Finder of changes, and that manipulate files on disk. For a summary of events the Finder recognizes, see "Overview of Finder Events." This issue's CD includes the complete code for listings in this section.

A complete list of properties the Finder recognizes can be found in the *AppleScript Finder Guide,* the Finder's dictionary resource (viewable from the Script Editor), or the *Finder Event Suite* document on this issue's CD.•

GETTING AND SETTING THE FINDER SELECTION

Determining which files have been selected by the Finder is something developers have been trying to do for a long time. Many ingenious and completely unsanctioned hacks and patches have been devised just to get this simple piece of information. Often, these patches fail to work beyond the Finder that they were designed for, and those that happen to work with multiple Finders may not be compatible with future versions. With the Scriptable Finder, there's no need to patch, hack, or guess which items are selected in the Finder; one simple event will return the answer.

You can obtain the Finder's selection by sending a Get Data event (event class kAECoreSuite, event ID kAEGetDataEvent) to the Finder, and specifying an object specifier for the property pSelection of the null container in the direct object. By

OVERVIEW OF FINDER EVENTS

The Scriptable Finder recognizes most of the events in the Required and Core suites, and defines a few events of its own in the Finder suite.

The Finder recognizes events that:

- count, get, and set data
- test whether an object exists
- open, reveal, select, and close objects
- tell applications to print documents
- move and copy files and folders
- make folders, aliases, and suitcases
- eject and unmount disks
- put away objects in the Trash or on the desktop
- empty the Trash
- query Gestalt
- quit and restart
- shut down and sleep

Of these events, Get Data and Set Data are the most versatile, as they can be used to determine and change a wide variety of properties of the Finder. Some of these properties are:

- the name, label, creation date, modification date, position, custom icon, and comment of an item, as well as its logical and physical size
- the creator, type, and version of a file
- the original item of an alias file
- the partition size of application files
- file sharing and view settings of folders
- the capacity and free space of a disk
- the memory used by a running process

default, the result returned by the Finder will be an object specifier (if one item is selected) or a list of object specifiers (if multiple items are selected). It's also possible to have the results returned as an FSSpec, an alias record, or a pathname by filling in the optional parameter keyAERequestedType of the Apple event. The recognized types are typeFSS, typeAlias, and typeChar (which will return a pathname to the object in the result string). Listing 2 shows how to get the Finder's selection.

Notice that before sending the event, we put in the optional "requested type" parameter. Coercing the data descriptor to a list isn't necessary when sending an event to the Finder, but it *is* required by the OSL specification, so it's a good habit to get into.

You can also change the Finder's selection with a Set Data event (event class kAECoreSuite, event ID kAESetDataEvent): The direct object should again be an object specifier for the property pSelection, and the parameter keyAEData should contain the items to be selected. The key data parameter may contain an object specifier, an FSSpec, an alias record, an empty list (to clear the selection), or a list that contains multiple objects.

As you'll see in the rest of this article, the Get Data and Set Data events are very powerful and can be used for a wide variety of purposes.

GETTING THE FRONTMOST FINDER WINDOW

Although it's not a terribly difficult thing for an ingenious bit of code to obtain a pointer to the Finder's frontmost window, a well-behaved application never peeks at another process's window list. The Scriptable Finder will tell you which windows are open if you ask nicely; once again, the event to use is the Get Data event. The direct object should be an object specifier to the window whose index is 1, as the frontmost window is always the first window in the window list.

Listing 2. Getting the Finder's selection

```
// tell application "Finder"
//    get selection
// end tell
//
// Get the address of the Finder and make a Get Data event.
TAEvent     ae;

TDescriptor target = GetAddressOfFinder();
ae.MakeAppleEvent(kAECoreSuite, kAEGetData, target);
target.Dispose();

// Make an object specifier for "selection" and put it into the
// direct object of the event.
TDescriptor directObjectSpecifier;
TDescriptor keyData;
TDescriptor nullDescriptor;

keyData.MakeDescType(pSelection);
directObjectSpecifier.MakeObjectSpecifier(cProperty, nullDescriptor,
   formPropertyID, keyData, true);
ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
directObjectSpecifier.Dispose();

// Put in the optional "requested type" parameter.
TDescriptor dataDescriptor;

dataDescriptor.MakeDescType(typeAlias);
dataDescriptor.CoerceInPlace(typeAEList);
ae.PutDescriptor(keyAERequestedType, dataDescriptor);
dataDescriptor.Dispose();

// Send the event, extract the reply, and dispose of event and reply.
TAEvent  reply;

ae.Send(&reply, kAEWaitReply);
TDescriptor selectedItems = reply.GetDescriptor(keyAEResult);
reply.Dispose();
ae.Dispose();
```

Note that the event to get the frontmost window always returns an object specifier. It isn't possible to get the Finder to return an FSSpec, alias record, or pathname to a window, because FSSpecs, alias records, and pathnames cannot represent a window — they always point to file system objects. For the event to return an alias to the file system item whose contents are displayed in the frontmost window, its direct object must specify "item of window 1," that is, the item that owns window 1. In most applications, the window's owner would be accessed via the specifier "document of window 1," but because the Finder doesn't have documents, its windows are owned by "items" instead. Listing 3 shows how to get the owner of the frontmost window.

The frontmost Finder window will usually be a folder window, but it could also be an information window, a sharing setup window, or even the About This Macintosh window or Finder Shortcuts window. To limit the window returned to only folder

Listing 3. Getting the owner of the frontmost Finder window

```
// tell application "Finder"
//    get item of window 1
// end tell
//
// Get the address of the Finder and make a Get Data event.
TAEvent ae;

TDescriptor target = GetAddressOfFinder();
ae.MakeAppleEvent(kAECoreSuite, kAEGetData, target);
target.Dispose();

// Make an object specifier for "item of window 1" and put it into the
// direct object of the event. Note that the Apple Event Registry class
// for "item" is cObject.
TDescriptor directObjectSpecifier;
TDescriptor frontWindowSpecifier;
TDescriptor keyData;
TDescriptor nullDescriptor;

keyData.MakeLong(1);
frontWindowSpecifier.MakeObjectSpecifier(cWindow, nullDescriptor,
   formAbsolutePosition, keyData, true);
keyData.MakeDescType(cObject);
directObjectSpecifier.MakeObjectSpecifier(cProperty,
   frontWindowSpecifier, formPropertyID, keyData, true);
ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
directObjectSpecifier.Dispose();

// Specify that we would like the result returned as an alias record
// rather than an object specifier.
TDescriptor dataDescriptor;
dataDescriptor.MakeDescType(typeAlias);
dataDescriptor.CoerceInPlace(typeAEList);
ae.PutDescriptor(keyAERequestedType, dataDescriptor);
dataDescriptor.Dispose();

// Send the event, extract the reply, and dispose of the event and
// reply. frontWindowOwner will contain an object specifier to the
// frontmost window.
TAEvent  reply;

ae.Send(&reply, kAEWaitReply);
TDescriptor frontWindowOwner = reply.GetDescriptor(keyAEResult);
reply.Dispose();
ae.Dispose();
```

windows, change the desired class from cWindow to cContainerWindow. Similarly, the open information windows can be identified by the class cInfoWindow.

The sample application Finder Snapshot on this issue's CD illustrates a very useful reason for requesting the list of open Finder windows. When launched, it records the set of open Finder windows in a document; opening the document results in the same set of windows being opened again and positioned in the same locations that they were in at the time that the document was created. This application provides a simple way to make multiple "working sets" of Finder windows, easily accessible through items in the Apple Menu Items folder, or perhaps via documents sitting on the desktop.

GETTING AND SETTING CUSTOM ICONS

The icon bitmap of a file is available through ordinary file system calls, but there are a couple of different cases to contend with: the icon might be stored in the desktop database, or it could be a custom icon stored in the resource fork of the file. Some files are "special," and only the Finder really knows what their icon bitmap should be. The simplest way to get the exact icon bitmap for a file is to ask the Finder what it is. Once again, Get Data and Set Data are the events to use.

The result of a Get Data event that specifies the icon property of some object is an AERecord that contains the entire icon family for the item's icon. The record contains parameters whose key is the same as the individual resources of an icon family (for example, 'ICN#' and 'icl8'); the data stored in these parameters is identical to the data found in a resource of the same type. A Set Data event takes a record in the same format, or an empty list if the intention is to remove the custom icon.

Listing 4 shows how to remove the custom icon from every item in the selection. Note that the specifier "icon of selection" is equivalent to the more complex specifier "icon of every item of selection."

The sample application Finder Tricks on the CD has a feature that changes the icons of all the items in the frontmost Finder window — each item is given some other item's icon. Other than serving as a useful example of how to send events to the Finder, this sample doesn't have much utility, although it does do an admirable job at messing up the appearance of Finder windows.

An application can change an item's icon by writing the custom icon directly into the appropriate resources in the file and then setting the "custom icon bit" using the file system, instead of sending an event to the Finder — but the change won't take effect right away. The reason for the delay is that the Finder isn't notified when the contents of the disk change, so it must periodically poll the file system to find out whether it needs to redraw any items in its open windows. Polling happens only every now and again, so that the Finder doesn't eat up every spare CPU cycle on the machine when it's just sitting idle in the background.

UPDATING FINDER CONTAINERS

As just mentioned, the Finder sometimes takes a while to notice when the contents of the disk change. If an application writes new information into a folder, it may inform the Finder via an Apple event that the item changed (Listing 5). This event is most useful after an application has created a new file or has changed some visible attribute of an existing file — its type or creator, for example. If an update event isn't sent, the Finder will eventually notice the change and redraw the item; however, there's a several-second delay that's somewhat disconcerting, particularly if the user has just saved a new document to the desktop with the Standard File dialog and expects to see it show up right away.

SETTING UP SHARING

Setting the sharing properties of a folder is a task that many users find confusing. Although scripting this task isn't necessarily any easier, the availability of file sharing

Listing 4. Removing custom icons from the selection

```
// tell application "Finder"
//    set icon of selection to empty
// end tell
TAEvent  ae;

TDescriptor target = GetAddressOfFinder();
ae.MakeAppleEvent(kAECoreSuite, kAESetData, target);
target.Dispose();

// Make an object specifier for "icon of selection" and put it into the
// direct object of the event.
TDescriptor directObjectSpecifier;
TDescriptor selectionSpecifier;
TDescriptor keyData;
TDescriptor nullDescriptor;

keyData.MakeDescType(pSelection);
selectionSpecifier.MakeObjectSpecifier(cProperty, nullDescriptor,
   formPropertyID, keyData, true);
keyData.MakeDescType(pIconBitmap);
directObjectSpecifier.MakeObjectSpecifier(cProperty, selectionSpecifier,
   formPropertyID, keyData, true);
ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
directObjectSpecifier.Dispose();

// Obviously, a Set Data event needs data. In the case of this sample,
// the data we want is "empty," which is represented by an empty list.
TDescriptor emptyList;

emptyList.MakeEmptyList();
ae.PutDescriptor(keyAEData, emptyList);
emptyList.Dispose();

// Send the event and dispose of it once it has been sent.
TAEvent  reply;

ae.Send(&reply, kAENoReply);
ae.Dispose();
```

scriptability makes possible applications that could walk through the process or could provide a more intuitive user interface than the Sharing dialog (commonly referred to in technical circles as “the evil grid of checkboxes”). Listing 6 shows how to enable file sharing on every folder in the current selection.

Unfortunately, not every file sharing feature is scriptable. It’s possible to set the sharing properties of a folder (everything that can be set from the Finder’s Sharing menu item), create a new user or a new group, and rename a user or a group; however, currently it’s not possible to set a user’s password, allow a user to connect to file sharing or program linking, add a user to a group, or remove a user from a group. This capability will be available in some future version of Macintosh system software.

Listing 5. Updating a Finder container

```
// tell application "Finder"
//    update alias "HD:Documents:"
// end tell
void UpdateFinderContainer(FSSpec& changedContainer)
{
   TAEvent  ae;

   TDescriptor target = GetAddressOfFinder();
   ae.MakeAppleEvent(kAEFinderSuite, kAEUpdate, target);
   target.Dispose();

   // Make an object specifier for the FSSpec and put it into the direct
   // object of the event.
   TDescriptor directObjectSpecifier;

   directObjectSpecifier.MakeFSS(changedContainer);
   ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
   directObjectSpecifier.Dispose();

   // Send the event and dispose of it once it has been sent.
   TAEvent  reply;

   ae.Send(&reply, kAENoReply);
   ae.Dispose();
}
```

Listing 6. Sharing every folder in the selection

```
// tell application "Finder"
//    set shared of every folder of selection to true
// end tell
TAEvent  ae;

TDescriptor target = GetAddressOfFinder();
ae.MakeAppleEvent(kAECoreSuite, kAESetData, target);
target.Dispose();

// Make a specifier for "selection."
TDescriptor selectionSpecifier;
TDescriptor keyData;
TDescriptor nullDescriptor;

keyData.MakeDescType(pSelection);
selectionSpecifier.MakeObjectSpecifier(cProperty, nullDescriptor,
   formPropertyID, keyData, true);

// Make a specifier for "every folder of..."
TDescriptor everySpecifier;
```

(continued on next page)

Listing 6. Sharing every folder in the selection *(continued)*

```
keyData.MakeOrdinal(kAEAll);
everySpecifier.MakeObjectSpecifier(cFolder, selectionSpecifier,
   formAbsolutePosition, keyData, true);

// Make a specifier for "shared of..."
TDescriptor directObjectSpecifier;

keyData.MakeDescType(pSharing);
directObjectSpecifier.MakeObjectSpecifier(cProperty, everySpecifier,
   formPropertyID, keyData, true);
ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
directObjectSpecifier.Dispose();

// Set the property to true.
TDescriptor sharedSetting;

sharedSetting.MakeBoolean(true);
ae.PutDescriptor(keyAEData, sharedSetting);
sharedSetting.Dispose();

// Send the event and dispose of it once it has been sent.
TAEvent  reply;

ae.Send(&reply, kAENoReply);
ae.Dispose();
```

MOVING FILES — AND AN UNDOCUMENTED PARAMETER

The Finder also has events that move and copy files from one container to another. Strictly speaking, there's little reason for an application to use these events, since file copying can be done quite acceptably using the file system directly. However, it may take less code to tell the Finder to create a copy than to make the appropriate file system calls and put up a copy progress dialog. The events to use are kAEClone and kAEMove, both of which have the event class of kAECoreSuite.

A new parameter was added to the Move and Copy events of the Scriptable Finder after the *AppleScript Finder Guide* went to press, but before the Finder Scripting Extension shipped with System 7.5. The new parameter allows a Move event to specify the position of every item being moved inside the destination container. This parameter was not originally a part of the Finder Event Suite because a script that needed to position items being moved to another container could always go back and set the position property of the destination items after the move was completed. The new Find File desk accessory included with System 7.5, however, needed to be able to move and position items all in one atomic operation; otherwise, the user would see the items move from an intermediate position to a final position, which would look jerky. The new parameter was added to fill this need; its use is shown in Listing 7.

The code in Listing 7 specifies the position of the item in local coordinates of the destination window. To specify the position in global screen coordinates, use the parameter keyGlobalPositionList instead of keyLocalPositionList.•

Listing 7. Moving a file with the optional position parameter

```
// tell application "Finder"
//    move item "x" to preferences folder positioned at ¬
//       location {10, 10}
// end tell
TAEvent  ae;

TDescriptor target = GetAddressOfFinder();
ae.MakeAppleEvent(kAECoreSuite, kAEMove, &target);
target.Dispose();

// Make a specifier for item "x" and place it in the direct object.
TDescriptor directObjectSpecifier;
TDescriptor keyData;
TDescriptor nullDescriptor;

keyData.MakeString("\px");
directObjectSpecifier.MakeObjectSpecifier(cObject, nullDescriptor,
   formName, keyData, true);
ae.PutDescriptor(keyDirectObject, directObjectSpecifier);
directObjectSpecifier.Dispose();

// Make a specifier for the preferences folder and place it in the
// destination parameter.
TDescriptor preferencesSpecifier;

keyData.MakeDescType(pPreferencesFolder);
preferencesSpecifier.MakeObjectSpecifier(cProperty, nullDescriptor,
   formPropertyID, keyData, true);
ae.PutDescriptor(keyAEInsertHere, preferencesSpecifier);

// Put the point {10, 10} into the local position list.
Point destinationPosition;

destinationPosition.h = 10;
destinationPosition.v = 10;
keyData.MakePoint(destinationPosition);
keyData.CoerceInPlace(typeAEList);
ae.PutDescriptor(keyLocalPositionList, keyData);
keyData.Dispose();

// Send the event and dispose of it once it has been sent.
TAEvent reply;

ae.Send(&reply, kAENoReply);
ae.Dispose();
```

TEACHING THE FINDER NEW TRICKS

From the previous sections it should be clear that the events the Finder recognizes are all very similar, and the code to generate them looks pretty much the same. The event class and message ID may vary, and the contents of the direct object might specify different objects, but there's nothing substantially different between the code that sends an event to open the System Folder and the code that sends an event to get the view setting of the frontmost window.

Be careful, though, when using the constants defined in AERegistry.h; they're intended for use with the old System 7.0 Finder Event Suite. Using the old events (events whose class is kAEFinderEvents, or 'FNDR') has the advantage that they're recognized by earlier System 7 Finders, but in general they should be avoided. The old events are buggy, they don't work with the OSL, and they won't ever be upgraded or changed to support new Finder features. Events in the new event suite (events whose class is kAEFinderSuite, or 'fndr') work better, return meaningful results, and are compatible with the OSL.

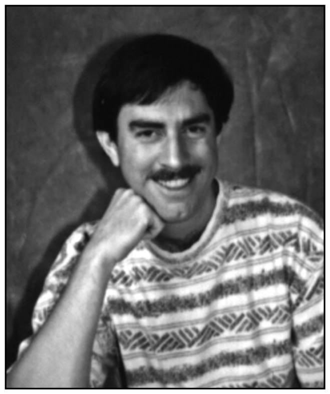
Programmer's documentation of Finder events can be found in the *Apple Event Registry: Standard Suites* and the *Finder Event Suite* document on this issue's CD. The old Finder events are described in the Finder Events chapter of the *Apple Event Registry*. The Scriptable Finder supports the Required and Core suites, as described in the *Apple Event Registry*, and also provides new events that are described in the Finder Event Suite. These documents list the events defined in each suite, the parameters that they take, the classes of objects defined in the suite, and the properties of those objects.

So, the next time you're tempted to disassemble the Finder, poke around in private Finder data structures, or hack your way to Finder properties, remember the new event suite for the Scriptable Finder. Really cool integration with the Finder doesn't have to be painful any more.

RELATED READING

- *Inside Macintosh: Interapplication Communication* (Addison-Wesley, 1993), Chapter 6, "Resolving and Creating Object Specifier Records."
- *AppleScript Finder Guide* (Addison-Wesley, 1994).
- *Apple Event Registry: Standard Suites*, on this issue's CD and available in print from APDA.
- "Apple Event Objects and You" by Richard Clark, *develop* Issue 10.
- "Better Apple Event Coding Through Objects" by Eric M. Berdahl, *develop* Issue 12.

Thanks to our technical reviewers Sue Dumont, Max McFarland, Donald Olson, and Greg Robbins.•



DAVE HERSEY

PRINT HINTS

Improving QuickDraw GX Printer Driver Performance

In this column, we go spelunking in the frost-covered caverns of QuickDraw GX. We'll discover how QuickDraw GX I/O buffering works and how to use that knowledge to squeeze optimal performance from a printer driver, whether PostScript™, raster, or vector. We'll also learn how to find (and avoid) the common bottlenecks.

Suppose you've been working on your first QuickDraw GX printer driver, and the big moment has arrived. Your printer's innards begin to whir and spin, and your heart beats a little faster. Your driver is actually printing! As you see that image being drawn on the page, your breathing quickens, and then . . . the printer stops. You run to your Macintosh to see if your driver has crashed (again), but no, not this time. A few seconds later the printer starts up again. And stops. And starts. This repeats until, several minutes later, the page is finished.

What's going on? Is your printer defective? Maybe. But then again, the problem may lie elsewhere. You probably have a data delivery problem on your hands. For one reason or another, the data isn't getting to the printer fast enough to keep it busy. To understand why, we need to look at what goes on behind the scenes when a driver tells QuickDraw GX to send data to your printer.

Your first reaction might be, "Ah, I need to implement some sort of asynchronous I/O to keep a steady stream of data going to my printer." That's a good thought, but QuickDraw GX already provides asynchronous I/O. Let's look a little deeper.

There are four QuickDraw GX printing messages that are used to implement buffering:

- GXBufferData — sent to move data into an available buffer
- GXWriteData — sent to write data to the printer immediately without buffering it first
- GXDumpBuffer — sent to move a buffer full of data to the printer
- GXFreeBuffer — sent to ensure that a buffer has been processed and is available for new data

How do you get GXBufferData and GXFreeBuffer to work asynchronously, so that the driver's data is sent to the printer as fast as possible? GXBufferData, in its default implementation, already works asynchronously. However, GXFreeBuffer has to work synchronously. Let's look at why.

In the following figures, assume that we have a driver with four buffers, and that at every time interval (a, b, c, and so on) half of a buffer can be filled by the driver. (In reality, the time it takes to fill a buffer will vary as rendering time varies.)

First, let's say that the device can't process the data fast enough to empty out the first buffer before that buffer is needed again. Figure 1 shows what will happen. At the following time intervals shown in Figure 1, here's what takes place:

a. None of the buffers have been used.

b. The first buffer is being written to with GXBufferData.

Time	a	b	c	d	e	f	g	h	i	j
Buffer 1	–	*	○	●	●	●	●	●	●	●
Buffer 2	–	–	–	*	○	●	●	●	●	●
Buffer 3	–	–	–	–	–	*	○	●	●	●
Buffer 4	–	–	–	–	–	–	–	*	○	●

–	Empty buffer
*	Half-full buffer
○	Full buffer, asynchronous write initiated
●	Full buffer, pending I/O completion

Figure 1. Device processes data very slowly

DAVE HERSEY (AppleLink HERSEY) is known to small relatives as "Uncle Mommy." He spent the last three years working with QuickDraw GX and helping developers learn its wily ways. In his spare time, Dave helps his nephews and niece hunt for buried pirate treasure on Joe's Island in Wayne, Maine.•

The best reference for writing QuickDraw GX printer drivers is *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers.*•

c. The first buffer has been filled, so QuickDraw GX sends GXDumpBuffer, which starts an asynchronous write of the data in buffer 1.

d. The first buffer is pending I/O completion, and the driver begins filling the second buffer.

e. The second buffer has been filled, so QuickDraw GX sends GXDumpBuffer for it. It can't be written, however, until the first buffer is finished writing.

f. The first and second buffers are pending I/O completion, and the driver begins filling the third buffer.

g. The third buffer has been filled, so QuickDraw GX sends GXDumpBuffer for it. We're still waiting for the first and second buffers to finish writing.

h. The first through third buffers are pending I/O completion, and the driver begins filling the fourth buffer.

i. The fourth buffer has been filled, so QuickDraw GX sends GXDumpBuffer for it, but it can't write until the first through third buffers finish.

j. All buffers have writes pending. For the first buffer, QuickDraw GX sends GXFreeBuffer, which will wait for I/O to complete on that buffer before returning. GXFreeBuffer must behave synchronously, because its return signifies "This buffer can now be reused."

This is a worst-case scenario from the CPU's point of view. The device's communications pipe can't take the data fast enough to keep up with the buffering. Data buffering is delayed until pending writes are completed. There isn't any alternative — you must free up a buffer in order to have a place to put the new data. Note that it may take several seconds before a buffer is freed. During this delay, the CPU sits idle, although it could be preparing more data.

Figure 2 shows another nonoptimal situation. The buffers are being filled and processed so quickly that at any given time, two — or even three — of the buffers aren't even being used. This is a waste of memory, and also increases the latency between buffers.

Figure 3 shows the ideal situation. This is what you should strive for, although it may not be attainable, depending on your device. In this case, there's always a buffer free. Data is buffered as fast as it's available and (with luck) is sent to the device as fast as the device can service it. In practice, this may be a difficult (if not impossible) scenario to achieve. In a moment, we'll see why. First, let's take a look at the resource that specifies the buffering parameters for a QuickDraw GX printer driver.

Time	a	b	c	d	e	f	g	h	i	j	k	l	m
Buffer 1	–	*	○	●	–	–	–	–	–	*	○	●	–
Buffer 2	–	–	–	*	○	●	–	–	–	–	–	*	○
Buffer 3	–	–	–	–	–	*	○	●	–	–	–	–	–
Buffer 4	–	–	–	–	–	–	–	*	○	●	–	–	–

Figure 2. Device processes data very quickly

Time	a	b	c	d	e	f	g	h	i	j	k	l	m	n
Buffer 1	–	*	○	●	●	●	●	●	–	*	○	●	●	●
Buffer 2	–	–	–	*	○	●	●	●	●	●	–	*	○	●
Buffer 3	–	–	–	–	–	*	○	●	●	●	●	●	–	*
Buffer 4	–	–	–	–	–	–	–	*	○	●	●	●	●	●

Figure 3. Device and buffers are working optimally

THE GXUNIVERSALIOPREFSTYPE RESOURCE

The gxUniversalIOPrefsType ('iobm') resource controls the behavior of the standard buffering and device communication for QuickDraw GX printing. Here's what this resource looks like:

```
type gxUniversalIOPrefsType
{
   longint   standardIO = 0x00000000,
             customIO = 0x00000001;
   longint;     // number of buffers to allocate
   longint;     // size of each buffer
   longint;     // number of I/O requests that
                // can be pending at once
   longint;     // open/close timeout in ticks
   longint;     // read/write timeout in ticks
};
```

The first field in the resource specifies whether you're using QuickDraw GX's standard communications methods (like PAP or serial) or if you're going to provide custom device communications routines (to support SCSI or Centronics printers, for example). If you set this field to customIO, QuickDraw GX won't perform needless memory allocation or initialization to support the standard I/O routines.

The next field indicates the number of buffers you'd like QuickDraw GX to allocate for you (0 indicates none). In low-memory situations, fewer buffers than this number may be created.

Following the number of buffers is the size of each buffer, and then the intimidating "number of I/O requests that can be pending at once" field. A good value for this field is the number of buffers + 3. This represents the possibility of a pending write (or read) on each buffer, as well as a pending status, read, and close connection request.

The rest of the fields in this resource are used to set timeout thresholds.

If a driver doesn't include an 'iobm' resource, the system defaults to two 1K buffers and 10-second timeout values. Because every device is different, it's unlikely that the default options will be ideal for your printer.

DIFFERENCES BETWEEN IMAGING SYSTEMS

PostScript, raster, and vector drivers send differently formatted data to their devices, and this has an effect on how you should set up your buffers.

PostScript drivers. PostScript drivers send text or binary data to their printers, and are generally connected via PAP (Printer Access Protocol). As it turns out, the low-level PAP driver in QuickDraw GX makes sure that no more than (512 * flow quantum) bytes are sent to your device at a time. The flow quantum (normally 8 for LaserWriters) is specified in your gxDeviceCommunicationsType ('comm') resource. So, if your PAP printer uses a flow quantum of 8, a maximum of only 4K will be sent to the printer at a time, even if your buffer size is 8K. This means that a buffer size of (256 * flow quantum) or (512 * flow quantum) usually works well for PAP devices.

Vector drivers. There are some distinct differences between vector drivers and other types of drivers:

- Vector drivers send text commands, but not in the quantity that their PostScript counterparts do. Vector devices tend to understand graphics commands that are only a few characters long but describe graphics that may take several seconds to plot. This is especially true for pen plotters and cutters.
- Because vector devices usually have very basic graphics primitives, operations such as clipping and converting text into polygons are often performed on the Macintosh before the data is sent to the plotter.
- Unlike most PostScript and some raster devices, vector devices rarely wait to start imaging until the entire page is received. It's therefore more efficient to begin the plot as soon as possible, and then send small chunks of data as quickly as possible.

As a result, vector drivers work best when they use several small buffers — for example, buffers of 256 bytes each. This helps keep both the Macintosh and the printer busy.

Raster drivers. Raster drivers send bitmaps to their printers, often with control codes to skip over white areas in the image. The way you set up your buffers for raster drivers can have a dramatic effect on performance — more so than for other types of drivers. The bitmap for a US Letter–sized page on a 24-bit, 300-dpi color device can require 24 megabytes of data. With that much data to process, your code has to be as efficient as possible. For raster drivers, your buffers should be at least the size of one (preferably two) maximum-sized scan lines for your device.

BUFFERING BOTTLENECKS

There are several things that can have an impact on the flow of data to your device. We'll discuss the most common ones here.

The number of buffers specified in your 'iobm' resource. If you used only one buffer in your printer driver, you'd constantly hit the "pending write" lock-out situation described earlier. As soon as you finished filling the buffer, you'd have to wait for it to empty before buffering more data. You should therefore always have at least two buffers.

In an ideal situation, two buffers are all you'd need — one would be always available for buffering while the other is sent to the device. However, you'd need a very fast device to manage this, as we'll soon see. In practice, three or four buffers is a good start for PostScript and raster drivers. For vector drivers, start with eight buffers.

The size of the buffers specified in your 'iobm' resource. As mentioned earlier, this is critical for vector and raster drivers. For vector devices, even moderate-sized buffers (2K) can cause your plotter to stall while data is being buffered, and your Macintosh to stall while that data is being plotted. Remember, a little vector data goes a long way. Start with 256-byte buffers.

If you're writing a raster driver using the default implementation of GXRasterDataIn, make sure that at least one worst-case scan line of data will fit in your buffers. (Keep in mind that your compression scheme might expand the data.) Your buffers must be this large because the gxDontSplitBuffer buffering option is used by the default implementation of GXRasterDataIn. If your buffer isn't big enough to hold an entire scan line, you'll get into an infinite loop as QuickDraw GX keeps rejecting buffers and asking for one that can hold all the data.

There are two reasons for using the gxDontSplitBuffer option:

- It allows for some degree of error recovery. If data is sent to the printer, and the printer is off-line and discards the data, you can just repackage the same scan lines and resend the buffer. If scan lines are split across buffers, it's a little more work to keep track of what to send again.
- Some devices are modal in that they must be set to "graphics mode" before receiving graphics data, and set to other modes before receiving other types of data. Imagine that you split a buffer containing a "start graphics mode" command, followed by some graphics data, followed by an "end graphics mode" command. In between the two GXBufferData calls, the driver might want to query the device with GXWriteData. This could result in chaos or ignored requests because the printer is set to graphics mode and might not accept such queries.

Using the gxDontSplitBuffer option does mean that some portion of each buffer will probably be unfilled. If splitting the data between buffers isn't a problem for your device, override GXRasterDataIn and don't specify gxDontSplitBuffer when you buffer the data.

How big should your buffers be? As mentioned before, probably at least the size of two maximum-sized scan lines. In a minute, we'll see how you can tune your buffer size.

How fast QuickDraw GX can prepare data. It's going to take QuickDraw GX time to prepare the data that it hands your driver. For raster drivers, make sure that your gxRasterPrefsType ('rdip') resource is set up to ask only for the data that you need. Don't make QuickDraw GX spend any more time or pass more data than it needs to.

Time hits from postprocessing. This applies to drivers that do their own halftoning and the like. Can you gain significantly by doing your own halftoning? It's possible, but keep in mind that QuickDraw GX offers a wide range of halftoning and dithering options, and using these methods is likely to take a similar amount of time as just passing your driver the raw data and having it halftone that.

The throughput of the communications pipe. Your device might want to process data faster than the computer sends it due to hardware constraints of, for example, the serial port.

How fast the device can receive data. Similarly, the device itself might be the bottleneck. Keep in mind that the speed the manufacturer claims may not refer to using the printer for printing graphics. Graphic images usually take longer to process than text. The Macintosh (with some minor irrelevant exceptions) prints in graphics mode only, so the claimed rate may not be realistic.

WHICH BOTTLENECKS AFFECT YOU?

Before you can improve the performance of your printer driver, you have to find your bottlenecks. Here are some tests that help determine where your bottlenecks are.

How long does it take QuickDraw GX to prepare data? If you're writing a raster driver, implement a GXRasterDataIn override that does nothing but return noErr. For PostScript or vector drivers, do the same thing in a message override for GXBufferData or GXVectorPackageShape, respectively. If your PostScript or vector driver renders some shapes on its own, you should also override GXPostScriptProcessShape or GXVectorVectorizeShape. In this override, simply forward the message unless you're passed a shape that your driver will render itself. In that case, don't forward the message; just return noErr. This way, your calculations won't include time spent rendering shapes that your driver will be handling completely on its own.

Next, print a typical several-page document and see how many pages per minute you get. If this is slower than the device can print, you might want QuickDraw GX to create an image file of the data before sending it.

Calculating pages per minute is easy. Suppose your "typical" 4-page document takes 72 seconds to render. Then (72 seconds ÷ 4 pages) = 18 seconds per page and (60 seconds ÷ 18 seconds per page) = 3.3 pages per minute.•

To create an image file, override GXCreateImageFile and forward the message along with a combination of the image file options (such as "gxMakeImageFile | gxEntireFile"). There are options for creating image files for each plane, each page, or both. For details, see the QuickDraw GX interface file PrintingMessages.h.

If you use the debugging version of QuickDraw GX, rendering is slower. For accurate benchmarks, use the nondebugging QuickDraw GX extension for timing tests.•

How long is your code taking to postprocess data? Do the same thing as you just did, but include any of your own code (for halftoning, compressing, or whatever) that you normally execute. Compare this to the rate you got from the last test to see how your code is affecting rendering time. Again, an image file might be an option if this is a problem. Also, consider using

CALCULATING DEVICE DATA REQUIREMENTS

A two-page-per-minute, 300-dpi, 4-bit device with a maximum page size of 8 by 10 inches requires (300 x 300 x 4) ÷ 8 bits per byte = 360,000 bits per square inch, or a little under 44K bytes per square inch. The entire page requires (45,000 x 8 x 10) = 3,600,000 bytes per page or about 3.5 megabytes per page. To achieve the device's maximum two-page-per-minute throughput rate, you need to pass twice this amount, or over 7 megabytes of data per minute.

Now, suppose you use compression and also remove beginning-of-line and interline white space to reduce a typical page to, say, 25% of its raw size. Then you're looking at 7 x .25 or about 1.8 megabytes per minute. That's still about 29K bytes per second or about 300 Kbaud to satisfy this device. This can still be a problem if your interface is running at only 9600 baud.

QuickDraw GX's built-in halftoning and dithering instead of your own.

How fast does the device want data? Suppose your device is a two-page-per-minute, 300-dpi, 4-bit device with a maximum page size of 8 by 10 inches. Some quick arithmetic (see "Calculating Device Data Requirements") tells you that you need over 7 megabytes of data per minute, though you can reduce this requirement substantially with compression.

There's another way to determine whether the communications speed is too low: Make your driver roll everything into an image file before sending anything to the printer. Then, print a typical document and see if the printer stays busy once it starts receiving data. If not, the data isn't being sent to your device fast enough. There's not much you can do about this except reduce the amount of data you send or redesign the hardware.

Finally, don't package white space and send it to your device if the device supports skipping it. The GXRasterDataIn message passes a rectangle that indicates where the nonwhite scan lines are in a given band. If you don't skip over the white space on a page, you're wasting time packaging and sending useless data.

Is the buffer usage optimal? Whenever you send GXBufferData, first send GXFreeBuffer. Check to see if GXFreeBuffer returns immediately. If it doesn't, the buffering is being blocked by a pending write. An alternate approach is to implement an override for GXFreeBuffer that subtracts the tick count determined before calling Forward_GXFreeBuffer from the tick count when the call returns. You could record this in a file and look at the information after a print job finishes. Large values indicate that your driver is blocked while waiting for a free buffer.

Try increasing your buffer size or adding more buffers until the lock-out goes away. Note that if your device isn't fast enough, you may never (with reasonable buffer allocation) reach a state in which you never have to wait. Your device (or the communications pipe) might be so slow that the only way to keep a buffer free is to allocate enough buffers to hold the entire page. That's what I would consider unreasonable buffer allocation. However, if you can reach this state of always having a buffer free, back off on the number of buffers or buffer size slightly so that you begin to get occasional lock-outs again. This is your optimal buffer configuration.

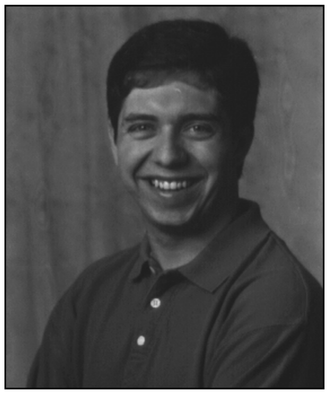
EYES TO THE FRONT, DRIVER

Now that you can optimize your QuickDraw GX buffering and printing, you can avoid the sporadic printing that so many driver writers fall prey to. Your drivers will have the printers humming steadily along, your users will be pleased, and other driver developers will stand in awe of you.

Thanks to our technical reviewers Hugo Ayala, Tom Dowdy, Daniel Lipton, and Harita Patel.•

NetWare Development on PowerPC

Apple and Novell recently announced a PowerPC version of NetWare. Under NetWare, network services such as file servers, print spoolers, and electronic mail dispatchers are written as NetWare loadable modules, or NLMs. By providing a software layer between the NLMs and the hardware they run on, NetWare makes more efficient use of the available hardware, improves portability, and allows programs to run on mixed networks tying together different platforms. This article shows you how to get started with NLM development.



JAMIE OSBORNE

Novell's NetWare has been around for years, and is considered by many to be the networking standard in the DOS/Windows world. NetWare servers have always been able to handle AppleTalk clients, too, but they have not been as prevalent in predominantly Macintosh environments. As a result, most Macintosh developers have never had the need or inclination to learn how to write software for NetWare servers. With Apple's NetWare for PowerPC™, you can now port your existing network products, or create new ones, to run on Apple Workgroup Servers under NetWare.

In this article, we'll take a brief tour of the NetWare environment and what it takes to write software for it. The article is intended primarily for developers of networking software, particularly running on networks using Apple Workgroup Servers or multiple platforms — but you should also find it of interest if you're just curious about NetWare or want to know about the available options for writing networking software.

NetWare is a *network operating system*, a framework for providing network services. Instead of running application programs, NetWare runs *NetWare loadable modules*, or *NLMs*, which typically implement network-based services such as file storage, printing, and electronic mail. An NLM can be loaded either on demand or automatically, and uses the NetWare Operating System (NOS) to allocate memory, communicate with clients, and interact with the underlying hardware. Once loaded, an NLM becomes an integral part of the operating system, with no architectural "middlemen" to slow it down. Figure 1 shows an overview of the NetWare architecture.

JAMIE OSBORNE (AppleLink JWO) In the short space of a year, Jamie Osborne has gone from doing three-dimensional user interfaces to working on exception-handling routines for a PowerPC Memory Manager to programming NetWare NLMs. When he's not sitting in front of a computer (which, according to his fiancée, is "only when he's sleeping, and maybe not even then"), he keeps busy writing television scripts for Paramount to reject. He hasn't quite figured out what he *really* wants to do with his life, but he's reasonably certain it involves long days lounging about in a large house in the hills of New Hampshire.•

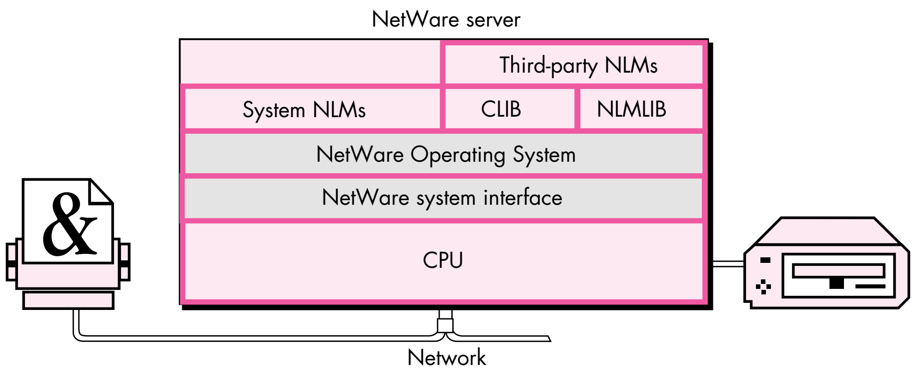


Figure 1. The NetWare architecture

The information in this article is based on Apple's NetWare for PowerPC, an implementation of Novell's NetWare 4.1. Earlier versions of NetWare are still in wide use, but NetWare 4.0 added some new features such as directory services and improved security. NetWare 4.1 is a more robust version of 4.0 and improves still further on these new features. Though earlier versions of NetWare client software can connect to NetWare 4.1 networks, they may not be able to use all the features of the available NLMs. If you want your NLMs to support all versions of NetWare clients, the NOS provides the necessary support libraries.•

WHY NETWARE?

If you're a Macintosh developer, why should you make the leap to NetWare development? To make your product available to the greatest number of users. NetWare is used on more than 60% of all servers in the DOS-compatible world. Here are some of the reasons NetWare is so popular.

EFFICIENT RESOURCE ALLOCATION

In a typical Macintosh network installation, a single server machine provides file storage, printing service, a mail server, and maybe even a scheduling server or other network services. Any one of these services might deliver acceptable performance on its own; but when you try to put them all on the same "box," they must contend for limited resources such as processor time, memory, and disk access. The result is that they all suffer performance degradation: even on a blindingly fast machine, such resource contention can slow all of your network services to a crawl.

NetWare helps alleviate this problem in three ways:

- OS-level resource access. Unlike other operating systems (such as UNIX®), NetWare has no protection scheme to prevent a process from accessing memory outside its domain. Once loaded, NLMs become part of the operating system itself, with unrestricted access to memory and other hardware resources. NLMs run faster without the overhead associated with memory protection. The cost, of course, is that an unprotected NLM can bring the entire system down if it crashes.
- "Lightweight" threads. The NOS is multithreaded, with threads from NLMs existing alongside those belonging to the system itself. Because they operate at the system level, NLM threads carry very little overhead and can be spawned, executed, and switched very quickly.

- Nonpreemptive multitasking. NetWare is a nonpreemptive multitasking system: the burden of deciding when to switch threads is placed on the individual NLMs, rather than on the operating system itself. So long as all NLMs are "good citizens," they can work cooperatively to produce a more efficient system.

CENTRALIZED DIRECTORY SERVICES

One of the biggest headaches for many network administrators is maintaining user and group lists for multiple servers and services. A single user may have an account on two or three file servers, a mail server, and who-knows-what else. Keeping the different accounts for that one person up to date can be a significant chore.

NetWare helps ease this burden by making centralized directory services available to all NLMs. NetWare users log directly into the network itself, not into a particular server. Using interfaces that NetWare provides, NLMs can access the directory services and use them for authentication. Thus a mail server and a file server, for example, can share the same user list instead of each maintaining its own. This centralized approach to directory services benefits everyone. NLM developers don't have to write the code to store, edit, and maintain their own user lists; network administrators only need to maintain a single centralized directory; and users don't have to remember half a dozen different passwords and authenticate themselves every time they move from one network service to another.

PORTABILITY

Networks are growing larger everywhere, as users discover that computers linked together, sharing services over a network, are far more useful than isolated workstations. While this growth presents many opportunities for developers of network services, it also presents the problem of diversity. The days are gone when a company's computers all ran the same operating system. Today, Macintosh computers, DOS-based PCs, and UNIX workstations must all coexist on the same network.

Porting a network service such as a mail server from one hardware platform to another can take a great deal of time and effort. Building your server as an NLM, however, gives it instant portability to any platform for which NetWare is implemented. NetWare's uniform API "virtualizes the hardware," so that NLMs don't have to interact directly with the platform they're running on. Since the interface to NetWare is the same from one platform to another, porting your NLM is a simple matter of recompiling.

THE RIGHT TOOLS FOR THE JOB

By now you should be convinced that NetWare has a lot of advantages to offer. The rest of this article details what it takes to build an NLM that will run on NetWare for PowerPC. Included on this issue's CD is sample code for a simple multithreaded NLM.

CHOOSING A DEVELOPMENT ENVIRONMENT

If you're already developing software for Power Macintosh computers, you probably have most of the tools you need to develop software for NetWare for PowerPC. In the next section, we'll see how to use these tools to do NetWare development in a Macintosh environment; but there are non-Macintosh options as well:

- If you have an IBM RS/6000 computer, you can use the **cset** compiler to develop your NetWare NLMs. (You'll need version 2.1.1 or later of the compiler, which can generate PowerPC code.)

This is the method Apple used to port many of the NLMs that are part of NetWare itself.

- If you have a DOS-compatible machine, you may be able to use Novell's UnixWare in conjunction with the Cygnus C/C++ Compiler for PowerPC (a PowerPC version of **gcc**). You should have at least an 80386 processor for this option, but an 80486 or Pentium is recommended.

If you already have the hardware and software, either of these non-Macintosh options can help you quickly begin producing high-quality code. But if you aren't already using an RS/6000 or a DOS-compatible system, you'll probably do better to go the Macintosh route. Purchasing an RS/6000 can be quite expensive, not to mention the additional cost of setup and maintenance. You can get DOS-compatible machines for much less, but not all of them are compatible with UnixWare. (Any certified UnixWare reseller should be able to help you determine whether UnixWare will run on your hardware.)

This article will focus exclusively on Macintosh development options. However, general information about NetWare's interfaces applies to any development platform you choose.

NETWARE DEVELOPMENT THE MACINTOSH WAY

MPW Pro (available from APDA) includes a PowerPC C/C++ compiler and linker that run as tools under Macintosh Programmer's Workshop (MPW). The compiler, PPCC, produces PowerPC object (.o) files, which you then pass to the PPCLink tool to produce an XCOFF (eXtended Common Object File Format) file.

Ordinarily, the next step would be to pass the XCOFF file, in turn, to the MakePEF tool, which turns it into a PEF (Preferred Executable Format) file ready to run as a Macintosh application. To build an NLM, however, you don't use MakePEF. Instead, you pass your object files to a special-purpose NLM linker that translates them into a finished NLM, using information about imports and exports taken from a *definition file* you supply. (See the next section for more information on the structure and contents of the definition file.) You invoke the NLM linker with an MPW script, NLMLink. As shown in Figure 2, you pass it your definition file along with the usual PowerPC runtime library, PPCRuntime.o, and another library, Prelude.o (provided with the NetWare for PowerPC Software Development Kit), that allows NetWare to load your NLM. The NLMLink script can also accept a list of .o files as arguments, in which case it calls the standard PowerPC linker, PPCLink, for you.

To run and test your NLMs, you'll need the developer's version of NetWare for PowerPC from Novell. You can install it on any Power Macintosh computer.

THE DEFINITION FILE

In addition to your NLM's object (.o) files, you must provide the NLMLink tool with a definition (.def) file. This file, which is usually named *NLMName*.def (where *NLMName* is the name of your NLM), contains information that NLMLink needs in order to turn your linked object file into a finished NLM. Among other things, the definition file includes a list of all routines imported to and exported from your NLM. Listing 1 shows an example definition file, taken from the sample NLM on this issue's CD.

The keywords **description, copyright,** and **version** give the information that will be displayed on the NetWare console when the NLM is loaded.

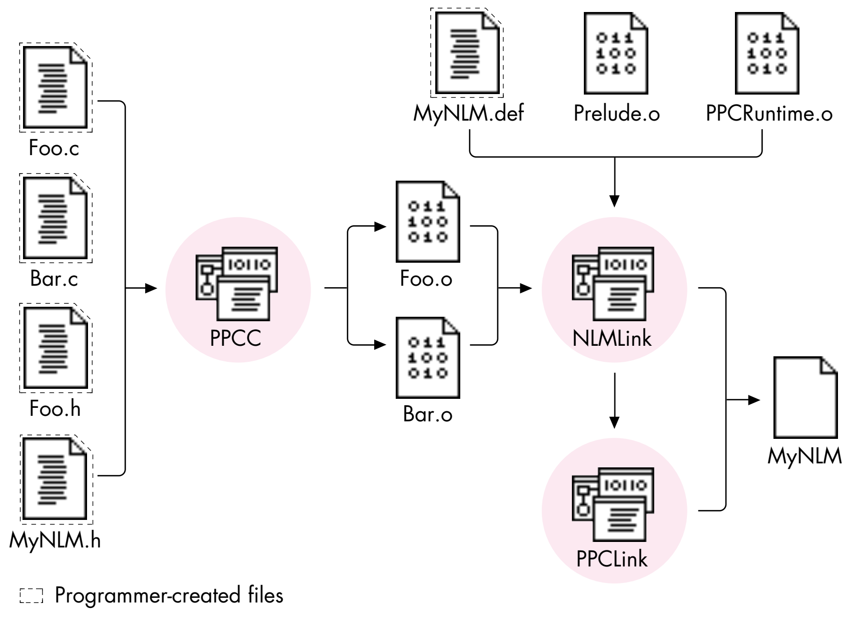


Figure 2. Building an NLM with MPW

The keyword **reentrant** specifies that the NLM can be loaded multiple times on the console, but only one copy of the NLM will reside in memory, with all threads sharing that same copy of the code.

The keywords **input** and **output** tell the linker what file or files to read and what to name the file it produces.

The keywords **start** and **exit** identify routines to execute when the NLM is loaded and unloaded, respectively. The Prelude.o file that you pass to the NLMLink tool defines default start and exit routines, named _Prelude and _Stop, to set up your NLM's threads at load time and clean them up at unload. If your NLM is reentrant, you'll probably supply a start routine of your own to handle reentrant loading; if not, you can just omit the **start** keyword (to use the default start routine _Prelude). If you define your own start routine, make sure it calls _Prelude the first time your NLM is loaded.

The keyword **import** is followed by a list of the routines that your NLM needs to have available at run time. These routines usually come from the NetWare C Interface, but they could be exported by any other NLM running on the server. Any routine your code calls that is not part of your NLM must be listed here, or the NLMLink tool will report an error.

Finally, the keyword **export** is followed by a list of the routines that your NLM makes available for other NLMs on the server to call. You need not export any routines at all; however, if you want to give other NLMs access to any of your routines, you must list them here. (Our sample NLM doesn't actually export any routines, but we've included a fictitious one in the sample definition file, just for illustration.)

These are just some of the keywords you can use in a definition file. The NetWare for PowerPC Software Development Kit documentation describes all of the possible keywords and how to use them.

Listing 1. Example definition file

```
description "AppleTalk Demo NLM"
copyright "Apple Computer, Inc."
version 1, 1, 1
reentrant
input ATDemo.out
output ATDEMO.NLM
start HandleMultipleLoad
exit _Stop
import
   ATAtpClose
   ATAtpGet
   ATAtpOpen
   ATAtpSendRsp
   ATDdpNetinfo
   ATNbpParseEntity
   ATNbpRegister
   ATNbpRemove
   ATZipGetMyZone
   exit
   free
   GetFileServerName
   malloc
   printf
   strcat
   strlen
   strncat
   _StartNLM
   ImportSymbol
   _TerminateNLM
   _SetupArgv
   atexit
   __get_errno_ptr
   BeginThread
   ExitThread
   strcpy
export
   FooBar
```

A BRIEF TOUR OF THE NETWARE INTERFACE

The NetWare C Interface provides more than 1000 functions for interfacing with the NOS. It's a load-time interface, meaning that function calls are resolved at the time an NLM is loaded rather than at link time. Trying to document the entire NetWare interface here would be like trying to summarize all of *Inside Macintosh*. We can, however, look at some highlights. The NLMs that make up the NetWare C Interface include, among others, CLIB (**C LIB**rary), DSAPI (**D**irectory **S**ervices **API**), THREADS, NWSNUT (**N**et**W**are **S**cree**N** **UT**ility), and TLI (**T**ransport **L**ayer **I**nterface). Together, these NLMs offer interfaces to the following NetWare services:

- high- and low-level I/O
- directory services
- file manipulation

- memory management
- threads
- communications protocols
- math functions
- human interface utilities

These services (which are described in more detail below) are actually only a few of those available through the NetWare C Interface. Currently, more than 40 different services (analogous to Macintosh Toolbox managers) are available to developers of NLMs. The NetWare for PowerPC Software Development Kit contains documentation on all of these services, as well as the latest development utilities and sample code.

INPUT/OUTPUT

NetWare provides APIs for both synchronous and asynchronous I/O.

DIRECTORY SERVICES

The Directory Services API provides access to the distributed directory services database on a NetWare 4.1 network. NetWare directory services offer functionality very similar to that of the Apple Open Collaboration Environment (AOCE) — in fact, Apple and Novell are pursuing ways to integrate the two services. Typical calls include NWDSAuthenticate, NWDSCreateObject, and NWDSSearch.

FILE MANIPULATION

NetWare provides several sets of interfaces for file manipulation. Calls such as open, fopen, DFSOpen, and AFPOpenFileFork all open a disk file, but take different parameters and follow different I/O models. You decide which you need, depending on whether you want speed, convenience, or compatibility.

MEMORY MANAGEMENT

NetWare uses the standard C malloc/free model for memory management. NetWare doesn't use a virtual-memory scheme to increase a server's memory capacity beyond that of the available physical RAM, so you must be careful to allocate only as much memory as you actually intend to use. It's also a good idea to include code for handling low-memory conditions in a graceful way.

THREADS

NetWare is a nonpreemptive operating system, meaning that each NLM is responsible for voluntarily giving up control of the processor from time to time, to allow other NLMs to run. Every NLM has at least one thread of execution, known as the *main thread*, created when the NLM is initially loaded. The NLM can then optionally spawn as many additional threads as it needs. At any given time, exactly one thread is in active control of the processor; all others are temporarily suspended. The NOS maintains a *run queue* of such suspended threads awaiting execution.

The NetWare interface provides a set of routines for thread management, including BeginThread, Delay, EnterCritSec, and ThreadSwitch. The "good citizen" routines CYieldIfNeeded, CYieldWithDelay, and CYieldUntilIdle yield control of the processor to give other NLMs their turn at bat. In addition, many other system calls automatically block (suspend the execution of a thread) while waiting for some external occurrence such as the arrival of a packet on the network, so it isn't always necessary to relinquish the processor explicitly.

COMMUNICATIONS

NetWare's AppleTalk interface implements all of the protocols defined in *Inside AppleTalk*, from the Datagram Delivery Protocol (DDP) to the AppleTalk Filing Protocol (AFP). NetWare also provides native interfaces to IPX/SPX and TCP/IP. For greater flexibility, you can use the NetWare Transport Layer Interface (TLI). Because TLI functions are independent of the underlying transport layer, they allow you to use a variety of protocols, such as AppleTalk, IPX/SPX, and TCP/IP, without having to write extra code.

MATH FUNCTIONS

The Math Services API provides common mathematical functions like min, sqrt, and cos. For maximum efficiency, the implementation on PowerPC servers uses the processor's built-in floating-point capabilities whenever possible.

HUMAN INTERFACE UTILITIES

Many NLMs present a human interface on the server machine, allowing an operator or administrator to perform needed server-management tasks. If you want your NLM to have a graphical (as opposed to a command-line) interface, you can use the NetWare NLM User Interface Services to create windows, menus, and dialog boxes on the server console. Although the interfaces aren't as sophisticated as those of the Macintosh Toolbox, they can help you provide a convenient human interface to your NLM.

A SIMPLE NLM

To illustrate how NLMs are written, we'll look at two examples. Both will run on any server running NetWare. You'll find the code for both examples on this issue's CD.

Our first sample NLM is the obligatory "Hello, world" example, which we'll name HELLO.NLM. The code is short enough that we can show it right here:

```
/* HELLO.NLM
   This NLM prints the traditional message "Hello, world." on the server
   console. Type "Load Hello" on the server to see it run.
*/
#include <stdio.h>
main()
   {
      printf("Hello, world.\n");
   }
```

Pretty painless, right? The reason for including this example is to show that writing an NLM doesn't require you to learn an entirely new programming method. Much of the programming you do when writing an NLM is the same as if you were writing an application for Macintosh or UNIX.

A MORE INTERESTING NLM

Now let's look at another sample NLM, named ATDEMO.NLM. When loaded on your server, ATDEMO watches for an incoming AppleTalk connection and provides the client with server statistics on request, such as the number of clients connected, directory listings, and so forth. (The CD also contains a small Macintosh client application that connects to the server and queries ATDEMO for this information. The client application is provided in "fat binary" form so that it can run in native mode on both PowerPC and 680x0 platforms. We won't examine its code in this article, but it's included on the CD in case you're interested.)

ATDEMO demonstrates three important points about writing NLMs:

- how to make calls to the NetWare C Interface
- how to use the AppleTalk protocol in an NLM
- how to create and schedule threads for execution by the NOS

The code for the entire NLM is contained in a single file, ATDemo.c. It begins with a list of #include macros. Some of these, such as stdio.h and stdlib.h, are standard ANSI includes. Others, such as nwenvrn.h, nwthread.h, and nbp.h, are headers for the NOS. Remember that all routines you call from these header files, even the ANSI C routines, are linked at load time and implemented by the NLMs described earlier. *Do not* use the ANSI headers from your MPW CIncludes folder — there may be subtle differences in the header files that could cause debugging nightmares later on.

Like all C programs, an NLM written in C must have a main routine. Macintosh applications typically have a structure based on a main event loop:

```
main()
   {
      /* Do preliminary setup and initialization. */
      ...

      do {
         ...
         WaitNextEvent(...);
         ...
      } while (1);

      /* Do final cleanup and exit. */
      ...
   }
```

The structure of an NLM is a bit different, because the server itself is doing the event processing. Instead of polling the system for events and processing them one at a time in our main thread, we block for action and spawn a separate thread to handle each incoming event. This removes the bottleneck associated with a single point of event processing, with each spawned thread doing its own work and blocking only for its own needs.

The structure of ATDEMO looks like this:

```
main()
   {
      /* Do preliminary setup and initialization. */
      ...

      do {
         ATAtpGet(...);
         ...
         BeginThread(...);
      } while (1);

      /* Do final cleanup and exit. */
      ...
   }
```

At first glance, our NLM seems to spend all its time in a busy **do-while** loop. In fact, this is not the case. ATAtpGet is a NOS function that waits to receive an ATP (AppleTalk Transaction Protocol) packet. Like many other functions in the NetWare C Interface, ATAtpGet is a *blocking function:* it suspends execution of the calling thread while waiting for an ATP packet to arrive, allowing the NOS to run other scheduled threads in the meantime. The thread calling the blocking function gets placed at the end of the run queue; eventually it will work its way back to the front of the queue and resume execution from the point of the suspension. (If our NLM didn't call such a blocking function, we could instead call ThreadSwitchWithDelay at the end of our **do-while** loop, to relinquish control of the processor explicitly and allow other threads to run.)

When the server eventually receives an ATP packet addressed to our NLM, it reactivates our main thread, causing control to return from the ATAtpGet call and resume with the next instruction. Our NLM next calls the NOS function BeginThread to create a new thread to respond to the packet. The actual code issues this call by means of a subsidiary function, SpinNewSession:

```
int SpinNewSession(...)
   {
      int   completionCode;

      completionCode = BeginThread(HandleClientSession, ...);
      return completionCode;
   }
```

BeginThread takes a function pointer as its main parameter, creates a new thread to execute the function, and adds the thread to the NOS's run queue. When the time comes to run this thread, the NOS will call the specified function (in this case, an ATDEMO function named HandleClientSession). Thus, instead of a single thread of execution that handles all communications with all clients, we have a main thread that waits for new clients to initiate communications and spins off a subsidiary thread for each such client connection.

Like our main function, HandleClientSession also has a **do-while** loop that uses ATAtpGet as the main blocking function:

```
void HandleClientSession(...)
   {
      /* Allocate data structures and open a connection. */
      ...

      ATAtpSendRsp(...);

      do {
         ATAtpGet(...);
         quitRequest = HandleRequest(...);
         if (quitRequest) {
            ATAtpClose(...);
            ExitThread();
         }
      } while (1);
   }
```

ATAtpGet will continue getting packets from the client until the client notifies the server to break the connection. Each time HandleClientSession receives a packet, it calls another ATDEMO function, HandleRequest, to examine the content of the message and determine what specific information the client is requesting. When HandleRequest reports that the client has asked to break the connection, HandleClientSession calls the NOS function ATAtpClose to close the connection and then destroys the thread with the NOS function ExitThread.

NETWARE DEVELOPMENT TIPS

Here are some key points to keep in mind when writing an NLM or porting an existing program to NetWare.

- Don't rule out a NetWare version of your software just because it's not "real Macintosh." Apple's future plans for client/server solutions feature NetWare as a core OS.
- Try to make your NLM platform-independent, if possible. If you avoid the platform-specific features of NetWare for PowerPC, you'll be able to port your NLM to NetWare running on other platforms, including DOS-compatible machines, with little effort.
- Remember that your NLM runs on a server that handles many different clients. Try to make it compatible with clients of any type.
- If your NLM is intended to run on multiple platforms or support multiple transport protocols, use the NetWare Transport Layer Interface (TLI) rather than native protocols. This will save you time and effort when porting to another platform.
- Be careful when developing NLMs with a Macintosh development environment. You're writing an application that has no Macintosh Toolbox underneath: use only the headers and libraries that come with the NetWare for PowerPC Software Development Kit.
- Test your NLM carefully and extensively. While in development, an NLM can be run in isolation, where it can't damage anything but itself. Once it's ready for prime time, it will be loaded as part of the NOS; then if it crashes, it may bring the entire server down with it.
- NetWare is a nonpreemptive multitasking operating system. Be sure to design your threads of execution so that they relinquish the processor frequently, to give other NLMs their fair share of processor time.
- Because NetWare servers have no virtual memory, they're limited to the physical memory available in the computer. Don't be a memory pig!
- Be sure to take advantage of the numerous developer programs and information available from Apple and Novell.

Once you've finished developing and testing your NLM, you'll probably want to get it certified by Novell before you release it to the world. Following the guidelines above will help make that process as speedy and painless as possible.

WHAT NEXT?

This article should give you enough basic information to get started with NetWare and NLMs. Novell recommends that anyone interested in setting up a NetWare system hire a trained specialist called a Certified NetWare Engineer, or CNE, to help with the installation. Many NetWare developers also attend training classes to learn how to port their existing software or develop new NLMs. If you don't want to go to the time and expense of a training course, a couple of good books can go a long way to help you learn what you need to know. See the list at the end of this article for some suggestions.

Before embarking on serious development, you'll definitely want to get Novell's NetWare for PowerPC Software Development Kit (which should be available soon if not by the time you read this). In it you'll find a complete set of software and documentation to assist you in developing your own NLMs. For more information, call Novell at 1-800-NETWARE (1-800-638-9273). For information from Apple on NetWare developer programs, call Apple's Developer Support Center at (408)974-4897 or send a message to AppleLink DEVSUPPORT.

In today's networking environment, more and more products, especially client/server programs, are being written for multiple platforms. NetWare gives you a convenient way to develop portable network software without having to rewrite all your code separately for each new platform. Whether you're porting an existing network application or writing a new one, you can benefit greatly by doing it in NetWare.

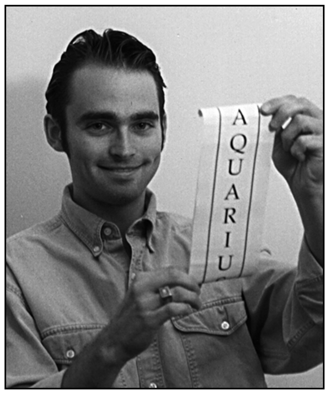
RECOMMENDED READING

Here are some suggestions for further reading on the subject of NetWare and NLMs:

- *Novell's Guide to NetWare 4.0 NLM Programming* (Novell Press, 1993).
- "Writing Your Own NetWare Loadable Modules," *PC Magazine* Volume 12 Issue 20 (November 23, 1993), pages 355–364.
- "Concurrent Programming With the Thread Manager" by Eric Anderson and Brad Post, *develop* Issue 17. Information on multithreaded programming.

Thanks to our technical reviewers Rob Hawley, Bob Heldt, Rich Kubota, Michael McDaniel, and Clara McKenzie.•

If you found this article interesting and would like to see more in-depth NetWare articles in the future, send a note to AppleLink NWDEV.•



DAVE EVANS

BALANCE OF POWER

PowerPC Branch Prediction

The PowerPC processors try to predict which way your code will execute. This sounds surprisingly astrological for a digital machine, but it becomes very useful for a pipelined processor and will often speed up your code. In this column I'll go over why and how this works, focusing especially on the new PowerPC 604 processor prediction techniques, and I'll answer the question "Can a Power Macintosh really tell the future?"

PSYCHIC DECISIONS

Typically about one-seventh of the instructions in your code are branches, either to call subroutines or to make logical decisions in your program. The PowerPC processor would ordinarily tend to stall at branches, since it tries to work on more than one instruction at a time and it's not always sure which code it should execute after a branch. It could either take the branch or fall through, and often the processor won't know which until a couple of cycles later.

So the PowerPC processors allow for speculative execution, meaning they'll guess at the most probable direction the branch will go and then will issue those instructions. But the processor doesn't let the instructions commit until it's sure the guess was correct. Usually it guesses right, and a few instructions are already completed when the branch is decided. If the guess was wrong, it throws out those results and starts over with the correct code.

This predictive skill helps keep the processor executing successfully without stalls, and better prediction techniques will yield better overall performance. The new PowerPC 604 processor improves on earlier prediction techniques; I'll discuss all of them in detail below.

But first, a relevant astrological note: The "birthday" of the 601 makes it a Taurus, whereas the 603 is a Libra. The 604 chip had a birthday in April, so it's an Aries.

TAURUS AND LIBRA ARE COMPATIBLE

The PowerPC 601 and 603 processors use basically the same techniques to predict branches. For simple unconditional branches, for example, they both process and remove the branch early in the instruction issue stage. This operation, called *branch folding*, keeps the instruction stream moving without having to wait for the branch to be processed. The branch is handled early, and the new instructions are fetched from the cache immediately.

For conditional branches, both processors first try to handle the branch early in the instruction issue stage. If the condition being tested has already been evaluated, the branch is folded out of the instruction stream. But if the condition being tested is still in the pipeline, the processor must guess at the branch direction.

Prediction of guessed branches are based on two things: the direction of the branch and a software "hint" bit. If the direction is negative — backward in your code — the branch is taken (because loops often iterate a few times backward before falling through, and this heuristic is more often true). All other branches fall through by default. The hint bit is a way for the compiler to reverse this heuristic: if the bit is set, the prediction will be reversed.

As far as I know there are no compilers that allow you to specify the hint bit in your code, although this could be a valuable feature. Also, profilers or similar tools could take statistics on your code flow and then set the bits for you from trial runs of your software.

THE TEMPERAMENT OF ARIES

The PowerPC 604 has much better branch prediction, which means better performance. Because branch statements most often repeat themselves, it remembers recent branch results to make its predictions:

- It has a cache of the last 64 branches that it has taken, and any time it sees one of these branches again it will immediately predict to the same branch destination. This technique, called *dynamic branch prediction*, is used on the Pentium and other processors with great results.
- It keeps a history of all other branches and predicts based on the recent directions that branch took.

DAVE EVANS (Aquarius, January 20–February 18) Look for opportunities to communicate. You are bound to have fun. Love is in the air; don't work too much or you'll miss it. Apple continues to hold promise for you. Compatible with Sagittarius.•

The cache technique has the advantage of being very fast. When the 604 fetches an instruction, it also sends the instruction's address to the branch cache. If the instruction is a recently executed branch, the cache will return the address of where the branch last went. This is immediately used to fetch the next instruction. Because this all occurs during the fetch of the branch instruction itself, there's no delay in fetching the first predicted instruction.

For conditional branches that aren't in the branch cache, the 604 keeps a history of recent times it saw that instruction. It keeps 512 such histories, each two bits wide, to remember whether the branch was taken during the last few executions. The processor hashes the instruction address to keep the branch histories distinct, and hash collisions are very rare.

Each history is set to one of four states: strongly taken, taken, not taken, and strongly not taken. The current state determines the branch prediction as taken or not taken. After the branch commits, the state is updated. Each update adjusts the state one step toward strongly taken or strongly not taken. The two intermediate steps are a hedge so that it will usually take two mistakes before a prediction changes. Because branches tend to repeat, this algorithm generally results in the following prediction:

- If the branch was taken during the last two executions, the 604 predicts it will again be taken.
- If the branch wasn't taken during both of the last two executions, the 604 predicts it again won't be taken.

Also with the 604, branches on the count register base their prediction on the current count value. This will usually predict loops correctly and yield good performance, since loops count down for a number of iterations before the final iteration causes an incorrect prediction.

But these techniques also come with a tradeoff: the 604 has an extra pipeline stage to dispatch instructions. This means instructions take longer to get through the pipe, and mispredicted branches are more expensive.

ARIES RISING

The 604 is the fastest PowerPC processor yet, and I can't talk about it here without also going into why it's such a fast engine. Besides its advanced branch prediction hardware, it has significantly more integer and floating-point hardware, which yields improved overall performance. Given that it's produced with a more advanced silicon process than the original 601, it's clocked above 80 MHz and offers blazingly fast computation for your code.

As a backbone for the chip, the instruction issuing and control logic allow the 604 to issue up to four instructions per clock, compared to the 601's and 603's effective three. As mentioned above, however, its pipeline has one extra decode stage and branches are issued and handled in their own branch unit. To help it speculatively execute more instructions than the other chips, it also comes with twice the number of "rename" registers than the 603. Twelve extra general-purpose and eight extra floating-point registers are available to hold speculatively produced results until a branch commits. The 604 is also the first PowerPC processor that can speculatively execute two branches at once. This, combined with advanced branch prediction, should keep the processor screaming even through complex code flow.

What most people will notice, however, is the additional integer math performance on the 604. At any one time, the 604 can have two add-subtract instructions and one multiply-divide instruction completing in a cycle. IBM says that it therefore has three integer units, but the multiply-divide hardware is also used for logical and bit manipulation operations. The bottom line is much better integer performance than the Power Macintosh 8100/80. As an example of this, the following code should execute nearly twice as fast on the 604 than on the 601:

```
do {
   unsigned long   datapoint;
   datapoint = *(dataarray + datasize);
   if (datapoint > kThreshold) {
      if (datapoint > kMaxLong - accumulate)
         MyOverflowError();
      accumulate += datapoint;
      samplecount += 1;
      }
   } while (datasize--);
```

Looking at this code, we see a few integer operations that will be dual-issued on the 604. As long as the datapoint values aren't too erratic, the 604 will better predict the first **if** statement's branch: it will assume that the current datapoint is on the same side of the threshold as on the previous iteration, which in fact is where it will tend to be. And the second **if** statement, which checks for an overflow, will (barring an exception) get predicted correctly out of the loop. The 601 or 603 may predict it incorrectly. So even though one integer unit will be busy doing the math, the overflow checking will effectively occur without stalling the pipeline.

The floating-point hardware was also supercharged. On the 601 and 603 processors, a single-precision floating-point instruction can issue and complete each cycle, but double-precision numbers take twice as long. The 604 allows one full double-precision multiply-add instruction to be issued and one to complete each cycle. The chip is twice as fast as the 601 and 603 for these double-precision calculations.

THE FUTURE IS IN THE STARS

So can Power Macintosh tell your future? It certainly tries to with the prediction techniques described above, and in doing so yields better performance. With the simple methods of the 601 and 603, or the dynamic prediction of the 604, your Power Macintosh will speculatively execute your code with seemingly psychic results.

What about the future of the Power Macintosh? The PowerPC architecture allows excellent growth. When I saw the specifications for the first processor, the 601, I was very impressed. It's an excellent design and it has proven to be a potent engine for the Macintosh. When I saw the specifications for the follow-on chips, however, I was really blown away. The 603 and 604 offer incredible performance for the price, and prove that the PowerPC architecture scales well both into low-cost/low-energy solutions and to the cutting edge in performance. And the technology applied to the 604 can be expanded in future chips, adding more execution units and advanced caches at higher clock speeds. The latest IBM POWER2 processors can issue two load/store, two logic/branch, two floating-point, and two integer instructions per cycle. These processors point to the future of PowerPC performance.

So without any additional tuning on your part, PowerPC will continue to improve your performance in the future. I also feel compelled to reiterate this advice from my previous columns: tune your critical code. Tuning often trades performance for code readability and maintainability, so carefully choose which code to tune and use code profilers (and the stars?) to guide your way.

Thanks to Phil Sohn, Peter Steinauer, and Eric Traut for reviewing this column.•

Macintosh Q & A

Q *I'm having problems getting PICTs to display with the colors I want. I'm converting GIF files to PICTs by drawing the GIF into an offscreen GWorld. I'm using the Palette Manager to set up the colors, but there's no way to associate a palette with an offscreen PixMap. After I'm done drawing the GIF to a PixMap, I open the picture with the offscreen PixMap as the current port and use CopyBits to copy the PixMap onto itself, creating the picture. The problem is that if I use srcCopy, the colors are incorrect in the PICT when opened with TeachText (and other applications). But if I use ditherCopy the colors are saved correctly. I can use srcCopy if I do a CopyBits to/from a "color" window with the window's palette changed to my color palette. Is there a way to assign a palette to use for OpenPicture and still use CopyBits from an offscreen bitmap with srcCopy?*

A You can associate a palette with a GWorld, but it won't solve your problem: since a GWorld never becomes "active," the associated device's colors are never changed to match the palette. The solution is to use a custom color table with the GWorld. And you can easily use Palette Manager routines to convert your palette to a color table.

Use the Palette2CTab routine to perform the conversion. Palette2CTab takes a PaletteHandle and a CTabHandle and copies all the colors from the palette into the color table, resizing the color table as necessary. If the palette handle is nil, no change takes place.

Now you have a color table that you can associate with your GWorld. You can pass it to NewGWorld when you create your GWorld initially; the fourth parameter is a handle to a color table. You need to explicitly set the depth in this call for best results. (If you pass nil for the depth, the color table parameter will be ignored and the depth of the GWorld will be set to match the deepest device that intersects the GWorld's boundary rectangle.) The other possibility is to associate the color table with an existing GWorld using UpdateGWorld.

Q *I'm having a bit of a problem with DiffRgn. I start out with a "wide open" rectangular region (-32767, -32767, 32767, 32767) and then use DiffRgn to subtract a group of smaller rectangles from it. When I'm done, the bounding box of the region isn't what it should be. Any idea what's happening?*

A What you need to do is create your clipping region so that it's not quite wide open (bottom and right coordinates of 32766 will work). If you do this, all your DiffRgn calculations will work fine.

While this isn't explicitly documented anywhere, it does seem to be a quirk in the way regions work. Due to the internal storage format of regions, the number 0x7FFF (32767) causes problems if it appears as a point inside a region. 0x7FFF is used as a flag in the internal region data structure to signify a "barrier." When this flag is used as a data point in a nonrectangular region, region parsing becomes completely screwed up.

QuickDraw tries to catch the creation of regions that will be poorly formed and turn them into properly formed (but slightly incorrect) regions, but it isn't 100% successful.

Q *I'm erasing my windows with a color other than white, by setting the window's background color and calling EraseRect. But in cases where the Window Manager gets there first (window ordering changes or a window's size gets larger) I still get flicker,* *because the Window Manager erases with the wContent color from the window's color table (white by default) and not the port's background color. Is there a friendly, clean way to avoid that flicker? (I notice that in System 7.5, background colors are implemented with EraseRect, just as I'm doing it. Are you simply assuming the flash will be minimal?)*

A One of the Window Manager's functions is to ensure that the content region of a window is opaque when it needs to be: that's why the Window Manager "pre-erases" the window when the content region grows, before your application gets a chance to. As you point out, if your application is then erasing large areas of the window to a different color, you'll get a noticeable flicker in those parts of the content region that needed to be opaque. This is an unfortunate side effect of a necessary maneuver by the Window Manager. Any system dialogs that set a background color and use EraseRect will suffer from the same flicker (although you won't spot it so often, since for the most part they're modal, nonresizable, and relatively small).

There are two solutions: If you create your windows from 'WIND' resources, you can create 'wctb' resources with the same ID and an appropriate wContent color and they'll automatically be used when the window is created. Alternatively, you can use the SetWinColor routine to apply a color table to a window after it has been created.

Q *How can we write native PowerPC versions of our QuickDraw GX printer drivers? Native QuickDraw GX applications are easy, but I can't find any documentation for writing native drivers. One problem I can see is the jump table at the top of the 'pdvr' code resource, where each of the jump table entries is supposed to contain a 680x0 instruction to jump to the appropriate override procedure within the resource. How should we proceed?*

A First of all, in our experiments with native QuickDraw GX drivers we've found little or no performance increase, so we don't really recommend writing native printer drivers. The bottlenecks in typical QuickDraw GX printer drivers are in the file and network I/O, which aren't affected much by the driver's code (see the Print Hints column in this issue of *develop* for more information). Unless you have some repetitive and time-consuming operation in which you can expect a huge win, your code will most likely just get bigger, and possibly even slower in some cases because of the context switches.

That said, I'll tell you how to make any or all of your overrides into "fat" overrides. Each of the fat overrides needs to consist of three parts:

- a "safe routine descriptor" (see below)
- the 680x0 code
- the PowerPC code

The safe routine descriptor allows a simple JMP instruction (such as those found in the 'pdvr' jump table) to run either 680x0 code or PowerPC code, depending on what type of Macintosh it's running on. The beginning of the safe routine descriptor contains 680x0 code that's executed the first time through and that determines whether the Mixed Mode Manager is present. If so, the routine descriptor from the PowerPC chunk of code is copied to the top of the resource. If not, a 680x0 branch instruction that jumps to the 680x0 code is inserted at the top of the routine descriptor. This code munging happens only the first time through — after that, the code resource is set up to run the 680x0 code or the PowerPC code immediately upon jumping to it. This is described in more detail in *Inside Macintosh: PowerPC System Software* and in MixedMode.r.

Here's how a fat override would be called from QuickDraw GX: First, execution jumps to the desired entry in the printer driver's jump table. The 680x0 JMP instruction in the table is executed and jumps to the safe routine descriptor in the override. The safe routine descriptor then executes either the 680x0 code or the PowerPC code, depending on the machine.

The real trick is building the makefile. First the makefile has to compile each of the override functions into a 680x0 code resource using C and Link, and into a PowerPC code resource using PPCC, PPCLink, makePEF, makesys, and Rez. Next, it needs to use Rez to combine the results of the compiles into fat resources with safe descriptors. Finally, it needs to use Rez to combine the jump table with all of the fat resources. This last step is a doozy. You'll need to write an MPW tool that concatenates all the fat resources while keeping track of the offsets of each one. These offsets will need to be stuffed into the jump table at the top of the final 'pdvr' resource.

Q *I'm trying to display from my QuickDraw GX printer driver a movable modal dialog box that contains a list. What would be the best way to do it?*

A Since you're displaying the dialog from a printer driver, you can let QuickDraw GX do most of the work. Use GXAlertTheUser to put up a 'plrt' resource that specifies the printingStatus type. That makes a movable modal dialog. Then override GXInitializeStatusAlert to build your list information for the dialog, and GXHandleAlertEvent so that you can update your dialog, handle clicks, dispose of the dialog, and so forth. You'll probably also want to override GXHandleAlertFilter to help out with that.

Q *I noticed that the "Larger Print Area" checkbox has been taken out of QuickDraw GX's LaserWriter Page Setup Options. I assume this means one of three things: (a) it's now on all the time, (b) it's now off all the time, or (c) it's lurking someplace else. Which is it?*

A The option has been removed, as you noted, but the functionality is still available, in the guise of papertypes. To decide where on the page to print, QuickDraw GX printing uses the papertype that a page is formatted for. Some of these papertypes come standard (built into QuickDraw GX or into specific drivers), but users can also create their own papertypes using the papertype editor that accompanies QuickDraw GX.

For example, a user could create a papertype called "Company Letterhead" that's based on US Letter but has an imageable area that excludes the part of the page at the top where the address and logo might be, as well as the line or two of text at the bottom of the page. When dropped into the Extensions folder, these custom papertypes become available from QuickDraw GX applications, and users can format their documents with them. In the case of this letterhead example, the result would be that you wouldn't have to fiddle around with your text to avoid printing over the type and graphics on the paper.

The "Larger Print Area" feature isn't compatible with the concept of papertypes because in order for QuickDraw GX to honor the imageable area of a papertype, it can't change the dimensions that the papertype is set up for. Again, using the letterhead example, if we were to expand the imageable area it would probably result in text printing over the company logo or address information.

We realized that some customers would still expect this functionality to be around, so we've included some papertypes in the LaserWriter GX driver that mimic the old behavior. When you format for the LaserWriter GX driver (or for a desktop printer for that driver), you'll notice that two new papertypes become available in the QuickDraw GX page setup dialogs: "A4 Letter (7.8 x 11.4)" and "US Letter (8.5 x 11)." These are "larger print area" versions of their counterparts. In applications that aren't QuickDraw GX aware, these papertypes can be found in the Page Setup dialog's Paper Type menu.

Q *I was wondering whether QuickDraw GX, like LaserWriter 8, uses PostScript printer description (PPD) files. How do PPD files and the new printing architecture integrate? What happens in QuickDraw printing compatibility mode?*

A The short answer to this question is no, QuickDraw GX does not use PPD files. The longer answer is, well, longer.

The main use for PPD files was to extend the functionality of the LaserWriter 8 driver. As you know, the contents of many of LaserWriter 8's dialogs depend on the contents of the PPD file. The user can associate a PPD file with a printer via the Chooser from LaserWriter 8 onwards, by using the Setup button in the Chooser dialog.

As you're also aware, the mechanism for choosing printers has changed with QuickDraw GX. This means that PPD files aren't needed, except for one case: using QuickDraw GX Helper to print with LaserWriter 8 with QuickDraw GX installed. In this situation, printing will occur in the same way as before — in other words, the print process will be the same as for plain old LaserWriter 8, including the use of PPD files. In QuickDraw printing compatibility mode (that is, printing with a pre–QuickDraw GX application on a system running QuickDraw GX), the emulation will use the QuickDraw GX driver, so PPD files won't be used unless the driver specifically supports them.

By the way, there's no reason a printer driver manufacturer can't incorporate PPD files into a QuickDraw GX printer driver, if it's deemed appropriate.

Q *When my application creates a new media (of text type in this case) for a new track in a movie created with NewMovieFromScrap, the dataRef and dataRefType should be set to nil, according to the QuickTime documentation. The problem is that later I want to edit that media (adding a text sample to it, for example), but BeginMediaEdits returns -2007 (no data handler found). I assume I can get around that by first saving the movie to a file, but this seems slimy since the movie won't end up on disk in the end. Any suggestions for a better approach?*

A You're correct — BeginMediaEdits complains if the movie has been created with NewMovieFromScrap. Unfortunately, BeginMediaEdits doesn't think memory-based movies are on a media that will support editing. The workaround is, as you thought, to store the movie in a temporary file until you're finished editing it.

Fortunately, this is easy to do. When you call NewTrackMedia, pass an alias to a new file in the dataRef parameter instead of nil. Passing nil (the usual approach) indicates that the movie's default data reference should be used, but because your movie came from the scrap and not a file, it has no data reference — hence the error you're getting. By the way, using the data handler in QuickTime 2.0, you can create a movie entirely in memory.

Q *I need to alter the pixel information of a QuickTime movie frame after it has been decompressed but before it's displayed on the screen. Is there any way that a user-defined procedure can be called by QuickTime at this point? If not, how can I accomplish this?*

A As long as you use QuickTime 1.6 or later, you can do this easily. The mechanism is described in the Macintosh Technical Note "QuickTime 1.6.1 Features" (QT 4) under the heading "SetTrackGWorld."

SetTrackGWorld lets you force a track to draw into a particular GWorld. This GWorld may be different from that of the entire movie. After the track has drawn, it calls a "transfer procedure" that you've written to copy the track to the actual movie GWorld. You can also install a transfer procedure and set the GWorld to nil. This results in your transfer procedure being called only as a notification that the track has drawn — no transfer takes place.

You should do your image manipulation in your transfer procedure. Bear in mind, of course, that calling resource-intensive or time-consuming routines in your transfer procedure will have an adverse effect on the playback performance of your movie.

Q *I'm looking for the documentation for five routines in the Communications Toolbox that aren't in my 1.0 documentation. The routines are "PBxxx-style" asynchronous routines: CMNewIOPB, CMDisposeIOPB, CMPBRead, CMPBWrite, and CMPBIOKill. Where are these calls documented?*

A These calls are documented in the Communications Toolbox 1.1 Engineering Notes. The calls were added to allow "overlapping I/O," that is, issuing a _CMWrite call and then issuing another before the first completes. The Communications Toolbox version 1.0 specifically prohibits this behavior.

Q *I have an application that needs to be able to detect invisible folders. The "invisible" bit for files is in the fdFlags in the FInfo structure, but that obviously won't work for directories. What's the recommended way to do this?*

A PBGetCatInfo will get you the information you need. The ioDrUsrWds field of the DirInfo structure that you get by calling PBGetCatInfo is a DInfo structure:

```
struct DInfo {
   Rect            frRect;       /* folder rect */
   unsigned short  frFlags;      /* flags */
   Point           frLocation;   /* folder location */
   short           frView;       /* folder view */
};
```

The frFlags field has the same layout as the fdFlags field in an FInfo structure. Many of the bits don't apply to directories, but fOnDesk and fInvisible do. The fInvisible bit is set for almost all invisible folders. The only exceptions to this are certain special folders that the Finder can create, such as the "Temporary Items" folder, which you shouldn't have to worry about.

Q *When a user of my application saves a file, I want to automatically save a few files, each of which has a different extension to the name the user chose. For instance, if the user saves the file as "MyFile" I want to create "MyFile.a," "MyFile.b," and "MyFile.c." If the user gives a filename that already exists, but the filename with the extension doesn't exist, is there a way to avoid the appearance of the Replace/Cancel dialog? Conversely, can I make the Replace/Cancel dialog appear for the filenames with the extensions? I assume CustomPutFile is the way to go here, but I'm not sure how to proceed.*

A CustomPutFile is the answer, all right. One way to get rid of the Replace/Cancel dialog is to write a simple dialog hook function that includes the following code:

```
/* If it's the "Replace Existing?" dialog... */
if (refCon == sfReplaceDialogRefCon)
   /* ...and the dialog is just about to appear... */
   if (item == sfHookFirstCall)
      /* ...then "hit" the Replace button automatically. */
      return 1;
```

The problem you pose is a little different, however, so we recommend a different approach. In addition to avoiding the standard Replace/Cancel warning, your dialog hook function will also need to make sure each filename-plus-suffix combination is a valid filename (that is, not too long), put up Replace/Cancel dialogs in case any filename-plus-suffix combination already exists, and override other warnings, such as the "That name is already used by a folder" dialog.

First, define a data structure that can be passed to CustomPutFile so that your dialog hook function can access the list of suffixes you're working with and return the results (that is, the FSSpec and FInfo) of each filename-plus-suffix combination. Then, your dialog hook function should do the following when the user clicks the Save button:

1. Make sure that each filename-plus-suffix combination is a valid filename.
2. Put FSSpecs for each filename-plus-suffix combination into the data structure so that you can get at them later.
3. Put up Replace/Cancel dialogs for any filename-plus-suffix combination that already exists.
   a. If the user cancels any of these, remap the Save item to nothing by returning sfHookNullEvent.
   b. If the user accepts all of these, remap the Save item to Cancel by returning sfItemCancelButton and, before returning, mark the sfReply as good, so that your application can tell that the user really clicked the Save button. By changing the Save item to the Cancel item, you bypass all the standard warnings, which means that your dialog hook function is responsible for warning the user if there's anything wrong with the filename. These warnings should maintain the spirit of the normal Standard File warnings as much as possible.

Inside Macintosh: Files, pages 3-26 through 3-40, provides additional information on this subject.

Q *I need to obtain the location of my application from within my application. How can I do this?*

A The Process Manager routine GetProcessInformation returns an FSSpec to the current process if you use the process serial number kCurrentProcess, as shown in the following code:

```
OSErr GetCurrentProcessSpec(FSSpec *spec)
{
   ProcessSerialNumber  currentPSN;
   ProcessInfoRec       info;

   /* Get current process FSSpec with GetProcessInformation. */
   currentPSN.highLongOfPSN = 0;
   currentPSN.lowLongOfPSN = kCurrentProcess;
   info.processInfoLength = sizeof(ProcessInfoRec);
   info.processName = NULL;  /* we don't need process name here */
   info.processAppSpec = spec;
   return (GetProcessInformation(&currentPSN, &info));
}
```

If GetCurrentProcessSpec returns with a noErr result, spec.vRefNum is the volume reference number of the volume your application file is on, spec.parID is the directory ID of your application file's parent directory, and spec.name is the name of your application file.

If your application is a 680x0 application and might run under System 6 (where the Process Manager and the Apple Event Manager aren't available), you can use GetAppParms to get the reference number of your application and then pass that number to PBGetFCBInfo to get the location of your application. PowerPC applications must use GetProcessInformation because GetAppParms isn't available to them.

The following code shows how to use GetAppParms and PBGetFCBInfo to get the location of your application. You must define OBSOLETE in your source code before you include SegLoad.h; otherwise GetAppParms (and the other obsolete Segment Loader routines CountAppFiles, GetAppFiles, and ClrAppFiles) will not be defined.

```
/* Obsolete System 6 way of getting the application location. */

#define OBSOLETE
#include <SegLoad.h>
#include <Files.h>

OSErr GetCurrentAppLocation(short *vRefNum, long *parID, Str63 apName)
{
   OSErr       result;
   FCBPBRec    fcbPB;
   Handle      apParam;

   /* Get application reference number from Segment Loader. */
   GetAppParms(apName, &fcbPB.ioRefNum, &apParam);

   /* Get application location from File Manager. */
   fcbPB.ioNamePtr = apName;  /* return application name here */
   fcbPB.ioVRefNum = 0;
   fcbPB.ioFCBIndx = 0;
   result = PBGetFCBInfoSync(&fcbPB);
   if (result == noErr) {
      *vRefNum = fcbPB.ioFCBVRefNum;
      *parID = fcbPB.ioFCBParID;
   }
   return (result);
}
```

Q *I have a handle to a resource and I want to find the location of the file it came from. Can I do this?*

A Yes, you can use HomeResFile to get the file reference number associated with the resource and then pass that number to PBGetFCBInfo to get the location of the resource file, as follows:

```
OSErr GetFileLocationFromResource (Handle theResource, short *vRefNum,
                        long *parID, Str63 name)
{
   OSErr       result;
   FCBPBRec    fcbPB;

   /* Get resource file reference number from Resource Manager. */
   fcbPB.ioRefNum = HomeResFile(theResource);
   result = ResError();
   if (result == noErr) {
      if (fcbPB.ioRefNum != 1) {      /* Is resource in ROM? */
         if (fcbPB.ioRefNum == 0) {   /* Is it in the System file? */
            /* Get System file's real refNum. */
            fcbPB.ioRefNum = LMGetSysMap();
         }
         /* Get resource file location from File Manager. */
         fcbPB.ioNamePtr = name;   /* return filename here */
         fcbPB.ioVRefNum = 0;
         fcbPB.ioFCBIndx = 0;
         result = PBGetFCBInfoSync(&fcbPB);
         if (result == noErr) {
            *vRefNum = fcbPB.ioFCBVRefNum;
            *parID = fcbPB.ioFCBParID;
         }
      }
      else {
         /* Resource was in ROM, not a file. Return paramErr. */
         result = paramErr;
      }
   }
   return (result);
}
```

Q *I need to write an extension that will launch an application at a specified time. I've looked at various ways to do this, but they all seem dangerous or difficult, mostly because I can't call either LaunchApplication or AESend (to send an 'oapp' event) at interrupt time. What's the best/safest/simplest way to do this?*

A Write a BOA (background-only application, also called a faceless background application) that just sits in the background and periodically checks the time. When the correct time arrives, the BOA can launch the application and quit (or remain running if this is a periodic task). Make your BOA have a file type of 'appe' (application extension) and it will be installed in the Extensions folder and launched at startup.

Q *I need to insert a two-dimensional array, such as "long double myArray[512][3]," into an Apple event. Is there any way to use AEPutArray? I know that I can probably loop over each item and add it to a list of typeFloat, but that's extremely cumbersome and slow. Also, I could send it as typeData, but that wouldn't let users view the data in a script editor. Any ideas? I'm looking for the most elegant and speedy solution. I'll also need the inverse: a way to extract that data from an incoming Apple event.*

A There's no standard way of implementing two-dimensional arrays in Apple events. Apple event arrays are limited to one-dimensional arrays of integer, char, handle, or descriptor types. While it may be possible to represent your data in one of these formats, this is not going to be a very efficient solution.

There are several other possible solutions to the problem, but only you can decide which one will work best. Perhaps the most straightforward is to create an AEList, which in itself contains a set of AELists, to represent your data. While this will allow your data to be displayed by the Script Editor, it could be very inefficient for large arrays. It's also consistent with the way that the Table Suite specifies a table (which is basically what your two-dimensional array is): type cTable consists of a list of typeRow descriptors, which in turn consist of a list of typeCell descriptor records.

Another more efficient approach (which is also more trouble) is to use your own private data format and install a system coercion handler to coerce this data into a form that can be displayed in the Script Editor (typeChar, for example). When the data is returned from your GetData handler, the coercion handler will be called to translate the data, so it would be displayed as text in the results window.

Do you really need to provide a two-dimensional array in a form that can be read in the Script Editor? Often the user will just want to request certain elements of the array, rather than requesting the entire array. The best solution of all might be to avoid sending large arrays completely, at least in a format that can be displayed in the Script Editor. There's very little the user can do to view and manipulate large amounts of data from the Script Editor anyway. You could allow users to request pieces of data from the array, using a row and column approach to allow them to specify the data, but when they need to manipulate large amounts of data you might consider writing the requested data to a file and then returning an alias to the file.

Q *I'm implementing a dialog in which seven items need to be present all the time while the presence of the other items depends on various situations. I use Hide/ShowDItem to do the trick. Is there any other way to do it? The dialog's DITL is really a mess!*

A The approach of showing and hiding individual items is fine if you have only a couple of items in the dialog item list, but as you've found, it becomes a real mess when the dialog starts to become more complex.

You can dynamically add items to and remove items from a dialog box by using the AppendDITL and ShortenDITL routines. When you call AppendDITL, you specify a dialog box and a new item list resource to append to the dialog's existing item list resource. You also specify where the Dialog Manager should display the new items, using one of these constants to designate where AppendDITL should display the appended items:

```
CONST    overlayDITL          = 0;     {overlay existing items}
         appendDITLRight      = 1;     {append at right}
         appendDITLBottom     = 2;     {append at bottom}
```

You should create one dialog with the seven items that remain constant, and a series of associated DITL resources that contain the items you need to add to each variant of the dialog. Then use AppendDITL to add these as required.

Q *If I call the following stripped-down routine twice, my application crashes the second time. Why?*

```
static void CrashMeBaby (void)
{
   Rect        aRect;
   DialogPtr   aDialog;

   SetRect(&aRect, 50, 50, 200, 200);
   aDialog = NewDialog(NIL, &aRect, "\p", true, altDBoxProc,
            (WindowPtr) -1L, false, 0, GetResource('DITL', 400));
   DisposeDialog(aDialog);
}
```

A The problem is that DisposeDialog disposes of the item list (which you're obtaining with GetResource) by calling DisposeHandle, not ReleaseResource. This leaves an invalid reference lingering in the resource map, which is bad news. The next time the resource is needed, it would normally be read in again from disk. However, in this case the handle is no longer valid, and you crash. The workaround for this is simply to call DetachResource on your item list handle after you retrieve it.

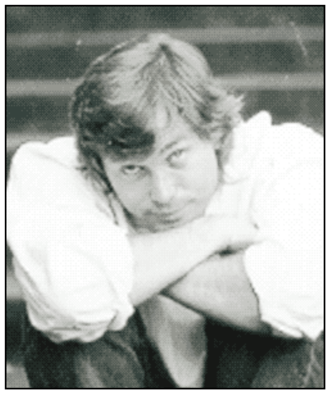
One way of finding this kind of bug is to use the DisposeResource extension, which can be found on this issue's CD. This traps instances of DisposeHandle being called to dispose of a resource. If you install DisposeResource and try it with your code, you'll see that this is what's happening in your case.

Q *How do fleas jump so high? Surely the power required for these prodigious leaps (easily 100 times their length!) can't be supplied by muscle. How do they do it?*

A Fleas have an organ (not a muscle) that's elastically deformable and can store energy like a rubber band. It's "charged up" over time, with a sort of ratcheting muscular action. So the flea "winds up" and then lets go all at once. An interesting side effect is that just after a flea jumps, it's unable to jump again; it needs time to recharge. (The time needed, however, is a mere tenth of a second.)

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Brian Bechtel, Joel Cannon, David Hayward, Dave Hersey, Peter Hoddie, Dave Johnson, Jon Lansdell, Jim Luther, Kevin Mellander, Jim Mensch, and Nick Thompson for the material in this Q & A column.•

Have more questions? Need more answers? Take a look at the Macintosh Q & A Technical Notes on this issue's CD.•



DAVE JOHNSON

THE VETERAN NEOPHYTE

Nothing Comes From Nothing

Take a good look around at all the different structures you see in the world. Not only the physical structures, like buildings and mountains and dogs and trees, but the conceptual structures, like language and government and the Dewey decimal system and the management hierarchy at your place of work. Some of these structures seem to exist independently of humans, but others seem to be completely invented, made up from whole cloth, so to speak. But how can that be? Can you really create something from nothing? I wonder.

I just got back from a sort of educational/recreational summer camp for adults (I'm writing this in early August). It was the California Coast Music Camp, a week of intensive classes and refreshing musical single-mindedness. There were no cars, no computers, no worries — nothing to do but play and sing and learn (and swim and eat and hike).

The camp itself was classic: squat brown bunkhouses scattered in small groups like big hollow dice; long low latrines, redolent with that cloyingly sweet chemical peculiar to outdoor bathrooms; enormous vats of jello and drowned salad at the ends of the long serving tables in the echoing dining hall; bug bites, bug spray, and bugs — you get the idea. And even though it was strictly adults, all the same feelings I remember from my one camp experience as a kid were there in the wide and ragged spectrum of emotions it generated. I had the same scary and uncertain feelings at the beginning, wondering if I really should be there at all; the same gradual discovery that it was all OK, and in fact was fantastic; and, at the end, the same bittersweet longing to start it all over again.

Music really is a lovely thing. One of the classes I took was called "Theory, Scales, and Chord Construction on the Guitar" and it was on the third day of class, as I was plunking away at yet another arpeggio, that I began to get my first glimpse of the underlying structure of the notes on the guitar fretboard. (And I do mean glimpse: it's something that will probably take a year or more to see clearly, and five or ten years to really feel — and that's if I practice every day.) I was suddenly struck by the notion that it's the discontinuities in musical scales that make them both difficult and interesting, that it's the complications in the structure of music that give it an interesting shape. If you've ever taken a music theory class, or even a piano class, you might know what I mean: if only there were a black key between every pair of white keys, things would be much simpler. It's that damned half step between B and C and between E and F that screws things up. But it also seems that those discontinuities — those bumps and dips in an otherwise smooth, even progression — are what give rise to all the beauty and complexity and subtlety. It's precisely those "flaws" in the structure that lend it an interesting texture.

However, the structure itself is artificial. Underneath, the range of musical tones is actually continuous, as any slide whistle demonstrates. It's a spectrum, a continuously varying quantity, in this case the frequency of vibration of a material. So the set of notes we use — our tuning system — is externally applied, a necessarily arbitrary set of discrete slots pasted onto an underlying continuum. It's a sort of quantization of something inherently smooth that lends it a tractable structure, that gives us a handle with which to manipulate it and a context in which to make sense of it. By conceptually making the smoothly varying curve into a step function, by arbitrarily chopping the continuous line into discrete chunks, we can somehow work with it more manageably and think about it more clearly.

If you look around a little, you'll find examples of this kind of artificial structuring of continuums all over the place. The computer in front of you (or wherever it is) is an excellent example. Many early computers were analog; they dealt with smoothly varying quantities (usually voltages in circuits). But they turned out to be too hard to program — in effect you had to create a physical circuit that modeled the problem you wanted to solve. By making the computer purely digital, we

DAVE JOHNSON has for years had the same favorite quote, from Albert Einstein: "He . . . who can no longer pause to wonder and stand rapt in awe is as good as dead; his eyes are closed." But recently he encountered another, in a book by Primo Levi, that came very close to unseating Einstein's: "It is enough to think of intestinal worms: they feed themselves at our expense with a food so perfect that, unique in creation, together perhaps with the angels, they have no anus." Although Dave howled uncontrollably at that one, Einstein still wins in the end.•

abstracted its operation away from the physical realm into the realm of pure logic, and that really opened up a lot of doors. Another much simpler example is a radio dial; it actually represents a continuous spread of frequencies, but we've arbitrarily divided it into bands so that we can parcel it out to those who want to use pieces of it.

More abstractly, in mathematics one often takes a continuously varying function and "pretends" for a moment that it's a step function, simply because it makes things easier to deal with. Then, when you've got a handle on the step function, you can use a nifty trick (called calculus) to sort of extrapolate what you've discovered about the steps and apply it to the whole curve. Time itself (which sure feels like a continuum, whatever it is) is conveniently chopped into bits by humans to make it easier to keep track of and to talk about.

But there are other quantities in our world that seem to come to us already divided up into discrete chunks, already structured. The periodic table of the elements certainly isn't a continuum. There's no smooth transition between sodium and magnesium, though they sit next to each other in the table. The distinction between the phases of matter — solids, liquids, and gases — seems pretty clear, too. (Well, OK, there are some bizarre in-between states you learn about in college, but they're encountered only in extreme conditions, usually artificially induced in laboratories and definitely inhospitable to mammals.) Living things appear to be made up of lots of discrete functional blocks — organs and cells and organelles and protein molecules and such — and the interactions of these discrete parts are what makes them "go." DNA itself, the structure that stores the instructions for building, is just a binary (or rather quaternary) string, with each discrete position capable of storing only four possible values. Even the seeming continuum of a fluid like water is an illusion. In reality it's made of discrete particles, and it's their interactions with one another that give rise to "fluidness."

This endless interplay between the continuous and the discrete, between discovering structure and creating it, seems to be at the heart of many (maybe most) human endeavors. On the one hand, we often labor mightily to reveal structures that are somehow already there. Many of the sciences, in particular, are precisely an attempt to make clear the underlying structure of the universe, to peel away the layers of obfuscation that our senses have piled on. But it's not limited to science: Michelangelo spoke of sculpting not as inventing the shape of the statue, but rather as freeing it from the stone in which it was imprisoned.

On the other hand, many human activities are all about applying structure to something formless, or about creating structure from nothingness. Again, the arts spring to mind. A painting, a poem, a story, a song — all these begin from nothing: from a blank canvas, from an empty sheet of paper, from silence. Computer programs, those awesomely complex logical constructions we devote ourselves to so slavishly, seem to be created from thin air, and serve as the structure for an otherwise "formless" machine. Business contracts, sheet metal ventilation ducts, bingo games, steering committees, and acoustic guitars — these are all structures that we've created from, essentially, nothingness.

Ah, but there's the real question: are the structures we build really new? Do we really just invent them, whole, from nothing? Or do they grow from and reflect other underlying structures that are already there? The latter seems much more likely to me. If you look closely, even a structure that at first blush seems really new turns out to be a recombination or an extension or a reworking of some existing structure. Sometimes I think of evolutionary processes this way, as a sort of extrapolation, a patient elaboration, of a very few essential, innate principles that lie buried far beneath the surface, an endless cycle of structure standing on the shoulders of what has gone before. Perhaps the macroscopic shapes of living things hint at the underlying nature of matter: quantum reality writ large, for all to see.

So, getting back to music, does the tuning system that we use reflect some underlying structure, some relationship among the frequencies, or is it truly arbitrary, decided by some bewigged old coot in the dim and dusty past? To find out, I did a little snooping around in the local library and on the net. The answer turns out to be complex (no surprise there), and it has as much to do with accidents of history, people's personalities, and the practicalities of tuning instruments as it does with mathematics. There is an underlying structure beneath the western tuning system and many others (the relationships among the harmonics of a vibrating string or column of air, first elaborated by — you guessed it — the Greeks). But it also turns out that most music has evolved away from those "ideal" frequencies for a variety of reasons that have little to do with mathematics or physics. Here's what I think: I think the structure of music is largely determined by the structure of us.

Sunburst shapes — mandalas — appear over and over in pictures drawn by children, no matter what their culture or language or what part of the world they live in. You could argue that it's simply because there are lots of radially symmetric things in the world, and that children are simply drawing what they see. But I prefer to think that mandalas somehow mirror the internal structure of the human mind, that they are, in a sense, pictures of humanity.

Humans spend enormous amounts of time shaping things, refining things, expressing things, creating. Where does all the structure come from? I think it's simply an outgrowth, an elaboration, a reinterpretation and repackaging, of the structure inside ourselves, which in turn reflects the structure of the universe we live in. Taken all together — all the songs, all the buildings, all the stories, all the social groupings and computer programs and bad jokes and trash and art — the structures we humans reveal and create form a churning, turbulent, clouded mirror, a mirror that occasionally, if we look very closely, may afford us a glimpse of who and what we really are.

RECOMMENDED READING

- *Catapult: Harry and I Build a Siege Weapon* by Jim Paul (Villard Books, 1991).
- *Weetzie Bat* by Francesca Lia Block (Harper & Row, 1989).
- *Cane Toads: An Unnatural History* by Stephanie Lewis (Doubleday, 1989).

Thanks to Lorraine Anderson, Jeff Barbose, Brian Hamlin, Mark "The Red" Harlan, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their always enlightening review comments.•

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe.•

Hard at work

Meredith Best, Dave Johnson, Caroline Rose, and Alex Dosher deal valiantly with the unexpected ups and downs of working on *develop* at an offsite meeting held at a local amusement park. As the meeting coasted to a close, they all agreed that they should do it again. And again. And again.

Newton Q & A: Ask the Llama

Q *I could really use some help with speeding up my Newton application. Have you got any tips on performance?*

A You're not the only one who wants this; my llama senses have recently been overwhelmed by a call for information on performance. All the questions in this issue's column will relate to performance in some way. Take a look and see if there's something here that will help you.

There are two important points to remember:

1. None of these tips will work by themselves; you must *measure* your code. Use Ticks, use the **trace** global (see below), use Print. Find out where your code is slow, or where your application is bloated.
2. There is no silver bullet for a problem; you must *experiment* with different solutions.

In the words of my wise programming master: "When is a llama not a llama? . . . When it is a guanacos." Or, "When you can snatch these coconuts from my hand, then it will be time for me to leave."

Q *I'm building an application that has a large set of static data. I search on a key term (a string) and get all the data associated with that string. Mike Engber's "Lost In Space" article (in the May 1994 issue of PIE Developers magazine) says that I should include this data in my package and things will be fast. But this doesn't seem to be the case. I have thousands of frames of data. Each frame contains one or more slots with strings that contain the key terms. I use FindStringInFrame to find all references to a key term but this takes a long time. Am I doing something wrong?*

A This may seem like a simple question, but it isn't. The root of the problem is that you've made an assumption that functions provided in the ROM are fast, so they'll solve your problem. In this case, you assumed that FindStringInFrame would be fast. You're both right and wrong.

FindStringInFrame is fast, but it still has to linearly search every slot in every frame recursively. That means that if you have thousands of entries, it's checking thousands of frames. You can talk about how long something will take by calculating the worst case. FindStringInFrame has to search all your data frames (thousands of items), and for each frame it has to check each slot to see if it's a string. If so, it then has to check to see if the string you gave it matches the string it's looking for (step by step down the string). So if you had *n* strings (not just data items), and the average length of a string was *m* characters, that's *n***m* checks. In computer science terms, you would say that FindStringInFrame is an O(*n* **m*) operation; this is called Big-Oh notation and, in its simplest form, refers to the worst-case time.

This means you should think about other data structures and methods of accessing them. In your case, a simple change of data representation would result in a massive speedup. The idea is to make the expression in the Big-Oh notation have the smallest possible value. One way to do this is to reduce the

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Personal Interactive Electronics (PIE) division.•

Send your Newton-related questions to NewtonMail DRLLAMA or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.•

search time for your key phrases. Since you have a fixed set of data, you can sort them and use a binary search algorithm. You can store the actual data in arrays and store indexes along with the key items.

The nice thing about a binary search is that you're always cutting your search space in half. On average, you only have to check log to the base 2 of the data. In Big-Oh notation, that's O(log *n*). Of course you still have to do the individual string comparisons, so you end up with O(*m* log *n*). So for 1000 items, FindStringInFrame takes 1,000,000 time units, but the modified method takes 3000, a speedup of 300 times! It's unlikely that a function implemented at a low level performs 300 times faster than custom NewtonScript code.

This excursion into computer science should make you think about your data structures and how you access them. Of course an academic exercise can take you only so far. You also have to get your feet wet and test the code. You can use Ticks to get rough estimates of time, and Stats (after a GC) to get estimates of memory.

Q *The following is a viewClickScript from a pickList button in my application. Why does it take so long to execute?*

```
viewClickScript.func(unit)
begin
   currentPickItems := [];
   for i := 0 to Length(defaultPickItems) - 1 do
      if i = currentSelectedItem then
         AddArraySlot(currentPickItems,
            {item: defaultPickItems[i], mark: kCheckMarkChar});
      else
         AddArraySlot(currentPickItems, defaultPickItems[i]);
   if :TrackHilite(unit) then
      DoPopUp(currentPickItems, :LocalBox().right+3,
             :LocalBox().top, self);
end
```

A There are several possible reasons why your code would execute slowly. Since they potentially apply to lots of code out there, I'll go through each one separately. At the end is a rewritten function that should execute considerably faster.

- Lookup costs. Assuming that currentPickItems, currentSelectedItem, and defaultPickItems are slots somewhere in your view hierarchy, at best they're slots in the pick button, at worst they're in your base application view. Remember that each access to a variable requires an inheritance lookup: check locals, then globals, then current context, then the _proto chain, then the _parent chain. This cost isn't high for single references but can be deadly in loops. Every cycle through your loop, you're doing three lookups; that's a lot of overhead. The solution is to use local variables for faster access.
- Unnecessary object creation. The AddArraySlot call will grow, and potentially copy, the array on the NewtonScript heap, resulting in a lot of unnecessary memory movement. Since you know the length of the currentPickItems array in advance, you should preallocate the array and use the array accessor (that is, [n]) to add array elements. You can use the Array function call to allocate the array:

```
local pickItems := Array(Length(defaultPickItems), nil);
```

- Unnecessary execution. You need to create a new pick list only if the call to TrackHilite succeeds. You should make the TrackHilite conditional be the outer conditional:

```
if :TrackHilite(unit) then
   begin
      // construct pick list and DoPopUp
      ...
   end;
```

- Inefficient variable initialization. It's inefficient to use a loop for initializing currentPickItems from defaultPickItems, because currentPickItems has only minor differences. It's better to use Clone for initialization. This way you get a new array whose elements are references back to the array items in defaultPickItems. All you need to do is replace the individual references in currentPickItems with their new or modified values. It's the difference between an O(*n*) operation (traversing all the array items in defaultPickItems) and an O(1) operation (accessing only the changed item). In other words, expect about an order of magnitude difference.
- Unnecessary slot. In this case you don't need to have a currentPickItems slot since its value is recreated each time the viewClickScript is executed. You're better off using a local variable.

The modified code is shown below. To illustrate the savings, I ran a brief test using a defaultPickItems array of ten elements. Each function is called 100 times (note that TrackHilite was always true). I found the following code to be over six times faster than the original code.

```
viewClickScript.func(unit)
begin
   if :TrackHilite(unit) then
   begin
      local pickItems := Clone(defaultPickItems);
      local selectedItem := currentSelectedItem;
      local l := :LocalBox();
      if selectedItem then
         pickItems[selectedItem] :=
            {item: pickItems[selectedItem], mark: kCheckMarkChar};
      DoPopUp(pickItems, l.right+3, l.top, self);
   end;
end
```

Q *I've written my own IsASCIIAlpha, IsASCIINumeric, etc. functions. They seem to be really slow. Why is that? Here's my IsASCIIAlpha:*

```
// returns true if s is an alpha string (i.e., between a..z or A..Z)
IsASCIIAlpha.func(s)
begin
   local c := Upcase(Clone(s));
   local i;
   for i := 0 to StrLen(c) - 1 do
      if (StrCompare(SubStr(c, i, 1), "A") < 0) or
         (StrCompare(SubStr(c, i, 1), "Z") > 0) then
         return nil;
   true;
end;
```

A The main source of the slowness is that you're using string functions when character functions would be faster. The distinction is subtle but important. In the code above, you loop through each length 1 substring of the target string to determine whether it's an alpha character. All this takes time. The Upcase call is O(*n*), as are the SubStr and StrCompare. Of course, the StrCompare isn't really that slow, but it's still slower than you need.

The SubStr call is returning a single character at a time, but in the form of a string. That means there is a memory allocation for at least two characters (the content and the null terminator) for each call to SubStr. A better way is to compare each character of the string. In certain circumstances you can access a character at a time with the array accessor (that is, []). An example of a function that does this is IsASCIIAlpha3 (see the code on this issue's CD). In general, when you need either a single character from a string or character-by-character access, the array-like syntax is faster.

Note that the final fix to the code is that it doesn't do any preprocessing of the string; instead it uses a lookup in an pregenerated array of valid alphabetic ASCII characters. That gives it a significant speed advantage. Since timing in the Inspector is a useful technique, the code to do the timings and print results is included on the CD. Also note that this function is specifically for ASCII characters, so characters like é and ß would fail.

Something else to note: Newton is a Unicode-based device. ASCII is a subset of Unicode (from 0x0000 to 0x007F), but Unicode characters up to 0xFFFD are documented. Your routine is checking only some of the characters on page 0 (that is, characters of the form 0x00*nn*), but it must deal with *all* characters.

Q *I'm trying to use the* ***trace*** *global to get information on what methods are called. But I get lots of output that doesn't start or end where I want. What can I do?*

A There are really two questions here: how to use **trace** effectively, and how to use the output. Usually you would turn tracing on inside a method, then turn it off later on in the code. Unfortunately, you need to do more than just set the value of **trace**; you also have to force the interpreter to notice that **trace** has changed. The PIE Developer Technical Support NewtonScript Q&A on debugging (on this issue's CD, among other places) tells you how to do this.

```
// to turn tracing on for functions
trace := 'functions;
// force interpreter to notice change in state of trace variable
Apply(func () nil, []);

// to turn tracing off
trace := nil;
Apply(func () nil, []);
```

Once you have the trace output, you should cut and paste it into a text processor. There are three main bits of information you can get from a trace:

- You can look at how many messages are generated from an apparently simple call. You can use **trace** in conjunction with function call timings made using Ticks to see why a particular call takes so long. Using the find feature of your text processor, you can jump to the function call you're looking at.

- You can look at the values passed in and returned by function calls.
- Perhaps most useful of all, you can use the text processor to strip away all the extraneous information (things like the lines specifying return values — that is, lines that contain the string "=>" as the first non-whitespace entry) so that you're left with the messages sent. Then you can sort the messages and get a histogram of the results. This process is easier if you have a text processor that supports **grep**-like text substitution (regular expressions) and sorts.

Q *I'm using the Newton Toolkit layout editor to organize my data object classes in my application. I have 20 classes with one layout per object type. To access the objects, I declare each class layout to the main application. This gives me the benefits of parent inheritance. Unfortunately, even my test applications are memory hogs. I would expect a time penalty, but why is there such a large space penalty?*

A The space penalty is much larger than it needs to be. You're using a layout editor to edit your classes so that you can graphically edit the classes' slots. But this has the disadvantage that you have to specify each class as some sort of view class or prototype, perhaps a simple clView. It's the cause of your space problem, because you also carry all the memory and runtime allocation that goes with a view. Since your layouts are declared to your base application view, and since the default for a clView is visible, each of your classes is also a full runtime view. That can take a large amount of space on the NewtonScript heap. For a clView, the penalty is roughly 40 bytes, so that's an extra 800 bytes of NewtonScript heap that you can free.

A better solution is to avoid using the NewtonScript heap for your class (after all, that's one of the advantages of prototype inheritance). You can do this in one of two ways:

- If you still want to use a layout editor to edit your class, you can use a user prototype instead of a layout. At run time, you'll have access to the data class using the PT_<filename> syntax documented in the *Newton Toolkit User's Guide* (page 4-25). Remember that the user prototype will be read-only.
- The other option is to textually define the class. You can do this in your Project Data file, or use the Load command to read in a different text file. See the PIE Developer Technical Support NewtonScript Q&A document on this issue's CD for more information.

Thanks to our PIE Partners for the questions used in this column, and to jXopher, Bob Ebert, Mike Engber, Kent Sandvik, Jim Schram, and Maurice Sharp for the answers.•

Have more questions? Need more answers? Take a look at PIE Developer Info on AppleLink.•

Processed Cheese

KON and BAL still lay claim to the Puzzle Page, and they assure us they'll be back, but meanwhile they'd like to have some guest puzzlers take over for a while. (They say they're busy with work at Catapult Entertainment and Rocket Science, but we all know about those vacations they take!) This puzzle is from Cary Clark, presented in the form of a dialog between his astute pug dogs, Shelley and Byron (who, Cary says, will eat anything, including codecs and Texas Hold'ems). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. And please, help out KON and BAL by submitting puzzles of your own to AppleLink DEVELOP.

CARY CLARK, BYRON, AND SHELLEY

Shelley KON and BAL have gone the way of the dodo bird (or at least the way of the hedgehog), so it's time to replace all of their arcane QuickDraw knowledge with arcane QuickDraw GX knowledge.

Byron I tried to use QuickDraw GX, but since installing it off the 1994 WWDC CD, I can't launch any of my applications.

Shelley Are you sure?

100 Byron Well, the applications launch, but double-clicking a document doesn't do anything, even though the icons look all right.

Shelley The problem is in the desktop database, which the Finder uses to tell which document file types correspond to which applications. Did you try rebuilding your desktop?

90 Byron Sure, but it didn't help.

Shelley How about booting off a different disk?

85 Byron That works, but only for some files. So I removed QuickDraw GX altogether and rebuilt the original desktop by holding down Option-Command at system startup. Then everything worked fine.

Shelley That must mean that one of the files you removed got rid of the problem.

CARY CLARK, once on the QuickDraw GX engineering team at Apple, has joined BAL at Rocket Science. He's co-owner of Shelley and Byron, who often know more than he does. Cary was working on QuickDraw when BAL made Microsoft Word skanky and KON made GM (not General Magic) what it is today. Next time you see KON, ask him if he wants to cut for a hundred and watch him wince.•

80 Byron Nope, grr. I tried explicitly taking the QuickDraw GX extension out of my Extensions folder and rebooting, and nothing happened, except that my desktop printers went away.

Shelley So how do you like the LQ, anyway?

Byron It’s a lot faster than the Qume. But I miss watching it hammer the period to make my LisaDraw pictures.

Shelley So, what else is a part of QuickDraw GX? Let’s take a peek in the System Folder. There’s the printer drivers, ATM . . .

Byron That’s never caused any trouble.

Shelley . . . ColorSync, the new Color Picker, PrinterShare GX. Hmmm, that’s odd — PrinterShare GX’s icon is dimmed as if it were an open application, but it doesn’t show up in the process menu as PrintMonitor used to.

75 Byron Oh, you dog. That’s because PrinterShare’s file type is 'appe'. It’s a faceless application that’s always running in the background.

Shelley I don’t remember reading about that in the Processes volume of *Inside Macintosh*.

Byron That’s because dogs can’t read. But I’ve heard it’s briefly mentioned in Volume VI on page 9-41. Anyway, that’s beside the point. If you type **procinfo** in MacsBug, you’ll get a list that looks like this:

```
Displaying Process Information
PSN  Process Name      Size     Free     HeapAt   Type Crtr Status
2000 PowerTalk Manager 00044800 00003AE4 05892960 appe kl02 BgOnly
2002 Finder            00026C00 000012DC 05846E80 FNDR MACS Bkgnd
2003 File Sharing Ext… 00029C00 00005A2A 057A2C60 INIT hhgg BgOnly
2005 Eudora 1.4.2      0005CC60 000154F8 057242E0 APPL CSOm Front
2006 NCSA Telnet 2.6   00088000 0003E0B2 056982D0 APPL NCSA Bkgnd
2007 THINK Project Ma… 003E8000 001E1A68 052AC2C0 APPL KAHL Bkgnd
2008 Find File         00046000 00014F4A 052622B0 APPL fndf Bkgnd
200B Microsoft Word    00200000 000C1D5C 0505E2A0 APPL MSWD Bkgnd
200C PrinterShare GX   0001C000 00011488 0503E290 appe PtSr BgOnly
```

Shelley OK, try removing PrinterShare GX and rebuilding the desktop.

70 Byron Hey, that fixed it!

Shelley But the question remains, what’s wrong with PrinterShare GX? And how did you get into this sorry situation anyway?

65 Byron Well, I ran Norton Utilities on my disk, and it said it was fixing some applications with bad bundle bits.

Shelley So Norton must force the desktop to rebuild in order to register the applications it thought needed to be reregistered with the Finder’s desktop database. Bad dog. And the Finder fails when rebuilding the desktop.

Byron Wait a minute. PowerTalk Manager is also an 'appe'. What’s different about it?

60 Shelley AOCE requires that PowerTalk Manager always run as a background application, while QuickDraw GX needs PrinterShare GX to run only when a document is printing.

Byron So, how does the system know to run PowerTalk Manager, but not to run PrinterShare GX?

55 Shelley PowerTalk Manager contains a resource of type 'appe', ID = 0. This resource returns true when called as a Pascal function, which tells the Startup Manager to launch it. PrinterShare GX has no such resource.

Byron I bet we could figure this out from the Process Manager source, but barring that, let's use MacsBug to figure out why the Finder fails.

Shelley We can stop on file opens using **atb openrf** to figure out when the Finder is accessing PrinterShare GX.

Byron But how do you get it to stop only for PrinterShare GX?

Shelley Well, I need to find the filename. I put a break on _Open; then I display the parameter block using **dm a0 iopb**.

Byron **iopb**?

Shelley I/O parameter block. It looks like this:

```
Displaying IOParamBlockRec at 0008D720
0008D720  qLink             NIL
0008D724  qType             0000
0008D726  ioTrap            A000
0008D728  ioCmdAddr         NIL
0008D72C  ioCompletion      NIL
0008D730  ioResult          0000
0008D732  ioNamePtr         0008A8D6 -> "PrinterShare GX"
0008D736  ioVRefNum         FFFF
0008D738  ioRefNum          0000
0008D73A  ioVersNum         #0
0008D73B  ioPermssn         #4
0008D73C  ioMisc            NIL
0008D740  ioBuffer          NIL
0008D744  ioReqCount        00000000
0008D748  ioActCount        00000000
0008D74C  ioPosMode         0000
0008D74E  ioPosOffset       00000000
```

Byron So the 18th byte into the block can be a pointer to a string; we can dereference that and look for strings that start with 'Prin'.

Shelley You C mutt. You have to think Pascal; the first byte will be the string length, 15, so you want to break when **@@(a0+12)=0F507269**.

50 Byron By George, this won't work. The file is already open!

Shelley How do you know?

Byron I used the MacsBug **file** dcmd, and there it is, near the bottom of the list.

```
Displaying File Control Blocks
fRef File         Vol      Type Fl Fork    LEof    Mark  FlNum Parent FCB at
0002 System       fat      zsys dW rsrc #2303194   #920 008359 007bfe 2fb352
0060              fat      •••• dw data #1032192     #0 000003 000000 2fb3b0
00be              fat      •••• dw data #3096576     #0 000004 000000 2fb40e
011c Apple Chanc… fat      FFIL dW rsrc #269497 #219985 008220 007c02 2fb46c
017a Chicago      fat      FFIL dW rsrc  #48064  #38423 008351 007c02 2fb4ca
. . .
```

```
0874 PowerTalk M… fat      appe dW rsrc #507556 #413184 007ca8 007bff 2fbbc4
08d2 PrinterShar… fat      appe dW rsrc  #32978    #708 008236 007bff 2fbb22
0930 Finder       fat      FNDR dW rsrc #456553 #362328 00835c 007bfe 2fbc80
098e Finder Pref… fat      pref dW rsrc  #19983    #328 007cde 007c95 2fbcde
09ec Desktop DB   fat      BTFL dW data #196608 #150528 000011 000002 2fbd3c
0a4a Desktop DF   fat      DTFL dW data #351810 #101184 000010 000002 2fbd9a
0bc2 QMgrCatalog  fat      BTFL dW data  #65536   #1024 007ce5 007ce4 2fbf12
0c20 WSBTree      fat      BTFL dW data  #65536   #1024 007ce7 007ce2 2fbf70
131a              Mail En… ···· dw data      #0      #0 000003 000000 2fc66a
#74 FCBs, #35 in use, #39 free
```

45 Shelley That could be OK. It depends on whether the Finder is opening a read-only path on the file. We can tell by looking at the parameter block as we did before and seeing what's in the ioPermssn field.

40 Byron As I suspected, it's a 3, meaning the Finder is opening it for reading and writing, when it only needs to read the bundle resource.

35 Shelley Well, not exactly, since it wants to mark the file as initialized, so that next time it won't add the file to the database again.

Byron But isn't that data in the Finder info, which is in neither the data fork nor the resource fork, but just part of the file identifier, like the filename?

30 Shelley Oh, yeah. I know! The standard call OpenResFile always tries for read/write permission. The Finder desktop-building code is old and tired, but I'm sure when they rewrite it they'll correctly use OpenRF instead. In any case, the Finder is getting an error it doesn't expect when opening the file, so it gives up building the desktop database before it has retrieved the bundle information from all of the applications.

25 Byron So one possible fix is to prevent the Startup Manager from opening the file in the first place, since QuickDraw GX doesn't need it to be open all the time.

20 Shelley Or we could cause the file to be opened as read-only by setting its shared bundle bit. There's a more obscure way to solve the problem: if we put the right stuff in the 'appe' resource, the Startup Manager will be instructed to close the file after executing some code.

15 Byron I happen to have a copy of ResEdit right here. It's a little difficult to work with my paws, so give me a minute. Hey — look at that little man going in and out of the jack-in-the-box. I could watch this all day!

Shelley Cool. It even changes all the colors in the system palette, causing all of the screens to redraw, and then redraw again when you quit. State of the art, man — I mean, dog.

10 Byron Well, the problem is now obvious. PrinterShare GX is a background application that doesn't need to run all the time, but it doesn't have the shared bit set, and it doesn't have an 'appe' resource. The 'appe' resource is code, and I don't have that nifty ResEdit code editor.

5 Shelley We can use MacsBug to write it for us. All we need are enough instructions to create a Pascal function that returns false. We can cheat by disassembling PtInRect and steal a little code from there. We can verify our code by using the MacsBug command **dh** to disassemble our hex. I think **422e 0004 4e74 0004** ought to do the trick.

Byron	What does that do?
Shelley	It causes an illegal instruction error on a 68000 machine. Good thing Apple doesn't sell those anymore.
Byron	But I just bought a luggable for a steal at the flea market! What can I do?
Shelley	Hit your smushed-faced little head against it. QuickDraw GX runs only on 68020 and better, so this code is just fine.
Byron	Arf.
Shelley	Grr.

SCORING

80–100	Excellent! You're good enough to write your own puzzles. Heck, you could probably write the whole *develop* magazine. (While you're at it, try to get some work out of those slackers at Catapult Entertainment and Rocket Science.)
55–75	Pretty good. You could speed up the blits in QuickDraw GX. (When you do, let us know.)
30–50	Not bad. You could make the PostScript driver go fast even without taking it native.
5–25	Try not to hurt yourself with shape operations.•

Thanks to Dave Hersey, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column, and to Ron Voss for tracking the problem down and fixing it in the shipping version of QuickDraw GX.•

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to AppleLink DEVELOP or to:

Caroline Rose
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)974-6395

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
One Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: JOHNSON.DK
Internet: dkj@apple.com
CompuServe: 75300,715
Fax: (408)974-6395

Please direct all subscription-related queries to *develop*, P.O. Box 531, Mount Morris, IL 61054-7858 or AppleLink DEV.SUBS (or, on the Internet, dev.subs@applelink.apple.com). Or call 1-800-877-5548 in the U.S., (815)734-1116 outside the U.S., or (815)734-1127 for fax.

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•