ColorSync 2.0 • Source Control With SourceServer • About Scripting Dictionaries

develop

The Apple Technical Journal

Issue 23 September 1995

Music the Easy Way: The QuickTime Music Architecture

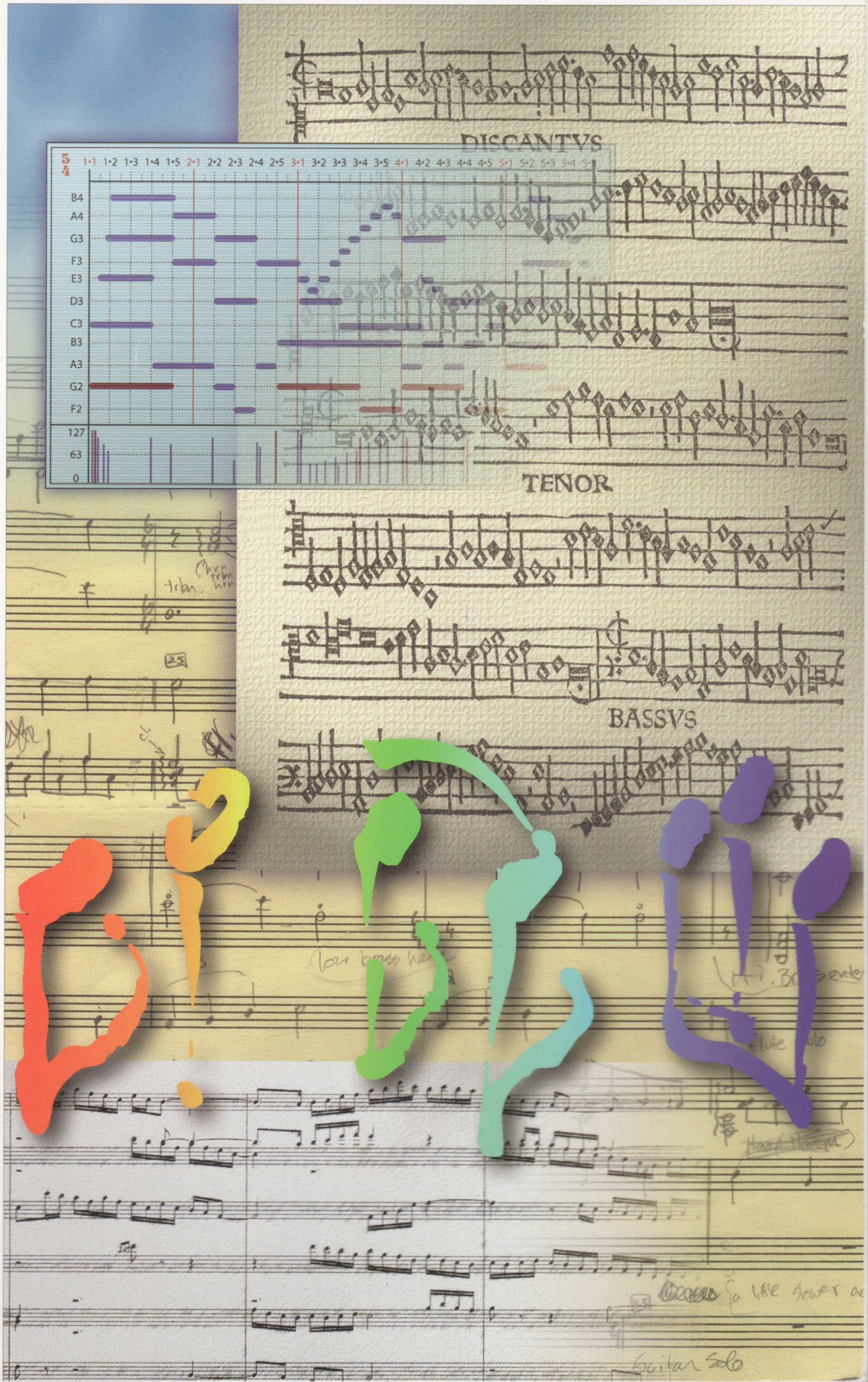
The Basics of QuickDraw 3D Geometries

Implementing Shared Internet Preferences With Internet Config

Multipane Dialogs

Document Synchronization

Power Macintosh: The Next Generation



$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Managing Editor *Toni Moccia*

Technical Buckstopper *Dave Johnson*

Bookmark CD Leader *Alex Dosher*

Able Assistant *Meredith Best*

Our Boss *Mark Bloomquist*

His Boss *Dennis Matthews*

Review Board *Brian Bechtel, Dave Radcliffe, Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Andy Shebanow, Nick Thompson, Gregg Williams*

Contributing Editors *Lorraine Anderson, Patria Brown, Steve Chernicoff, Toni Haskell, Judy Helfand, Cheryl Potter, Joan Stigliani*

Indexer *Marc Savage*

ART & PRODUCTION

Production *Lisa Ferdinandsen, Diane Wilcox*

Art Direction *Paul Luiso*

Technical Illustration *Mary Prusmack Ching, John Ryan, Laurie Wigham*

Formatting *Forbes Mill Press*

Photography *Sharon Beals*

Cover Illustration *Graham Metcalfe of Metcalfe/Shubert Design*

ISSN #1047-0735. © 1995 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, APDA, AppleLink, AppleScript, AppleShare, AppleTalk, ColorSync, EtherTalk, HyperCard, HyperTalk, LaserWriter, LocalTalk, Mac, MacApp, Macintosh, MacTCP, MPW, MultiFinder, Newton, NewtonMail, OpenDoc, PowerBook, Power Macintosh, QuickTime, and TrueType are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. A/ROSE, develop, Dylan, eWorld, Finder, NewtonScript, PowerTalk, QuickDraw, Sound Manager, and ToolServer are trademarks of Apple Computer, Inc. Adobe, Acrobat,and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. UNIX is a registered trademark of Novell, Inc. in the United States and other countries, licensed exclusively through X/Open Company, Ltd. All other trademarks are the property of their respective owners.

Printed on recycled paper

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. This CD contains a subset of the materials on the monthly *Developer CD Series*, 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 code is updated periodically, so always use the most recent CD.) The CD also contains Technical Notes, sample code, and other documentation and tools (these contents are subject to change). Items 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 in the Developer Services areas on AppleLink and eWorld and at ftp.info.apple.com. Selected articles are on the World Wide Web at http://www.apple.com, in the Developer Services area.

Macintosh Technical Notes. A designation like "(QT 4)" after a reference to a Macintosh Technical Note in *develop* 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 either AppleLink or eWorld addresses. We're currently in transition: a given AppleLink address may no longer work by the time this issue is published. If that happens, try the equivalent eWorld address. On the Internet, AppleLink address XXX translates to xxx@applelink.apple.com, eWorld addresss XXX to xxx@eworld.com, and NewtonMail address XXX to xxx@online.apple.com.

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., 1 Infinite Loop, 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 and back issues. You can subscribe to *develop* through APDA (see ordering information below) or use the subscription card in this issue. You can also order printed back issues from APDA. For all subscription changes or queries, contact APDA and *be sure to include your name, address, and account number as it appears on your mailing label.*

The one-year U.S. subscription price is $30 (for 4 issues and 4 *develop Bookmark* CDs), or U.S. $50 in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

APDA. To order products from APDA or receive the *Apple Developer Tools Catalog* of all the products available from APDA, 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 24 December 1995

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

I was visiting my friends Helen and John one night when Helen started telling me how excited about the World Wide Web John had become. He said, "Ask me anything at all, and I can find the answer for you." I asked what the new U.S. postal rate for international air mail was, knowing it had recently gone up from $.50. He delighted over finding a Web page for the Postal Service, and quickly found the rate: $.50. Wrong.

Later John showed me a spiffy magazine called *NewMedia*, and in it an article by longtime hypertext proponent Ted Nelson. Nelson expressed his joy that, with HTML and the Web, hypertext's time has finally come; we can now leave the insanity of "paper simulation" behind and write in a way that lets information take on its truer, interconnected form.

I found the article, and John's enthusiasm over the Web, a bit disconcerting. The Web is indeed a boon to humankind, but I don't see it entirely replacing what came before. The world's love affair with the Web reminds me of the early days of TV (so I'm told), when many people were sure that radio was dead. Out with the old, in with the new. But in fact the old still had its place in the world. The virtues of the Web don't mean we no longer need to get information from flesh and blood people sometimes, or from books and other media that we can hold in our hands. This may seem obvious, but from the near hysteria surrounding the Web these days, I'm not sure it is.

A few days after my visit with Helen and John, with John still smarting from his failed demonstration of the wonderfulness of the Web, Helen called and mentioned that she needed the lyrics to "House of the Rising Sun." I could hear John in the background, tapping away as he searched for them online. I said I'd use old technology and call back with them soon. The race was on.

After looking through my looseleaf binders full of song lyrics and a couple of big songbooks, I dug through my tapes and found an ancient recording of Woody Guthrie singing the song. After lots of rewinding and transcribing, I had more verses than Helen ever dreamed existed. When I triumphantly called back, John (several levels down in the Library of Congress) was mortified.

While old technology will typically not beat Web browsers in the search for nuggets of information, it will not die, and it deserves proper respect. There are some things we'll learn only through person-to-person contact. And there are emotions we'll experience only from hearing or reading good old-fashioned sequential deliveries. The World Wide Web is a valuable resource, but it is not, after all, the world.

CRose

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) enjoys editing *develop* so much that she fears she may forget to retire someday. She started out in technical writing and editing eons ago, eventually moving on to programming and even management before returning to her original calling. What seems to be calling her now is the sea: Her last vacation took her up to Puget Sound (stalking wild elk on the Olympic Peninsula on the way), and her next will be in a sailboat in the Bahamas. She may even have the opportunity to cruise the Pacific in a few years but she's not sure she'll be ready to leave *develop*, her cat, or terra firma.•

LETTERS

PROJECTDRAG IMPRESSES

I've been working with Tim Maroney's ProjectDrag (Issue 23, "MPW Tips and Tricks: Customizing Source Control With SourceServer"), and I'm very impressed. I've never found an adequate way of using a revision control system on the Macintosh (Projector is too clumsy to use when you're developing with CodeWarrior), and I had written off SourceServer completely after I had such a miserable experience with it under Symantec C++ 7.0. But Tim's article and software have given that dog some new tricks. His programs are easy to use and powerful at the same time.

Thank you very much for publishing Tim's work in this issue, and I hope to see more about ProjectDrag in the future.

— Phil Sulak

Thanks for the feedback; we're happy to know that you find ProjectDrag useful.

There were a few problems with the previous version of ProjectDrag, so on this issue's CD you'll find a new version with a few bug fixes and enhancements. Also, the previous version was missing the makefile; it's now on the CD.

— Caroline Rose

QTMA

I read the article on QTMA by David Van Brink in Issue 23, and have a few additional questions. As Director of Audio for Human Code (an Austin multimedia developer), I'm looking for a way to convey an other-worldly quality to the soundscape of a CD-ROM title we're developing.

First, is QTMA supported on the PC platform? If it only works on the Mac OS platform, I'm back to the drawing board.

Also, is it possible to seed a bank of custom-designed samples to be played using standard MIDI files with QTMA? If so, is there a developer's guide available for programming within QTMA?

— John Malcolm Smith

First of all, you've probably noticed that there were no changes to the Music Architecture in QuickTime 2.1 after all. These changes have been delayed until the next release of QuickTime (which should ship by early 1996). The code on this issue's CD has been revised so that it compiles with the 2.0 or 2.1 headers.

On the PC side, QuickTime music tracks are supported, but only inside movies. So, compose your score on the Macintosh, import it into a QuickTime movie using MoviePlayer, and then save it flattened with the "Playable On Non-Apple Computers" box checked. This movie will play through Windows' multimedia extensions, according to its MIDI setup.

As far as adding your own instruments, you should be able to do this in the next QuickTime release in two ways: by dropping a component into the System Folder, to make a sound library available to all applications, or by inserting a sound into the music track of a particular movie.

— David Van Brink

PUZZLE PAGE DOESN'T STINK

Re Lance Drake's letter in Issue 22 entitled "Puzzle Page Stinks": I *strongly*

IF YOU LIKE US, LET US KNOW

What do you like, or not like, about *develop* (besides the Puzzle Page)? We welcome your letters, especially regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK. Or you can write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014. 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).•

disagree. The Puzzle Page is the first article I read. From it I've learned new debugging tactics, and picked up cool MacsBug tricks and how to do more than just "G" from MicroBug. The fact that the "scoring" shouldn't be taken literally is obvious; after all, KON & BAL never get the answer till 10 or less. Don't let one humorless whiner ruin a good thing.

Keep up the good work; *develop* is a great resource.

— Steve Palmen

I wanted to let you know how much I enjoy the Puzzle Page. I just graduated and was lucky enough to land a job programming on the Mac. Issue 22 is my first and I've already looked back at all of the previous Puzzles because I enjoy reading about the deepest, darkest Mac knowledge that I hope to stuff into my brain one day. It's refreshing to have a technical journal that's not afraid to crack a joke every couple of pages. I haven't felt offended or mocked by your Puzzle Page.

— Matt Glazier

I just want to let you know that there are people out here who read and enjoy the Puzzle Page. I try to follow every twist and turn in the logic that leads to the final result. I've tracked down a few bugs in my own code that were complex and obscure enough to end up on the Puzzle Page, and it's nice to see the steps someone else follows.

— David Shayer

. . . WELL, MAYBE JUST A LITTLE

The letter from Lance Drake in Issue 22 about the Puzzle Page was, as you wrote, a surprise to you. To me it wasn't.

First I would like to state that the Puzzle Page is by far the best column in *develop* — technically very interesting and also amusing. This explains the good feedback you receive on it. Yet the "scoring" tables are indeed belittling, elitist, and intellectually arrogant. Even worse, they are offending. This is a detail, but it fully explains and justifies Mr. Drake's angry letter.

— Adriaan van Os

Many thanks for all the work you put into *develop*. The production qualities are superb. I have only one complaint: get rid of KON & BAL's Puzzle Page. I always feel depressed after reading it.

— Andrew Trevorrow

FINGER-CODED BINARY VARIATION

I'd like to comment on Tobias Engler's Finger-Coded Binary column in Issue 21. Although I agree with most of what he said, Tobias's approach, the 10-bit model, is far less natural than it needs to be. I find it much easier (at least more natural) to work with hands flat on the side of a table, using all fingers except thumbs — this results in the more commonly used 8-bit model. You can then use your thumbs for other things, such as branch prediction, status registers, or even complex instruction execution.

I have one advantage over many people. The fact that I'm missing part of my right thumb enables me to do fractions. No other digital system I know of can do 0, 1/2, and 1 digits.

— Martin-Gilles Lavoie

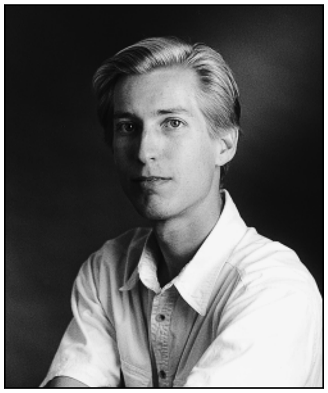
There's much more to the 10-bit model than you seem to realize. Have you ever had somebody tell you "You can eat as many Snickers bars as you can count on your hands"? Probably not. You wouldn't want to stop at 256, would you?

Concerning your fractional thumb: Your technological advantage over conventional digital systems will undoubtedly attract many copyists, which may result in a lot of unnecessary bloodshed. My advice to you is go and get a patent!

— Tobias Engler

Speeding Up whose Clause Resolution In Your Scriptable Application

The Object Support Library provides convenient mechanisms for scriptable applications to support complex expressions that may return multiple results (such as ***every item of container "b" whose name contains "a"****). However, the performance of applications that rely on the default behavior is nowhere near what it could be if the application took on some of the work itself. This article shows you how to gain ten- to a hundred-fold increases in the performance of* ***whose*** *clause resolution in your scriptable application. If your application is not yet scriptable, you'll find that the foundation classes presented in this article do most of the work required to support scripting.*



GREG ANDERSON

One of the greatest strengths of AppleScript is its built-in ability to do complex operations on groups of objects in a single line of script. For example, suppose you have a set of shapes in a scriptable drawing program, and you'd like to change the color of all the red shapes to green. In conventional programming languages, you'd need to write a loop that iterates over each object in the set, tests to see if its color is red, and then does a "set color to green" command for each red object that was found. Using AppleScript, you can do the same operation with the single statement **set color of every shape whose color is red to green**. In that statement, **every shape whose color is red** is called a ***whose** clause*, and it's the inclusion of **whose** clauses that makes AppleScript the powerful language it is.

You may at first doubt that using a **whose** clause is much better than writing the equivalent script with a loop. After all, the direction of modern processor design has been toward simplicity of the instruction set; RISC chips are able to gain incredible performance improvements by doing optimizations that aren't possible in CISC chips. Also, when all is said and done, the **whose** clause must finally execute the same loop-and-compare algorithm that you'd be forced to use if you wrote the script with the basic flow-of-control script commands, such as **do-while** and **if-then**.

Using a **whose** clause is, however, much more efficient than the alternative. AppleScript is based on the client/server paradigm: typically your script, the client, will be running in one application (usually the Script Editor or a script saved as a miniapplication), with the application being scripted acting as a server. In this

GREG ANDERSON is enjoying the hot days of late summer as he writes this, but by the time this issue is in your hands, he should be back on the ski slopes earning his nickname, "Air Bear." Greg spends most of his skiing time looking for some protrusion to jump or fall off of while wearing his favorite polar bear hat. He sometimes works, too; he recently moved to Japan to work on international software for Apple Technologies in Tokyo.•

situation, each script command that's directed at the scriptable application needs to be transferred between the two applications. A **whose** clause is a single script command, but with the loop approach many commands would need to be sent. Furthermore, AppleScript allows the scriptable application to reside on a different machine than the application running the script; if your script is running on a machine in Cupertino, California, and the server is on, say, Mars, reducing the number of round-trip messages would have a profound impact on the performance of the script. Remember, you can currently get only about 30 round-trip Apple events per second, so even if you aren't sending data to Mars, you'll still do a lot better with fewer events than with many.

There's another, similar reason that using **whose** clauses is superior to the equivalent loop-based script: AppleScript compiles scripts into byte codes that are interpreted during execution, whereas the individual script commands (once interpreted) are processed by a scriptable application typically written in a language that's compiled into machine code (be it 680x0 or PowerPC™). The loop-and-compare script will execute several lines of script for every item that's compared, whereas the **whose** clause is but a single line of script that triggers processing in a compiled application. It should be quite clear which will take less time to execute.

The Object Support Library (OSL) — the library that provides the API you use to make your application scriptable — enables your application to support **whose** clauses without requiring you to write a lot of additional code. You only need to provide an object-counting function and an object comparison function, and the OSL can resolve **whose** clauses for you. Since supporting **whose** clauses allows script writers to write more efficient scripts, you should always do at least this much. However, there are two other features of the OSL that can vastly increase the performance of scriptable applications but are often ignored by application writers: ***whose*** *clause resolution* (a way for your application to find the objects that match a **whose** test without using the OSL) and *marking* (a mechanism for efficiently handling collections of objects, such as those satisfying a **whose** clause). Using **whose** clause resolution, with the help of marking, will enable you to get the most out of your scriptable application. Resolving **whose** clauses can be a bit tricky, but with a little help from this article, you'll be on your way in no time.

If your application is not yet scriptable, you'll find the sample code included with this article (and on this issue's CD) to be invaluable in getting you up and running — particularly since it contains a lot of reusable code.

AN OVERVIEW OF THE OSL

Good descriptions of the OSL can be found in the *develop* articles "Apple Event Objects and You" in Issue 10 and "Better Apple Event Coding Through Objects" in Issue 12. If you need a quick review of the OSL and you don't feel like putting down this issue of *develop* to dig through your back issues, read on. If you can already generate tokens and resolve object specifiers in your sleep, by all means skip ahead to the next section.

When AppleScript is processing a script command such as **delete paragraph 2 of document "sample"**, it converts the command into an Apple event which it sends to the scriptable application that's referenced by the script. The Apple event's event class and message ID together specify the verb of the operation being performed — in this case **delete**. The object being operated on is passed in the keyDirectObject parameter of the Apple event, which is called, naturally enough, the *direct parameter* of the event.

The direct parameter is almost always an *object specifier* — a descriptor of type typeObjectSpecifier — although in some cases it may be something else. For example, in addition to object specifiers, the Scriptable Finder accepts alias records and file specifications in the direct parameter of events sent to it. If the direct parameter of an event is not of type typeObjectSpecifier, you're on your own to convert it into some format that's understood by your event handler. For descriptors that are of this type, though, all you need to do is call the function AEResolve, and the OSL will step in and help your application *resolve* the object specifier — that is, locate the Apple event objects it describes.

Object specifiers are resolved through *object accessor callbacks* that your application installs to allow the OSL to communicate with your application during object resolution. The accessor callbacks must take the description of the object requested by the OSL (for example, **document "sample"**) and return a *token* that describes the object in terms that the application can understand (for example, a pointer to a TDocument object). Tokens are passed back to the OSL in an AEDesc, a structure that contains a 32-bit descriptor type and a handle. Your application has complete control over what it stores in the token, as long as the AEDesc is valid (that is, it was created with AECreateDesc).

When the OSL calls your application's object accessor callbacks, it always passes either a token that represents the containing object (which it got from an earlier call to one of your object accessors) or a representation of the default container of the application, which is also called the null container of the application. So, to resolve the object specifier **paragraph 2 of document "sample"**, the OSL first asks for **document "sample"** from the null container. Then it asks the application to provide a token for **paragraph 2** from the token the application provided in response to the request for **document "sample"**. The token that the application provides for **paragraph 2** is returned as the result of the AEResolve call; the application will presumably use this token to process the Delete event.

Resolving object specifiers is explained in Chapter 6 of *Inside Macintosh: Interapplication Communication.* A figure illustrating the process of resolving object specifiers is on page 6-6.•

MARKING

Inside Macintosh: Interapplication Communication describes marking as a mechanism whereby items to be operated on are marked with some flag during resolution (that is, from the callbacks made by the AEResolve function); then, during execution, each marked item is processed and the mark is cleared. As described, marking doesn't sound very interesting and appears to be useful only in fringe cases.

Marking is actually very well suited for use as a general-purpose collection mechanism whenever the OSL needs to group tokens together to process an object resolution. For example, if the OSL is resolving the **whose** clause **every shape whose color is red** and there are multiple red shapes, the result of the call to AEResolve must be a collection of all the tokens that represent red objects. If your application supports marking, the OSL asks your application to create a special *mark token* to represent this collection. After your application provides the OSL with a mark token, the OSL will ask your application to add the tokens it provided for the red shapes to the mark token's collection. When AEResolve completes, the mark token is returned as the result of the resolution.

If your application doesn't support marking, the OSL will create collections of tokens for you by copying the data from your tokens into a descriptor list (an AEDescList).

It calls the standard Apple Event Manager routines for creating descriptor lists, which copy the data out of the data handle of the AEDesc and then store the token data somewhere inside the data handle of the descriptor list; the descriptor type of the AEDesc is similarly encapsulated.

Dealing with descriptor lists of tokens can be inconvenient, particularly if your application already supports collections of objects in some other way. The OSL marking mechanism gives you the flexibility to handle collections in any way that's convenient for your application.

To support marking, you must pass the flag kAEIDoMarking to AEResolve and implement the three marking callbacks that are passed to AESetObjectCallbacks: the create-mark-token callback (called just a "mark-token callback" in *Inside Macintosh*), the object-marking callback, and the mark-adjusting callback. The create-mark-token callback doesn't need to do anything more than create an empty mark token. The OSL will dispose of this token as usual by calling your token disposal callback when the token is no longer needed. Listing 1 shows an example implementation of a create-mark-token callback.

Listing 1. Create-mark-token callback

```
pascal OSErr CreateMark(AEDesc containerToken, DescType desiredClass,
      AEDesc* markTokenDesc)
{
   TMarkToken* markToken;

   markToken = new TMarkToken;
   markToken->IMarkToken();
   markTokenDesc->descriptorType = typeTokenObject;
   markTokenDesc->dataHandle = markToken;

   return noErr;
}
```

The object-marking callback is passed a mark token created from the create-mark-token callback and some other token created by one of your application's object accessor callbacks. Your object-marking callback should add a copy of the other token into the mark token (or apply a reference count to the token being added), because the OSL will dispose of the token added to your collection shortly after calling your object-marking callback. Listing 2 shows one implementation of an object-marking callback.

The mark-adjusting callback is called to remove ("unmark") tokens from the collection. Oddly enough, its parameters specify which tokens in the range to keep; all tokens outside the specified range should be discarded.

Implementing the marking callbacks is trivial. The only real work involved in supporting marking is handling collections of tokens when they're ultimately received by one of your event handlers (handling Move events, for example). The amount of code required to handle the marking callbacks and maintain your own collections is minimal; in fact, the time you'll save by not having to hassle with descriptor lists of tokens will more than make up for the implementation cost. You'll find more information on handling collections of tokens later in this article. Don't put off

Listing 2. Object-marking callback

```
pascal OSErr TAccessor::AddToMark(AEDesc tokenToAdd, AEDesc
      markTokenDesc, long markCount)
{
   AEDesc      copyOfToken;
   TMarkToken* markToken;

   // We know that the OSL will only give us mark tokens created with
   // our create-mark-token callback, but real code would do a test
   // before typecasting.
   markToken = (TMarkToken*) markTokenDesc.TokenObject();
   // Add a copy of the token to the collection, because the OSL will
   // dispose of tokenToAdd after passing it to you. A reference-
   // counting scheme is good here.
   copyOfToken = CloneToken(tokenToAdd);
   markToken->AddToCollection(copyOfToken);

   return noErr;
}
```

marking as an optimization to be done later; incorporate it into the design of your application from the very beginning.

For more details on the marking callbacks, see *Inside Macintosh: Interapplication Communication*, pages 6-53 to 6-54.•

WHOSE CLAUSE RESOLUTION

The only thing that a scriptable application needs to do to support **whose** clauses is provide an object-counting function and an object comparison function — the OSL will do the rest of the work. When the OSL does a **whose** clause resolution, however, it has no choice but to iterate over every element in the search set, repeatedly calling your application's object accessor, object comparison, and token disposal callbacks. Huge performance gains can be realized if you resolve **whose** clauses yourself, because you'll avoid the overhead the OSL requires to make these callbacks.

Passing the flag kAEIDoWhose to AEResolve tells the OSL that you'll resolve the **whose** clause yourself. The OSL calls your object accessor with the key form formWhose (see Listing 3). The key data is a **whose** descriptor — that is, an AERecord that describes the comparison to be performed in the search. Your application should interpret the **whose** descriptor and test every element of the container token to see if it matches the specified criteria. If the **whose** descriptor is too complex for your application, you can return the error code errAEEventNotHandled from your object accessor, and the OSL will do the resolution for you with the default techniques. This is very useful, as it allows you to maximize the performance of the most common **whose** clauses, yet still support complex **whose** descriptors that are likely to be encountered only rarely.

The astute reader will notice that the scheme presented in Listing 3 is very similar to the process that the OSL goes through to resolve **whose** clauses. There are still optimizations that could be made to speed up the resolution further, but we'll get to those later. To resolve **whose** clauses as shown in Listing 3, your application must be able to do the following:

Listing 3. Handling formWhose in the object accessor

```
pascal OSErr MyObjectAccessor(DescType desiredClass, AEDesc container,
      DescType /*containerClass*/, DescType keyForm, AEDesc keyData,
      AEDesc* resultToken, long /*hRefCon*/)
{
   switch (keyForm) {
      // case formAbsolutePosition, and so on
      ...
      case formWhose:
         // TWhoseDescriptor is a class that knows how to interpret
         // a whose descriptor and test tokens for membership in the
         // search set defined by the desired class and the whose
         // descriptor.
         TWhoseDescriptor whoseDesc(desiredClass, keyData);
         // TTokenIterator is a class that knows how to iterate
         // over the elements of a token.
         TTokenIterator iter(container);
         for (iter.Reset(); iter.More(); iter.Next()) {
            AEDesc token = iter.Current();
            if (whoseDesc.Compare(token) == kTokenIsInSearchSet) {
               // Add token to the collection stored in resultToken.
               AddTokenToResult(token, resultToken);
            }
         }
         break;
   }
   return noErr;
}
```

- Iterate over the elements of any token.
- Determine class membership of any token.
- Compare properties of the elements of any token.
- Convert a **whose** clause into some internal representation usable by your application.

The first two operations are required of any scriptable application, so yours probably can already do them. Comparing properties is something your application probably doesn't do yet, but in the worst case you could always write a few lines of code that call your property object accessor function, retrieve the data from the resulting property token, and then compare the descriptor that was returned. Obviously you can do better than this in terms of performance, and later on we'll investigate how. First, though, we'll look at how to interpret the contents of a **whose** descriptor.

THE CONTENTS OF A WHOSE DESCRIPTOR

Earlier I claimed that a **whose** descriptor was an AERecord, but I lied. A **whose** descriptor is actually a descriptor of type typeWhoseDescriptor. Internally, a **whose** descriptor is stored just like an AERecord, but you can't extract its parameters unless you first coerce it to type typeAERecord. In Apple events parlance, this type of descriptor is called a *coerced record*; its basic type is typeAERecord, and its coerced type is typeWhoseDescriptor.

The advantage of coerced records is that they allow clients of the Apple Event Manager (for example, the OSL) to define new descriptor types for AERecords that define the context in which the record will be used and specify (by convention) what parameters the client can expect to find inside it. The disadvantage is that it requires an extra memory allocation to coerce the descriptor back to typeAERecord before the parameters of the coerced record can be accessed. This is unfortunate, as one of the primary goals of performance optimization is to remove extraneous memory allocations; coercing the descriptor back to typeAERecord is part of the current design of the Apple Event Manager, though, so there's nothing we can do about it.

There are two parameters inside a descriptor of type typeWhoseDescriptor: keyAEIndex and keyAETest.

- The keyAEIndex parameter usually contains an enumeration whose value is kAEAll; this corresponds to the word **every** in the **whose** descriptor **every item whose name contains "e"**. The other possible values are kAEFirst, kAELast, kAEMiddle, and kAEAny for **whose** clauses that request the first, last, middle, or any (random) item. The keyAEIndex parameter might also be of type typeLongInteger or typeWhoseRange, to indicate a single item or a range of items, respectively.
- The keyAETest parameter contains another coerced AERecord whose type can be either typeCompDescriptor or typeLogicalDescriptor. In either case, you must coerce the descriptor to type typeAERecord to access the parameters inside it.

A comparison descriptor (typeCompDescriptor) contains three parameters: two objects to compare (keyAEObject1 and keyAEObject2) and a comparison operation to be performed on them (keyAECompOperator). Usually the first object to compare is a special type of object specifier that indicates a property to compare (for example, pName), and the second is a literal constant to compare it against (for example, **"e"**). The comparison operators include **contains**, **begins with**, **ends with**, **equal**, **not equal**, **greater than**, and a bunch of other relational operators. Because comparison descriptors can contain object specifiers (and usually do), they can become arbitrarily complex. You won't be able to resolve them all unless you reimplement the entire functionality of the OSL, at which point you might as well not call AEResolve either (thank goodness for errAEEventNotHandled, which allows you to fall back on the OSL if your application cannot parse a **whose** descriptor).

Fortunately, logical descriptors are much simpler than comparison descriptors. A logical descriptor contains two parameters: keyLogicalOperator and keyLogicalTerms. The logical operator indicates the Boolean logic to apply on the contents of the logical terms: **and**, **or**, or **not**. The logical terms descriptor is, as you may have guessed, a list of descriptors whose type is either typeCompDescriptor or typeLogicalDescriptor. Figure 1 shows the contents of a **whose** descriptor that corresponds to the script **every item whose name contains "e" and size is 0.**

The contents of whose descriptors are described in *Inside Macintosh: Interapplication Communication,* pages 6-42 to 6-45.•

PARSING WHOSE DESCRIPTORS

It may look like there can be a lot of different cases to handle in a **whose** descriptor, but it actually doesn't take too much code to convert a **whose** descriptor into a format that your application can understand. The next few listings show how this might be done. The code presented is somewhat simplified; it doesn't look at the keyAEIndex parameter of the **whose** descriptor (kAEAll is assumed), and it recognizes only very specific formats of comparison descriptors. Even this much of an effort is very useful,

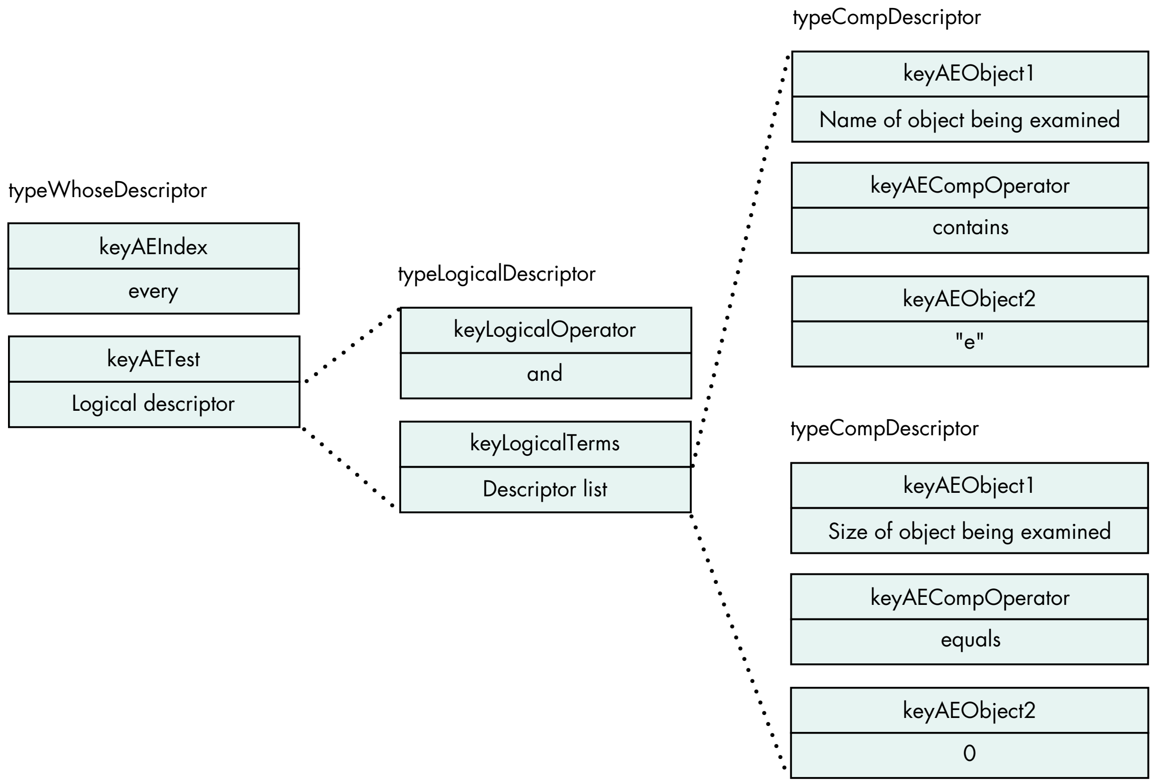


Figure 1. Contents of a **whose** descriptor

because it will cover about 90% of the **whose** clauses that your application is likely to encounter, and it's still possible to return errAEEventNotHandled and allow the OSL to take over for the rest. If you're expecting me to fall back on every *develop* author's favorite phrase, "This impossible task is left as an exercise for the reader," you're in for a surprise. The sample code on the CD will parse any valid **whose** descriptor passed to it and never falls back on the default handling provided in the OSL.

The top-level routine, ParseWhoseDescriptor, simply extracts the keyAETest parameter and passes it to ParseWhoseTest, returning the resulting search specification. These two routines are shown in Listing 4. (A *search specification* is an application-defined object that knows how to test tokens for membership in the search set defined by the **whose** descriptor; see the sample code on the CD for the implementation of the search specifications used in these listings.) ParseWhoseTest

Listing 4. Interpreting the contents of a **whose** descriptor

```
TAbstractSearchSpec* ParseWhoseDescriptor(TDescriptor whoseDescriptor)
{
   TAbstractSearchSpec* searchSpec = nil;
   TDescriptor          testDescriptor;

   whoseDescriptor.CoerceInPlace(typeAERecord);
   // Real code would call whoseDescriptor.GetDescriptor(keyAEIndex)
   // and at the very least check to see that its value is kAEAll,
   // and fail with errAEEventNotHandled if it isn't.
```

(continued on next page)

Listing 4. Interpreting the contents of a **whose** descriptor *(continued)*

```
   testDescriptor = whoseDescriptor.GetDescriptor(keyAETest);
   searchSpec = ParseWhoseTest(testDescriptor);
   testDescriptor.Dispose();
   return searchSpec;
}

TAbstractSearchSpec* ParseWhoseTest(TDescriptor whoseDesc)
{
   TAbstractSearchSpec* searchSpec = nil;

   switch (whoseDesc.DescriptorType()) {
      case typeLogicalDescriptor:
         TDescriptor logicalOpDesc, logicalTerms;
         DescType    logicalOp;

         whoseDesc.CoerceInPlace(typeAERecord);
         logicalOpDesc = whoseDesc.GetDescriptor(keyAELogicalOperator);
         logicalOp = logicalOpDesc.GetEnumeration();
         logicalTerms = whoseDesc.GetDescriptor(keyAELogicalTerms);
         searchSpec = this->ParseLogicalDescriptor(logicalOp,
                                   logicalTerms);
         logicalOpDesc.Dispose();
         logicalTerms.Dispose();
         break;

      case typeCompDescriptor:
         TDescriptor compOperatorDesc, obj1, obj2;
         DescType    compOp;

         whoseDesc.CoerceInPlace(typeAERecord);
         compOperatorDesc = whoseDesc.GetDescriptor(keyAECompOperator);
         compOp = compOperatorDesc.GetEnumeration();
         obj1 = whoseDesc.GetDescriptor(keyAEObject1);
         obj2 = whoseDesc.GetDescriptor(keyAEObject2);
         searchSpec = this->ParseComparisonOperator(compOp, obj1, obj2);
         compOperatorDesc.Dispose();
         obj1.Dispose();
         obj2.Dispose();
         break;
   }
   return searchSpec;
}
```

examines the type of the descriptor (either logical or comparison) and then extracts the appropriate parameters and passes them to either ParseLogicalDescriptor or ParseComparisonOperator, whichever is appropriate.

Since logical descriptor records can contain one or more terms, each of which is either a comparison or a logical descriptor record, ParseLogicalDescriptor calls back to ParseWhoseTest for each term in the record, creating a search specification for each (see Listing 5). If there's more than one term, ParseLogicalDescriptor compiles

Listing 5. Resolving logical descriptors

```
TAbstractSearchSpec* ParseLogicalDescriptor(DescType logicalOperator,
      TDescriptor logicalTerms)
{
   TAbstractSearchSpec* searchSpec = nil,
                        oneSpecification = nil;
   TDescriptor          oneTerm;
   TSearchSpecList*     specificationList = nil;

   FOREACHDESCRIPTOR(&logicalTerms, oneTerm) {
      oneSpecification = ParseWhoseTest(oneTerm);
      if (specificationList == nil) {
         if ((searchSpec == nil) && (logicalOperator != kAENot))
            searchSpec = oneSpecification;
         else {
            specificationList = new TSearchSpecList;
            if (searchSpec)
               specificationList->Add(searchSpec);
            specificationList->Add(oneSpecification);
            searchSpec = nil;
         }
      }
      else {
         if (oneSpecification != nil)
            specificationList->Add(oneSpecification);
      }
   }
   if (specificationList != nil)
      searchSpec = new TLogicalSpec(logicalOperator, specificationList);
   if (searchSpec == nil)
      FailErr(errAEEventNotHandled);
   return searchSpec;
}
```

the resulting search specifications into a list and returns that; otherwise, it returns a single search specification for the single term.

ParseComparisonOperator (Listing 6) first tests to make sure that the comparison operator is of the correct format. (Again, the code in this listing recognizes only a specific flavor of comparison operator; see the code on the CD for a more complete example.) If the operator passes that test, a new search specification representing the comparison is created and returned.

ABOUT THE SAMPLE APPLICATION

The code presented up to this point is the easy part: implementing the marking and **whose** callbacks, parsing **whose** descriptors, and creating search specifications can all be done with a small amount of isolated code. Doing a search on a set of elements or performing a complex operation on a collection of tokens is a bit more involved, though, and requires a well-integrated framework that supports these concepts uniformly. You're in luck — the sample application included on this issue's CD has such a framework.

Listing 6. Parsing comparison descriptors

```
TAbstractSearchSpec* ParseComparisonOperator(DescType comparisonOperator,
      TDescriptor& object1, TDescriptor& object2)
{
   TAbstractSearchSpec* searchSpec = nil;
   TDescriptor          desiredClassDesc, containerDesc,
                        keyFormDesc, keyData;

   if ((object1.DescriptorType() != typeObjectSpecifier) ||
       (object2.DescriptorType() == typeObjectSpecifier))
      FailErr(errAEEventNotHandled);

   object1.CoerceInPlace(typeAERecord);
   desiredClassDesc = object1.GetDescriptor(keyAEDesiredClass);
   containerDesc = object1.GetDescriptor(keyAEContainer);
   keyFormDesc = object1.GetDescriptor(keyAEKeyForm);
   keyData = object1.GetDescriptor(keyAEKeyData);
   if (containerDesc.DescriptorType() != typeObjectBeingExamined)
      FailErr(errAEEventNotHandled);
   if (keyFormDesc.GetEnumeration() == formPropertyID)
      searchSpec = new TGenericSearchSpec(keyData.GetDescType(),
                          comparisonOperator, object2);
   desiredClassDesc.Dispose();
   containerDesc.Dispose();
   keyFormDesc.Dispose();
   keyData.Dispose();
   return searchSpec;
}
```

The sample application is called Scriptable Database. As its name implies, it's a database that's fully scriptable; in fact, it's usable only through AppleScript — it has no user interface whatsoever. It's no coincidence that the model the database uses follows AppleScript's element containment model very closely. The Scriptable Database has documents that can be created, saved, and opened. Documents contain elements; elements have properties and data and may contain more elements. The database itself is completely generic; it doesn't care what the classes of the elements are or what properties they contain. To use it for a specific application, you'll have to edit Scriptable Database's dictionary, also called its *AppleScript terminology extension* ('aete' resource), to add the terms you'll need for your database.

AppleScript terminology extensions are described in *Inside Macintosh: Interapplication Communication*, pages 7-15 to 7-20 and Chapter 8.•

All the techniques described in this article are implemented in the source code of the Scriptable Database application — in particular, the application supports marking, and it resolves **whose** clauses itself (very quickly, I might add). It's an object-oriented application written in C++ based on a set of reusable foundation class libraries that you might find useful as a starting point in your own scriptable application. The source code is divided into the following subprojects:

- The Database subproject contains a standalone C++ object database. The code in this project is not discussed in this article, but you might find it interesting to peruse.

- The Base subproject contains pure C++ code that has no dependencies on any Mac OS or Toolbox routines, or any code from any other subproject in Scriptable Database.
- The Blue subproject contains C++ wrapper classes for Macintosh managers used by Scriptable Database.
- The Foundation subproject contains the foundation classes that Scriptable Database uses to implement scripting, and as such is the focal point of this article.
- The Scripting subproject contains the object accessors and event handlers needed to respond to the messages sent by AppleScript and the OSL.
- The Application subproject contains all the code that defines the Scriptable Database application; in fact, all the code specific to Scriptable Database is in this subproject. Every other subproject is also used in some other application that I've worked on.

Note that these subprojects are layered such that each one uses code found only within that subproject or in a more primitive subproject. The Database subproject is used only by the Scripting and Application subprojects; all other subprojects are used freely by any subproject listed below it. The foundation classes will be discussed in depth in this article; comprehension of the rest of the sample code is left as an exercise for the reader. (You didn't think I could write an entire article and not say that at least once, did you?)

ABOUT THE FOUNDATION CLASSES

The focal point of the foundation classes is the class TAbstractScriptableObject. This class was designed to serve as a base class, but it may also be mixed into an existing class hierarchy with multiple inheritance, as was done in the sample application (see the class TScriptableDocument). Any object derived from TAbstractScriptableObject can be used as a token for the OSL. Memory management of tokens must be done carefully; note that in most instances, tokens passed to the OSL are temporary and must be deleted when the token disposal callback is called. In other instances, though, it may be more convenient to use an existing object that the application has already created — for example, a document object.

Because of this, the token disposal callback must be able to unambiguously determine the difference between the temporary objects and those objects it should not delete, or disaster will result. *Designators* — objects that represent some portion of another object — are used for the temporary objects. The class TAbstractScriptableObject defines the methods CloneDesignator and DisposeDesignator, which do nothing in the abstract case. Designators override these methods to copy and dispose of themselves — sometimes in conjunction with a reference-counting scheme.

As you might expect, the methods of TAbstractScriptableObject are designed to provide functionality that closely matches the features of the OSL. All objects derived from this class have elements and properties and can be sent events generated from an Apple event that the application receives. There are virtual methods in TAbstractScriptableObject that you can override to provide each of these types of behavior in your objects.

ELEMENTS OF A SCRIPTABLE OBJECT

A scriptable object exports its elements by providing an iterator object that knows how to iterate over the appropriate set of objects. There are two methods that return iterators, ElementIterator and DirectObjectIterator.

```
virtual TAbstractScriptableObject* ParentObject();
virtual TAbstractObjectIterator* ElementIterator();
virtual TAbstractObjectIterator* DirectObjectIterator();
```

The ParentObject method returns the object that this object is an element of. The element iterator iterates over the elements of the object, as was previously mentioned; the direct object iterator usually returns an iterator that knows about a single object — the TAbstractScriptableObject that created it. If the object is actually a collection, however, its direct object iterator will iterate over every element in the collection. Once your application provides an iterator for the elements of its objects, the code in the foundation classes can handle most of the standard access methods for you. The access methods supported include formAbsolutePosition and formName, the default ordinals (all, first, last, and so on), and ranges of items (for example, items 1 through 10).

Your application's scriptable classes can support more specialized access methods by overriding the appropriate method:

```
virtual TAbstractScriptableObject* Access(DescType desiredClass, DescType
      keyForm, TDescriptor keyData);
virtual TAbstractScriptableObject* AccessByUniqueID(DescType desiredClass,
      TDescriptor uniqueID);
virtual TAbstractScriptableObject* AccessByOrdinal(DescType desiredClass,
      DescType ordinal);
```

The first method, Access, is the general object-accessor dispatch method that calls the more specific access method appropriate for the keyForm parameter. You can override this method to define custom access forms — for example, the Scriptable Finder defined the forms formCreator (to access an application by its creator type) and formAlias (to access a file or folder through an alias record). The method AccessByUniqueID provides a mapping from a unique ID to an object; override this method if your objects have unique IDs that scripts can use to access them. The method AccessByOrdinal handles ordinal access. All ordinals defined in the Apple Event Registry are supported by the implementation in the base class, so your application will probably never need to override AccessByOrdinal.

PROPERTIES OF A SCRIPTABLE OBJECT

Every scriptable object has at least a few properties that it must support. Almost all classes will have these five properties:

- pName, since most objects have names
- pClass, pBestType, and pDefaultType, since the Apple Event Registry requires that all objects support these properties
- pContents, since the foundation classes handle Get Data and Set Data events by using this property

To advertise the existence of a property, your scriptable classes can override the methods BestType, DefaultType, and CanReturnDataOfType; these methods are used by the Get Data event handler to determine what data type it should ask for when it requests the property data from the object through GetProperty.

```
virtual DescType BestType(DescType propertyName);
virtual DescType DefaultType(DescType propertyName);
virtual Boolean CanReturnDataOfType(DescType propertyName,
      DescType desiredType);
```

However, your application doesn't have to override these methods to provide information about every property of an object, since it's also possible (and more convenient) to describe the properties of an object in a property description table. For example, the properties defined in TAbstractScriptableObject are shown in the following property description table:

```
TPropertyDescription TAbstractScriptableObject::fPropertiesOfClass[] = {
   {  pName,        kReserved,     typeChar,         typeChar },
   {  pClass,       kReserved,     typeType,         typeType },
   {  pDefaultType, kReserved,     typeType,         typeType },
   {  pBestType,    kReserved,     typeType,         typeType },
   {  pID,          kReserved,     typeLongInteger,  typeLongInteger },
   {  pIndex,       kReserved,     typeLongInteger,  typeLongInteger }
};
```

Each entry in this table consists of four long words: the property identifier, a long word reserved for use by the class that defines the property, the property's best type, and the property's default type. The property description table is referenced through the class data table, so properties defined in one class are automatically inherited by any class that derives from it. The methods BestType and DefaultType return information from the property description table if an entry for the requested property can be found, and the method CanReturnDataOfType returns true if the desired type is the best type or the default type for a property.

See the files Object.cp and Object.h in the sample code for information on the class data tables. The macros DeclareMinClassData and ImplementMinClassData are used for classes that have no class properties; classes that do have class properties use the macros DeclareClassData and ImplementClassData.•

The reserved long word from the property description table is always passed to the GetProperty and SetProperty methods; it can be used to provide information to assist in obtaining the data for the requested property.

```
virtual TDescriptor GetProperty(DescType propertyName, DescType desiredType,
      unsigned long additionalInfo);
virtual void SetProperty(TTransaction* transaction, DescType propertyName,
      TDescriptor& data, unsigned long additionalInfo);
```

The reserved long word can have nearly any value, but should not be greater than or equal to the constant kReservedRangeForPropertyInfo (see AbstractScriptableObject.h).

In addition to making the application's properties easier to implement, the property description table is key in supporting the "properties" property (which returns the current value of all the properties of an object, as specified by the property description table). It's also very useful for accessing properties of collections of tokens, as described later.

The transaction parameter in the SetProperty method must be provided by the caller but is not used by the foundation classes. It's provided as a mechanism whereby transaction-based applications (such as Scriptable Database) can make all changes under the auspices of a transaction object. Once all changes are made successfully, the transaction changes are committed back into the database. If anything goes wrong, the transaction is aborted and all changes are backed out. To the foundation class, TTransaction is just a named object that has no methods. The event handlers in the Scripting subproject use code from the Database subproject to create a transaction to pass to SetProperty (and other methods that can change the contents of the database), and commit or back out of the changes as appropriate after the event completes successfully or fails.

In some rare cases, it may be undesirable to include a property in the property description table, or it may be inconvenient to implement all of the functionality of a property strictly through the GetProperty and SetProperty methods. For example, Scriptable Finder has a trash property that returns a reference to the Trash object on the desktop. In such cases, your application should override the method AccessByProperty to return an appropriate scriptable object that represents the property:

```
virtual TAbstractScriptableObject* AccessByProperty(DescType
      propertyIdentifier);
```

The object returned by AccessByProperty can be any sort of scriptable object; unlike properties described solely by the property description table, it can have properties above and beyond the minimum (for example, pClass, pBestType, and pDefaultType), and it can receive events (such as Empty Trash). Properties that are returned through AccessByProperty can also appear in the property description table, but if they do, the reserved long word should contain the magic constant kNeverCreateGenericProperty.

SENDING EVENTS TO A SCRIPTABLE OBJECT

Most scriptable applications use one of two dispatch techniques for handling Apple events: event-first or object-first dispatching. In event-first dispatching, an event is first dispatched to an event handler, which resolves the direct parameter and passes it a message appropriate to the Apple event being received. The advantage of event-first dispatching is that the parameters of the event are well known and can be extracted and passed to the object from the event handler, reducing the amount of duplicate code scattered through the various object event handlers. The disadvantage is that event-first dispatching requires a large number of very similar event handlers, and the message-passing API is often large (one method per event).

Object-first dispatching attempts to solve this problem by providing a single event handler that blindly resolves the direct parameter of the received Apple event and passes the event to the resulting object. This technique is much simpler than event-first dispatching, requires a smaller API, and usually does exactly the right thing. But object-first dispatching doesn't *always* do exactly the right thing. For example, an Apple event that copies a set of objects to some destination container would send a different Copy event to every item in the source; what you might prefer is to have a single Copy event sent to the destination object, with the list of items to copy included as a parameter to the event. You'd never get the latter with object-first dispatching.

The Scriptable Database application uses a combination of event-first and object-first dispatching. Most Apple events are processed by a common event handler that resolves the direct parameter and passes the message along, in object-first dispatching style. Certain special events, however, such as Move, Copy, and Create Element, are processed in their own event handler, which can send a message to some object other than the direct parameter of the Apple event. The two primary methods that events are sent to are AECommand and CreateNewElement.

AECommand is defined as follows:

```
virtual TDescriptor AECommand(TTransaction* transaction, TAEvent ae,
      TAEvent reply, long aeCommandID, TAbstractScriptableObject*
      auxObjects = nil, long auxInfo = 0);
```

Both the Apple event message and the reply are passed to the event handler, just in case they need to be accessed directly. The AECommand method should not put the command result into the reply directly, though, as it might not be the only object that's receiving this message. Instead, it should return the result as the return value of the method, and allow the event handler to collect all the results into a descriptor list and package them in the reply.

The meaning of the parameters auxObjects and auxInfo depends on the event handler that's processing the message; the aeCommandID parameter implicitly defines what the AECommand method should expect to find in these parameters. For example, in the Move and Copy events, the auxObjects parameter contains the set of objects that should be moved or copied. Providing a single method with general-purpose, multiple-definition parameters allows different scriptable applications that use the same foundation classes to define new events that have custom parameters without requiring them to change or expand the API of the foundation classes. This is one of the advantages of object-first dispatching that we definitely want to keep in our design.

The Create Element event is special enough to warrant giving it its own dispatch message:

```
virtual TAbstractScriptableObject* CreateNewElement(TTransaction*
      transaction, TAEvent ae, TAEvent reply, DescType newObjectClass,
      TDescriptor initialData, TDescriptor initialProperties, Boolean&
      usedInitialData, Boolean& usedInitialProperties);
```

In most cases, classes that override CreateNewElement only need to look at the newObjectClass parameter, create a new object of that class, and return a reference to the newly created object. The event handler calls the SetData method of the new object by using the **with data** parameter from the Create event, and then calls the SetProperty method of the new object with each of the properties specified in the **with properties** parameter from the Create event. The initial data and initial properties for the new element are also provided as parameters to CreateNewElement in case they're needed at create time. If the usedInitialData or usedInitialProperties parameter is set to true, the event handler is inhibited from calling SetData or SetProperty, respectively, on the new object.

TOKEN COLLECTIONS

As previously mentioned, objects derived from TAbstractScriptableObject can be grouped into collections of tokens that can be passed around as a single object. The class that implements most of this functionality is TProxyToken, which is publicly derived from TAbstractDesignator. (A *collection object* is a temporary object created only to manage the collection of tokens and must be disposed of when the collection is no longer needed; therefore, a proxy must be a designator.) There are a number of different types of collections, each derived from the class TProxyToken.

The classes of proxies provided in the foundation classes include the following:

- TEveryItemProxy — every element of an object
- TEntireContents — every item in the entire deep hierarchy
- TMarkToken — a collection of tokens accumulated from the marking callbacks or from resolving a **whose** clause

Other types of collections are also possible. For example, the selection token is a proxy for the set of items that are currently selected, so the token for the selection would also derive from TProxyToken (however, since the Scriptable Database has no user interface, it has no selection object).

Sending a message to a proxy token usually does nothing more than pass the message on to each of its delegates; for example, the **open selection** script would pass an Open event to every selected item. In other cases, however, the proxy token handles the event itself. For instance, **set selection to item 1** doesn't send a Set Data event to the selected items; instead, it deselects the currently selected items and selects the items in the direct parameter (such as **item 1** in the previous example). The exact behavior of the proxy is determined by the concrete class (for example, TEveryItemProxy) that derives from the abstract class TProxyToken, but the proxy token does provide some mechanisms that can be used by its descendants to control the meaning of certain messages.

Properties in particular are handled in a special way by proxies. Some properties will apply to the proxy object itself, whereas other properties will refer to the delegates of the proxy token. For example, the script **default type of selection** should return the default data type for the selection object (which would be of type typeAEList), whereas **default type of every item whose name contains "e"** should return a list of default types, one for each item that matches the query **every item whose name contains "e"**. There is no heuristic that can be used to determine which properties should apply to the proxy and which should apply to the proxy's delegates; the only solution is to list all the properties that should be sent to the proxy object in some way. In the foundation classes, this is done with the method PropertyAppliesToProxy:

```
Boolean PropertyAppliesToProxy(DescType propertyName);
```

Each class that derives from TProxyToken should override PropertyAppliesToProxy and return true for those properties that should be processed by the proxy object and false for those that should be sent to the proxy's delegates.

MORE ON SEARCH SPECIFICATIONS

Previous sections of this article described how a **whose** clause was received by the object accessors of an application, converted into a search specification, and then resolved with a simple element iterator. Now that you're familiar with the capabilities of the foundation classes, we can go into the workings of the search specifications in a little more detail.

As you may recall, there are two types of search specification: logical and comparative. The primary operation of a search specification is to take a token and return whether or not that item is a member of the set specified by the comparator. A logical specification contains a list of other specifications; it does nothing more than call the comparator method of each, and either logically AND or logically OR the results together. A comparative search specification needs to perform some test on a property of an object that was passed to it; it does so by calling the CompareProperty method of the object being tested.

```
virtual Boolean CompareProperty(DescType propertyIdentifier, DescType
      comparisonOperator, TDescriptor compareWith);
```

The property identifier, the comparison operator, and the literal data to compare with were all extracted from the **whose** descriptor, as described previously. The default implementation of CompareProperty calls the object's GetProperty method and compares the result with the literal data by using the specified comparison operator. (You'll find a routine that compares two descriptors in the file MoreAEM in the Blue subproject of the sample application.) Note, however, that calling GetProperty involves a memory allocation to create a descriptor for holding the property data.

Memory allocations are something best avoided in the inner loop of an operation that's supposed to progress quickly, so the performance of a **whose** clause resolution can be improved if you override CompareProperty and do common property comparisons without a memory allocation.

FURTHER OPTIMIZATIONS

Using the techniques described up to this point, your application can resolve **whose** clauses, and do so much faster than the OSL would. However, there are other optimizations that you can make to further improve performance.

The techniques described so far perform better than the OSL for two primary reasons:

- They limit the number of memory allocations needed, as much as possible.
- They reduce the number of callbacks that need to be made between the OSL and your application. This is particularly important if your application is PowerPC native but uses the emulated 680x0 OSL.

Also, note that if your implementation of access by index is O(N) rather than constant time, the OSL's **whose** clause resolution will be O(N^2), since it will have to call your O(N) access by index callback N times. Even if you ignore this article completely and don't resolve **whose** clauses yourself, you should as an absolute minimum cache the last token returned by your formAbsolutePosition accessor and ensure that the next call to the accessor can be done in constant time if the container token and desired class are the same and the index is 1 greater. This will speed up your **whose** clause resolution considerably.

However, even for all of the performance gains that these techniques provide, **whose** clauses are still resolved according to the same basic algorithm used by the OSL. As anyone who has dabbled in computer information-science theory knows, it's often more advantageous to switch algorithms completely and put off fine-tuning until after the correct algorithm has been found.

Unfortunately, it's not possible to do any better than what we've already done in the general case (a direct linear search of the search space, comparing every item to the search specification in order). Doing a binary search isn't possible unless your search space happens to be sorted by your search key — not very likely, and in any event it's impossible to know whether it is or not *unless you have specific knowledge about the search space*. Searching the entire contents of a deep hierarchy — such as all the folders on a disk — is one type of search space that can often be optimized.

In cases where the search space is well known, it's often possible to abandon the idea of direct iteration and use some other algorithm to search. For example, if you're writing code to search the entire contents of a disk, you would be much better off calling PBCatSearch, which walks through the entries in the catalog record in the order they happen to appear on the disk, ignoring the disk's hierarchy. This technique is so much faster than doing a deep traversal of the disk's catalog that doing a deep search of some subfolder on a disk is usually much better accomplished by searching the entire disk and weeding out the matches that aren't somewhere inside the search's root container. In cases where you have access to a search engine with characteristics similar to PBCatSearch, you should go out of your way to try to use it. Of course, this may well require yet another conversion of the search specification, but the performance gains will outweigh the initial cost. The foundation classes presented in this article have hooks that allow the incorporation of existing search engines to be incorporated into the process of resolving **whose** clauses.

When a **whose** clause is being resolved, the task of doing the search is delegated to the iterator object returned by the root of the search. Putting the method in the iterator rather than in the object allows different types of iterators to provide different search algorithms, each optimized to its own search space. The iterator returned by the TEntireContents proxy has a special implementation of AccessBySearchSpec; instead of using the implementation it inherits from TAbstractIterator, it uses a method called SearchDeep in the element iterator of the root object. The default implementation of SearchDeep does nothing more than compare every item in the deep hierarchy below each of its elements, and add those that match to the collection. This is really no different from what would happen if TEntireContents::AccessBySearchSpec just called through to Inherited::AccessBySearchSpec, but it does provide a hook enabling special iterators to insert their own search engines if they have a technique that will do deep searches faster than a straightforward deep iteration.

Listing 7 shows the default implementation of SearchDeep; note that it does a deep search on each of the elements of the iterator rather than simply a single deep search. The reason for this is that iterators aren't required to have a single root object that one could conceivably search deep from; once you have an iterator, the only knowledge at your disposal is the set of objects that the iterator "contains." The information as to where the iterator came from isn't available to every iterator, although some (such as TDeepIterator) do save a reference to it.

Listing 7. Doing a deep search

```
TAbstractScriptableObject* TDeepIterator::AccessBySearchSpec(DescType
      desiredClass, TAbstractSearchSpec* searchSpec)
{
   TObjectCollector  collector;

   TAbstractObjectIterator* iter = fRootItem->ElementIterator();
   iter->SearchDeep(&collector, desiredClass, searchSpec);
   iter->Release();
   collector.CollectorRequest(kWaitForAsyncSearchesToComplete);
   return collector.CollectionResult();
}

void TAbstractObjectIterator::SearchDeep(TAbstractCollector* collector,
      DescType desiredClass, TAbstractSearchSpec* searchSpec)
{
   TDeepIterator deepIter(nil);
   for (this->Reset(); this->More(); this->Next()) {
      TAbstractScriptableObject* elementToDeepSearch = this->Current();
      deepIter.FocusOnNewRoot(elementToDeepSearch);
      for (deepIter.Reset(); deepIter.More(); deepIter.Next()) {
         TAbstractScriptableObject* token = deepIter.Current();
         if (token->DerivedFromOSLClass(desiredClass) &&
               searchSpec->Compare(token))
            collector->AddToCollection(token);
         else
            token->DisposeDesignator();
         token = nil;
      }
```

(continued on next page)

Listing 7. Doing a deep search *(continued)*

```
      if (elementToDeepSearch->DerivedFromOSLClass(desiredClass) &&
            searchSpec->Compare(elementToDeepSearch))
         collector->AddToCollection(elementToDeepSearch);
      else
         elementToDeepSearch->DisposeDesignator();
   }
}
```

In Listing 7, rather than having the deep search iterator create and return a collection of tokens, a collector object is passed in and given the responsibility of making a collection from the results of the search, which it's passed one item at a time. This is done so that other parts of your scriptable application can call SearchDeep to do deep searches if they need to, and providing a collector object allows this code the flexibility to process the search results one item at a time, as they are found, rather than waiting for the entire search to complete.

Note the following line in Listing 7:

```
collector.CollectorRequest(kWaitForAsyncSearchesToComplete);
```

A search engine that's hooked into this code path might, in a multithreaded application, execute asynchronously under its own thread. In these instances, the search engine needs a way to tell the collector that it's still running, and might call collector->AddToCollection with more search results at any time. The search engine does this by attaching a dynamic behavior object to the collection that understands the kWaitForAsyncSearchesToComplete message (see "What Is a Dynamic Behavior?"). When this message is received, the search engine's collector behavior must block the current thread of execution until the search engine completes its search.

The use of a collector object and a dynamic behavior object allows the searching code to be flexible, optimized independently of other search engines, and reusable, even to other code that might not have exactly the same needs as the scripting code.

Also note the implementation of the functions TEveryItem::SearchDeep and TMarkToken::SearchDeep. Both of these call the function RecursiveSearchDeep, which calls SearchDeep on each of the elements of the iterator in turn. Without this special code path, a script such as **(entire contents of every disk) whose name contains "mac"** would end up using the slow deep-iteration search, and miss out on the optimized SearchDeep method of each disk. Calling the SearchDeep method of each disk independently enables different types of disks to have different types of search engines; for example, searches of remote disks might be optimized differently than searches of local disks, and not every type of volume supports PBCatSearch. In a framework that has provisions for optimizations, flexibility of design is extremely important.

WHAT WAS THIS ARTICLE ABOUT, ANYWAY?

It doesn't take too much work to vastly improve the performance of your scriptable application, and the techniques presented in this article will help you do just that. Resolving **whose** clauses yourself can speed up the execution of your event processing by a factor of ten to a hundred; a chance to gain that level of improvement is hard to ignore.

WHAT IS A DYNAMIC BEHAVIOR?

A dynamic behavior is an object that can be attached to some other object to change its behavior dynamically at run time. Only objects that are specially written to accept behaviors can have behaviors attached to them, and only certain methods of that object can be dynamically changed by the behavior object.

Methods that support dynamic behaviors contain additional code that first dispatches to any behavior attached to the object and then does the default action for that method. But the actual flow of control is somewhat different from that.

Suppose you have an abstract class TObject that supports behaviors, and an abstract class TBehavior that provides an interface for an object that can dynamically change the behavior of any TObject-derived object. If TObject has a method called Command that the behavior could modify, the implementation of TObject::Command would look like this:

```
TObject::Command()
{
   TBehavior*  behavior;
   behavior = this->FirstBehavior();
   if (behavior)
      behavior->CommandDynamicBehavior();
   else
      this->CommandDefaultBehavior();
}

TBehavior::CommandDynamicBehavior()
{
   TBehavior*  behavior;
   behavior = this->NextBehavior();
   if (behavior)
      behavior->CommandDynamicBehavior();
   else
      this->Owner()->CommandDefaultBehavior();
}
```

Given this definition for the Command method, some class derived from TBehavior could override the virtual method TBehavior::CommandDynamicBehavior, and call Inherited to execute the default action of the method it's overriding. This allows behaviors to do both pre- and post-processing. The cost to supporting behaviors is additional dispatch time, but the advantage is the powerful, dynamic extensibility of your objects.

AppleScript is one of the most compelling technologies that Apple offers — the ability to record scripts, modify them, and play them back later puts powerful automation into the hands of programming novices. However, AppleScript is only as cool as the scriptable applications available in the marketplace. If you've written a scriptable application, thank you. If you haven't yet taken the OSL plunge, by all means read some of the material referred to in this article and dive in. (You might also want to take a look at the "According to Script" column that follows this article.) In either case, you should find the sample code on this issue's CD to be a very useful aid in implementing fast and complete scripting support in your Macintosh application.

RELATED READING

- *Inside Macintosh: Interapplication Communication* (Addison-Wesley, 1993).
- "Apple Event Objects and You" by Richard Clark, *develop* Issue 10, and "Better Apple Event Coding Through Objects" by Eric M. Berdahl, *develop* Issue 12. These articles provide good descriptions of the OSL.

Thanks to our technical reviewers Dan Clifford, Eric House, Arnoldo Miranda, and Jon Pugh.•

CAL SIMONE

ACCORDING TO SCRIPT

Steps to Scriptability

To wind up my first year of writing about scripting in *develop*, this time I'll solidify the sequence of steps involved in making an application scriptable. A few of these steps have been mentioned before, while some material is new; here all the steps are organized so that you can work out a strategy for implementing scriptability. You may be surprised at what you'll find.

THE WRONG WAY

In the past, a programmer who was responsible for implementing Apple events support in a scriptable application usually set about this task in one of two ways:

- writing the code for the event handlers and object accessor functions first, then, just before shipping, deciding what to call things and throwing together a dictionary at the last minute
- jumping into the design of an object model hierarchy (in an attempt to implement the Core suite), then writing the event handlers and object accessor functions, and, again, putting together the dictionary last

These methods were fine back in the days when Apple events were used principally for direct communication between two applications — one program was usually the client of the other. But in today's world of scripting, it is users who are the clients. So in order to accomplish the goal of creating a human-friendly scripting vocabulary, developers need different methods for development.

THE NEW, BETTER WAY

Since your scripting interface is also a user interface to your application, it should be as full and rich as the graphical interface, and should be as intuitive as you can make it. In creating human-oriented scriptability, your goal is to make it as natural and as easy as possible for users to write sentences to communicate with and control your application. You want users to be able to write sentences that are as close as possible to the way they might think about what they want to do. Prepare to open up the full functionality of your application through scripting — you'll want to make it complete.

The following plan will help you develop a clean vocabulary that allows users to easily work with your application.

PRACTICE YOUR WRITING

The first set of steps will help you home in on the terms you'll use in your vocabulary.

Write down sentences. The very first thing to do is to write down as many sentences as possible describing actions that can be accomplished with your application. At this stage, don't try to make real scripting commands; just write down basic ideas. For example:

```
play movies
grab the customer's profile
print pages 2 through 5
translate this book from English to French
send this message to Bob at the Redmond office
find all the records containing "University"
delete all paragraphs containing the word
   "Windows"
```

Have users write sentences. Users think differently about the way they accomplish things with applications than programmers do. Invite users of your application to write down some general sentences. Encourage them to think about how they want to accomplish what they do. Ask them to write the sentences as if they were directing the computer by speaking to it. (You can do this simultaneously with the above step.)

Include users who are experienced with earlier versions of your application. These users don't need AppleScript experience. Consider inviting your documentation writers and your support people to participate. You'll see quickly how users think about your application from a task-oriented perspective.

Don't attempt to write code yet or design your object hierarchy around what users write. Just use this to help

CAL SIMONE (AppleLink MAIN.EVENT) wants *your* dictionary for the Webster database. He will be analyzing the terms in your vocabulary against others in search of similarities and differences. Send your 'aete' resources to him on AppleLink or at mainevent@his.com on the Internet.•

you think in broad terms about how something might be accomplished.

Write some commands. Write more sentences, this time attempting to make script commands. Try to fit them into the context of a possible scripting vocabulary. This is an iterative process, through which you can distill your broad ideas into useful terms.

When writing commands, keep one eye open for consistency — think a bit about existing AppleScript commands and objects. At this juncture, it may help to have some people with AppleScript experience write sample sentences to describe how they want to control your application.

The sentences should begin to take on the flavor of AppleScript statements, with verbs followed by objects. For instance:

```
tell "emailer" to send the file "Weekly Report"
   to "Bob" at "Redmond"
tell "Mail Order Store" to order item "CW056"
   with nextday delivery
tell the front window to select the first
   paragraph containing "Macintosh"
```

WRITING ANALYSIS

In the next set of steps, you'll develop your object model hierarchy from your early command writing.

Analyze your initial commands. The consumers of your product may surprise you. Some of the sentences they write will be too large in scope, but others will be highly focused to specific tasks. You're likely to find that they'll focus on the action first, then the objects. From those sentences, begin to determine the common verbs and objects. For example:

- verbs: play, get, set, translate, send, print , select, delete
- objects: movie, customer, paragraph, document, record, message
- properties: profile, leading
- enumerators: English, French, PowerTalk

Make a crude object model hierarchy. Based on the analysis of your commands, make a first cut at your object model hierarchy. Although many object classes in your vocabulary are types of objects that can be physically manipulated by your application, objects in scripting do not have to correspond to the objects on your screen. Nor should they match the objects in your internal code created by the programmers. Rather, script objects should be the most natural representation of what the user is trying to manipulate. Often these three — scripting, onscreen, and internal — will be nearly the same, but they don't have to be.

Remember that consistency in a scriptable application is often accomplished through the liberal use of setting and getting properties instead of through large numbers of verbs. For more information, read the section "Designing Your Object Model Hierarchy" in my article, "Designing a Scripting Implementation," in *develop* Issue 21.

WORK ON YOUR DICTIONARY

The key to a clean, intuitive scriptable application is its dictionary. It's now time to develop this all-important "window" to your application's soul.

Look at other application terminologies for consistency. Creating the AppleScript interface is a lot like creating the graphical interface. When designing dialog boxes, for example, most developers look at many other applications for examples of what works and what doesn't. Similarly, you should view and use the AppleScript terminology of other applications to see how well they work. Remember that AppleScript hasn't been around long enough for strong guidelines to be developed. Often you can do better than another application (in some cases, you can learn what *not* to do), but you also want your application to share as many elements as make sense with other applications your users might be familiar with. (When in doubt, refer to and practice with the Scriptable Text Editor; it's clean and simple.)

Make your first rough 'aete' and write commands. When you're ready, take a stab at making an 'aete'. Don't expect too much at this stage; just get comfortable with the structure of this resource. Write some commands with your crude 'aete'. You can even open up your 'aete' in the Script Editor and check the syntax of your commands against your dictionary. Even though you won't be able to execute the commands, you'll be able to practice writing sentences using the terms in your early dictionary.

Adjust the 'aete'. Looking at the commands written with your early terms, you'll begin to see where the sentences look more or less natural, and where they're awkward. Based on this, you can start improving on the terms in your 'aete'.

Make more commands; have users write commands. At this point, you're ready to write some serious commands. By now you should be able to write real sentences that follow the AppleScript command structure: verb [object] [keyword value] … These sentences should be similar in structure to standard commands that you can write for other scriptable applications. They should "feel" like AppleScript:

```
play the movie "1984 Commercial"
get the profile of customer "Caroline Rose"
print pages 2 through 5
translate the document "Tech Manual" from English
   to French
set the leading of paragraphs 1 through 3 to 10
send the document "Order 578" via PowerTalk
```

Note that the use of the word "the" is allowed in many places in AppleScript. Many of your users will include it in their commands. You should name your objects and properties so that they won't sound awkward when preceded by the word "the." And try to avoid property names that start with a verb.

Give your sample 'aete' to users and ask them to begin writing scripts to see how good your terminology feels and how it integrates and interacts with other applications. This interaction is crucial to understanding the value of AppleScript. All this can be done before any code is connected to the commands in the 'aete'. (Be sure to tell them that they can't run their scripts.)

NOW TO YOUR CODE

A well-conceived dictionary will serve as a specification for programmers. Only after you've gotten your vocabulary in fairly good shape and done some preliminary testing with users should you (or your programmers) begin to write the code behind the vocabulary.

Write object accessor functions. It's probably a good idea to begin writing some of your object accessor functions first, so that you'll have something to test your Apple event handlers against. Accessor functions must cover all possible combinations of object classes and containers. However, accessor functions can be combined to handle more than one object class in a container if the objects are similar or lend themselves to code that can be shared.

For example, the Scriptable Text Editor has an accessor function for document objects, such as windows, within the application (the null container). It has another accessor function for all text objects within documents, such as characters, words, and paragraphs, and a third accessor for text objects within other text objects, such as characters within words, or words within paragraphs. Characters, words, and paragraphs were combined because the code to handle each of them was easily shared.

Also consider the language, framework, and structure of your existing code. Some frameworks, such as MacApp, use internal object member functions that are very similar to the accessor functions you'll write, lending themselves to individual accessors for each object class. You'll certainly want your accessor functions to make use of the existing internal functions.

Write Apple event handlers. Now you're ready to write the code to handle the Apple events. Since you've made the effort to lay the groundwork, this should be relatively easy. If your dictionary contains a lot of properties, consider implementing **set** and **get** early in the game.

Test your code. AppleScript is very useful for testing your Apple event code. You can easily write AppleScript commands that accurately send Apple events to your application. This is considerably easier than writing test code to fake sending Apple events to yourself. Scripter from Main Event makes an ideal tool for this task because you can observe what's going on in a script as it happens.

Once the code is connected, let a wider audience try your scripting. See how well the previously written scripts perform.

Clean up your dictionary. After you've gotten your code working, go back and carefully look over your 'aete' one more time. Make sure that you've organized the terms well and that your comments are understandable and innovative. Use the guidelines in my last column, "Thinking About Dictionaries," in Issue 23.

A NEW PLACE TO GET HELP

There's now a resource on the Internet for posing questions relating to scriptability issues. It's a new mailing list: applescript-implementors@abs.apple.com. To subscribe, just send the following message to listproc@abs.apple.com:

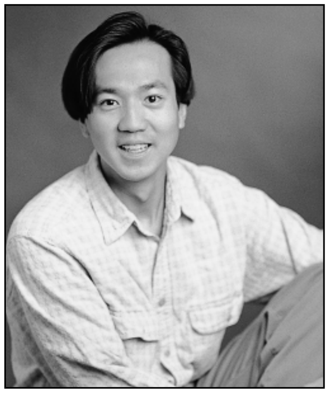
SUBSCRIBE applescript-implementors Your Name

As always, happy implementing!

Thanks to Eric Gundrum and C. K. Haun for reviewing this column.•

Getting Started With OpenDoc Storage

OpenDoc's structured storage model is an innovative departure from the traditional storage scheme. As you make the move into OpenDoc development, you need to understand the new storage model and its implications for the way data is stored and retrieved. This article introduces the new concepts and policies you'll need to know in order to use OpenDoc storage effectively.



VINCENT LO

In the traditional Macintosh user model, each application creates and maintains its own documents, storing each document in a separate file. A file has one creator signature and one file type, identifying the application it belongs to and the kind of document it contains. In OpenDoc, by contrast, a document can have multiple *parts*, created and maintained by different *part editors* (called *part handlers* in earlier versions of OpenDoc), which are analogous to the standalone applications of the traditional model. Because all of a document's parts are stored together in the same *container* (usually corresponding to a file), there has to be a way for separate part editors to share access to the same container without interfering with each other.

OpenDoc meets this need by providing a structured model for persistent storage (that is, for storing data from one session to the next). Each part is given its own *storage unit* in which to store and retrieve data. The part can thus operate as a standalone entity, independent of other parts and their storage. OpenDoc maintains all of the storage units and notifies each part when to read or write its data.

The same techniques that are used in dealing with persistent storage also apply to the various forms of data interchange between part editors, such as the Clipboard, drag and drop, and linking. Because all of these mechanisms use the same data storage medium (the storage unit), they all work essentially the same way from the part editor's point of view. For example, a part uses the same API calls to copy data to the Clipboard that it would use in writing the data to a file container. The same is true for drag and drop and for linking. Thus, once you learn how to work with OpenDoc storage units for file storage, you can use the same techniques to implement data interchange as well.

This article assumes that you're already familiar with basic OpenDoc concepts and terminology. If you need a quick introduction or refresher, see the article "The OpenDoc User Experience" in *develop* Issue 22. You can find additional information on some of OpenDoc's technical basics in the articles "Building an OpenDoc Part

VINCENT LO is Apple's technical lead for OpenDoc. When he isn't removing "unwanted features" or participating in design meetings, he divides his time equally among roller hockey, ice hockey, and explaining to his friends why he plays so much hockey. He has also been known to apply his body checking techniques in intense engineering discussions.•

Handler" in Issue 19 and "Getting Started With OpenDoc Graphics" in Issue 21. Developer releases of OpenDoc include the definitive documentation, the *OpenDoc Programmer's Guide* and *OpenDoc Class Reference.* Developer releases are available through a number of different sources, or you can request the latest release at AppleLink OPENDOC or at opendoc@applelink.apple.com on the Internet. The source code in this article is excerpted from a sample part included with the developer release.

Because OpenDoc was developed jointly by a consortium of companies including Apple, IBM, and Novell, its interfaces are designed for cross-platform compatibility, using IBM's platform-independent Standard Object Model (SOM). OpenDoc method definitions, including the ones in this article, are commonly written in a language-neutral Interface Definition Language (IDL). The SOM compiler converts these into equivalent language-specific declarations for whatever source language you happen to be using. The method definitions shown in this article, for instance, are taken from the OpenDoc interface file StorageU.idl. To use these methods in your program, you must include the corresponding language-specific *binding file* (such as StorageU.xh for a C++ program).

DRAFTS, DOCUMENTS, AND CONTAINERS

The OpenDoc classes responsible for providing storage capabilities are ODContainer, ODDocument, ODDraft, and ODStorageUnit. Collectively, a set of subclasses derived from these four is known as a *container suite.* A *container* represents the physical storage medium in which a document is stored, such as a disk file. Different container suites share the same API, but may use different low-level storage mechanisms and operate on different physical storage media. For example, the Bento container suite, which will be shipped with OpenDoc 1.0, supports both file containers and in-memory containers. A part editor can thus use the same code to store a part's data either to a file or in memory.

A single container may contain one or more documents, each of which in turn can include one or more *drafts.* A part ordinarily works with a draft, rather than directly with a document or its container. Each draft is a "snapshot" representing the state of the document at a particular point in its development. Together, the drafts embody the history of the document over time.

A part may need to interact with its draft for a variety of reasons:

- Persistent objects — Every persistent object (such as a part, a frame, or a link) is created by a draft.
- Data interchange — A part asks its draft to copy transferred objects to and from a data-interchange container, such as the Clipboard or a drag-and-drop container.
- Linking — A part uses its draft to create link specifications and copy data to and from link objects.
- Permissions — A part may need to find out whether it's allowed to write to the draft.
- Scripting — A part gets its scripting-specific identifier through its draft.

STORAGE UNITS

The basic entity of a container suite is the storage unit. Every persistent OpenDoc object has a storage unit in which to store and retrieve its data. Figure 1 shows a typical example.

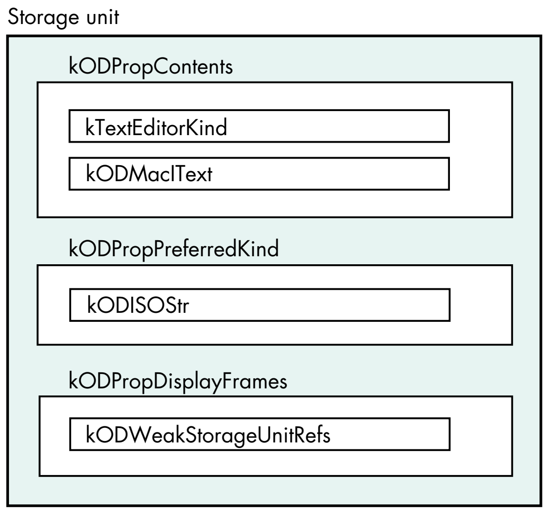


Figure 1. Structure of a storage unit

A storage unit consists of one or more *properties*, each of which in turn is associated with one or more *values* containing the data itself. The storage unit shown in Figure 1, for instance, has properties named kODPropContents, kODPropPreferredKind, and kODPropDisplayFrames; the kODPropContents property has values of types kTextEditorKind and kODMacIText.

Using multiple values allows a property to represent the same data in different forms. For example, a property holding a drawing may have three values representing the same data: one as a Macintosh PICT, one as a Windows metafile, and one in TIFF format. Although OpenDoc cannot enforce the principle, part developers are urged to use multiple values within a property only for multiple representations of the same data, not for storing unrelated data items.

The property names and value types shown in Figure 1 represent string constants of type ODPropertyName and ODValueType, respectively. For cross-platform extensibility, both of these types are defined as equivalent to an ISO string instead of a traditional Macintosh OSType: that is, they're 7-bit ASCII null-terminated strings, as specified by the International Standards Organization (ISO). The string values themselves are expected to follow a standard naming convention: for instance, the constants kODPropDisplayFrames and kODWeakStorageUnitRefs stand for the strings "OpenDoc:Property:DisplayFrames" and "OpenDoc:Type:StorageUnitRefs", respectively. The OpenDoc interface files StdProps.idl and StdTypes.idl define name constants for standard properties and value types; any property and type names that you define for yourself should follow the same naming conventions.

FOCUSING A STORAGE UNIT

The OpenDoc operations for manipulating values don't explicitly identify the value to operate on. Instead, you have to *focus* the storage unit on the desired property or value before invoking the operation. The method for setting the focus is defined in class ODStorageUnit as follows:

```
ODStorageUnit Focus(in ODPropertyName propertyName,
                    in ODPositionCode propertyPosCode,
                    in ODValueType valueType,
                    in ODValueIndex valueIndex,
                    in ODPositionCode valuePosCode);
```

This allows you to set the storage unit's focus in a variety of ways:

- to a property by name
- to a property by position relative to the current property
- to a value by type within a property
- to a value by position within a property
- to a value by position relative to the current value

Properties and values are ordered within the storage unit according to the sequence in which they were added. Values within a property are indexed from 1: that is, the first value has index 1, the second index 2, and so on. Positions relative to the current focus are specified with a *position code.* The same position code can refer to either a property or a value, depending on the current focus. For instance, if the storage unit is currently focused on a property, the position code kODPosNextSib designates the next property; if the current focus is on a value, kODPosNextSib designates the next value.

Another way to set the focus of a storage unit is with a *storage unit cursor:*

```
ODStorageUnit FocusWithCursor(in ODStorageUnitCursor cursor);
```

The cursor identifies a property by name or a value by its property name and its index or value type. Once created (with method CreateCursor or CreateCursorWithFocus of class ODStorageUnit), the same cursor can be reused multiple times to refer to properties or values within the storage unit.

Once you've focused a storage unit, you can create a *storage unit view* to refer to the same property or value again later without having to reset the focus:

```
ODStorageUnitView CreateView();
```

The view responds to all the same access methods as the storage unit itself, but applies them to the property or value that had the focus at the time the view was created, rather than at the time the method is invoked. It does this by automatically resetting the underlying storage unit to the original focus, then forwarding the method call to the storage unit for processing.

MANIPULATING VALUE DATA

The operations for manipulating data within a storage value are stream-based, very much like reading or writing to a sequential file. Each value has a current offset position that controls where the next operation will take place, similar to the file mark in the Macintosh file system. In addition to reading and writing data sequentially, you can also insert or delete data at the current offset position.

Class ODStorageUnit defines the following methods for manipulating value data:

```
void SetOffset(in ODULong offset);
ODULong GetOffset();
void SetValue(in ODByteArray value);
ODULong GetValue(in ODULong length, out ODByteArray value);
void InsertValue(in ODByteArray value);
void DeleteValue(in ODULong length);
```

The ODByteArray structure is used to pass data to or from a storage unit.

```
typdef struct {
   unsigned long  _maximum;   /* size of buffer */
   unsigned long  _length;    /* number of bytes of actual data */
   octet*         _buffer;    /* pointer to buffer containing the data */
} _IDL_SEQUENCE_octet;

typedef _IDL_SEQUENCE_octet ODByteArray;
```

(An *octet* is simply the SOM term for an 8-bit byte.) Listing 1 shows how to manipulate one of the values shown in Figure 1.

Listing 1. Adding data to a value

```
/* Focus the storage unit, using property name and value type. */
storageUnit->Focus(ev, kODPropContents, kODPosUndefined, kTextEditorKind,
                     0, kODPosUndefined);

/* Set up the byte array. */
ODByteArray ba;
ba._length = size;
ba._maximum = size;
ba._buffer = buffer;

/* Set the offset. (This step isn't really needed here, since the
   Focus operation automatically sets the offset to 0. It's included
   for illustrative purposes only.) */
storageUnit->SetOffset(ev, 0);

/* Add the value. */
storageUnit->SetValue(ev, &ba);
```

STORAGE UNIT REFERENCES

Storage unit references allow one storage unit to refer persistently to another. A part can use this mechanism to access information stored in a storage unit (which may or may not belong to it) across multiple sessions. A draft thus consists essentially of a network of storage units connected to each other with persistent references.

When a storage unit is cloned (copied to a data-interchange container), any other storage units it references are cloned along with it. Since all storage units in a draft are interconnected, cloning any one of them may cause the whole draft to be cloned. Because this may be an expensive and unnecessary operation, OpenDoc provides two levels of storage unit reference: strong and weak. Only strongly referenced storage units are copied when the unit that refers to them is cloned.

In Figure 2, frame A refers strongly to part A, which refers strongly to frame B, which refers strongly to part B. Thus if frame A's storage unit is cloned, all four storage units will be copied. On the other hand, cloning frame B's storage unit will copy those for frame B and part B only, since frame B's reference to frame A is weak rather than strong.

An object can use strong storage unit references to refer to other objects that are essential to its functioning, such as embedded frames. Weak references are mainly for informational or secondary purposes: a part might use them, for instance, to refer to its display frames.

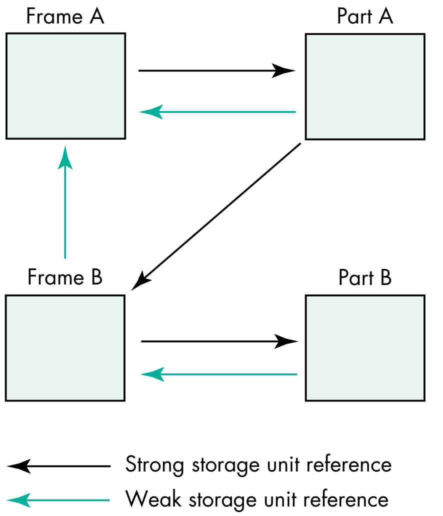


Figure 2. Strong and weak storage unit references

LIFE CYCLE OF A PART

Figure 3 shows the life cycle of a part and its associated storage unit. Because the part's lifetime may span multiple editing sessions, it must be able to *externalize* its internal state (save it to persistent storage) in order to reconstruct itself from one session to the next. The part's InitPart method, called when the part is first created, receives a storage unit as a parameter. The Externalize method can then use this storage unit to save the part's state. Once externalized, the part can be released from memory and later reconstituted from external storage by a method named InitPartFromStorage. Unlike InitPart, InitPartFromStorage can be called multiple

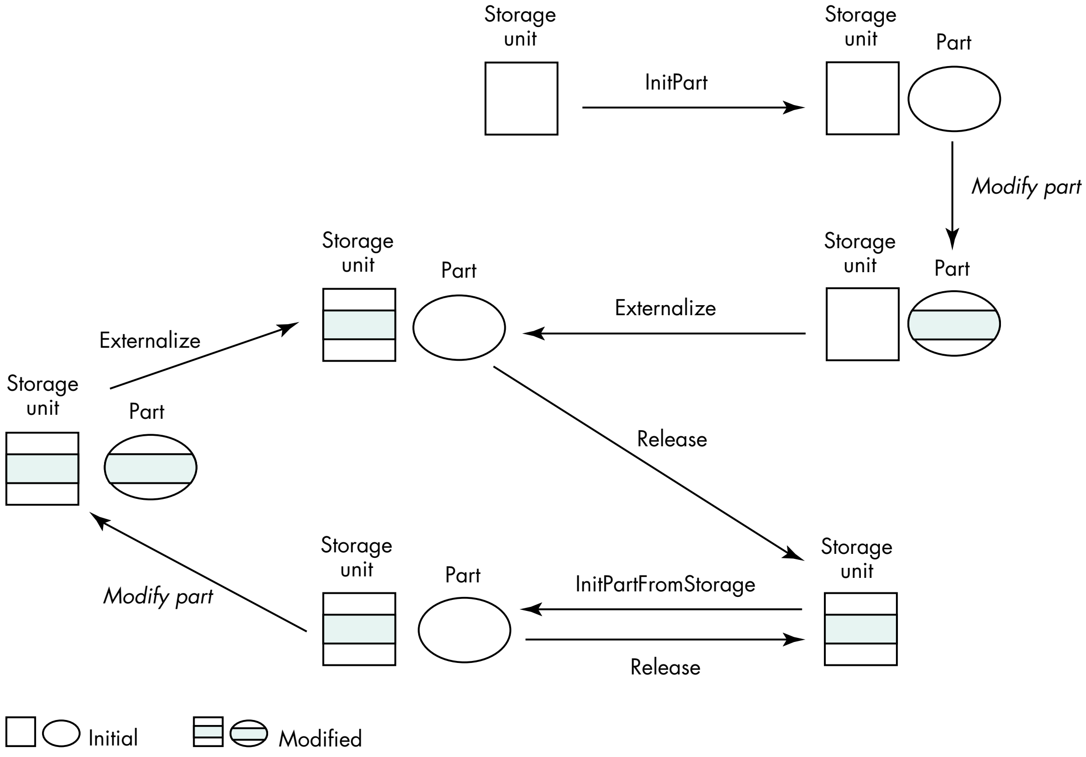


Figure 3. Life cycle of a part

times during a part's lifetime, whenever the part needs to be reconstructed from external storage.

Notice that externalizing a part is not the same as cloning it. Externalizing means writing the part's data to persistent storage, using a storage unit associated with the draft in which the part resides; cloning is transferring the part's data to a data-interchange container such as the Clipboard, using a storage unit associated with the container. Although the two operations are different, they're both based on the same ODStorageUnit API and can share much of the same code.

Another related operation is *purging*, which reclaims memory space by eliminating unnecessary runtime data structures such as caches. Because such structures can usually be reconstructed from persistent data, many OpenDoc programmers believe that a part's Purge method should always begin by externalizing the part's data before deleting unused or unnecessary memory. While this might sound plausible in principle, the externalization operation itself requires additional memory — the very thing that's in short supply during purging. As a general rule, the Purge method should avoid invoking externalization unless it's absolutely necessary.

All persistent objects carry a reference count, enabling OpenDoc to identify unused objects and reclaim the memory they occupy. The Acquire method, which creates a reference to a specified object, increments the object's reference count; the Release method destroys a reference and decrements the reference count. When the reference count goes down to 0, OpenDoc can safely delete the object from memory.

INITIALIZATION

The initialization method InitPart is called only once, to set up a part's initial state. It should take the following actions:

1. Call the parent class's InitPart method to perform any initialization required at the parent level.
2. Save the incoming part wrapper object (discussed below) in an internal field.
3. Set up an internal permissions field to indicate that writing to the draft is allowed.
4. Set up the part's runtime data structures.
5. Set the part's internal dirty flag to true.

Listing 2 shows an example. Notice that the SOM compiler, in translating the method declaration from language-independent IDL into a specific source language, adds two additional parameters at the beginning of the parameter list: a pointer to the object executing the method (somSelf) and an environment pointer (ev) used for error reporting. All of our example method definitions in this article begin with these two parameters.

Parent initialization. It's important for a part's initialization method to call that of its parent class. The parent's initialization method will in turn call that of *its* parent and so on up the inheritance chain, ensuring that all of the part's inherited properties are properly initialized. Inherited properties set up by ODPart and its parents, such as ODPersistentObject, include the following:

- kODPropCreateDate contains the part's creation date.
- kODPropModDate tells when the part's storage unit was last externalized.
- kODPropModUser contains the name of the last user who modified the part.

Listing 2. Initializing a part

```
SOM_Scope   void
SOMLINK     TextEditor__InitPart(SampleCode_TextEditor  *somSelf,
                                 Environment            *ev,
                                 ODStorageUnit          *storageUnit,
                                 ODPart                 *partWrapper)
{
   SampleCode_TextEditorData *somThis =
                                 SampleCode_TextEditorGetData(somSelf);
   SOMMethodDebug("TextEditor", "InitPart");

   SOM_TRY
      // Call the parent class's InitPart method. The parent will in
      // turn call its parent, and so on.
      parent_InitPart(somSelf, ev, storageUnit, partWrapper);

      // Store part wrapper object in an internal field.
      _fSelf = partWrapper;

      // Set a flag showing that this draft is not read-only.
      _fReadOnlyStorage = kODFalse;

      // Call common initialization code to set up our initial state.
      somSelf->Initialize(ev);

      // Set the dirty flag to true.
      somSelf->SetDirty(ev);

   SOM_CATCH_ALL
      // No explicit code needed here: cleanup will be performed by the
      // destructor, which is called automatically when an error is
      // thrown.

   SOM_ENDTRY
}
```

Part wrapper. Every part is *wrapped* by another object, called its *part wrapper.* Clients of the part object deal with it indirectly, through the part wrapper, instead of holding a direct pointer to the part object itself. The part wrapper receives all method invocations and delegates them to the actual part. This insulation of the part object allows the part editor to be changed at run time without affecting its clients.

The InitPart method should save the part wrapper object in an internal field. Then, whenever the part needs to pass an object representing itself as a parameter, it should pass the part wrapper in place of itself.

Draft permissions. A part editor needs to know whether a part is in a read-only draft. If so, its functionality may be restricted: for example, the part may not allow the user to change its contents, either through keyboard input or through menu operations such as Cut and Paste. Also, if the draft is read-only, its Externalize method need never be called on its parts or any persistent objects. When a part is created for the first time, its draft is guaranteed to be writable, so it should initialize its read-only flag to false.

Dirty flag. The purpose of a dirty flag is to let the part's Externalize method know whether it needs to write out the part's state to external storage. Externalization (especially to disk) can be a time-consuming and expensive operation; using a dirty flag can greatly improve performance by avoiding it whenever possible.

When a part is first created, its storage unit is empty. Since the state has not yet been written out, the part should initialize its dirty flag to true; the flag should also be set to true whenever the contents of the part are changed. After saving the state and content data to external storage, the Externalize method should clear the flag to false, indicating that the state need not be saved again unless the part's contents are changed.

EXTERNALIZATION

A part's Externalize method can be called at any time. Typically, it's called by the draft when the user chooses to save the document. Since a part has no idea when this may happen, it should always be ready to externalize itself.

The Externalize method should do the following:

1. Call the parent class's Externalize method.
2. Check that all required properties exist; if not, add them to the storage unit.
3. Clean up the part's contents if necessary.
4. Write out the part's state information and contents.
5. Clear the part's internal dirty flag to false.

Listing 3 shows an example.

The contents of a part must be written out to a special *content property* named kODPropContents. Like other properties, the content property can contain multiple values representing the same data in different forms. A value type used for content data is referred to as a *part kind.* To facilitate data interchange, part editors are encouraged to include one or more standard part kinds in their content property, much the way traditional Macintosh applications use common data formats like 'TEXT' or 'PICT' when writing to the Clipboard.

Each value in the content property should be a complete representation of the content data. A value may contain references to other storage units, but cannot depend on other values in the content property or on other properties in the part's storage unit. Even if every other property and value were deleted from the storage unit, the part editor should still be able to reconstruct the part using just that one content value.

The ordering of values within the content property is completely determined by the part editor. An important principle, however, is that values that represent the underlying contents with greater fidelity should precede those of lesser fidelity: formatted text, for instance, should precede plain (unformatted) text. The first value should be the one that represents the content most faithfully.

When a part editor reconstructs a part from an external storage unit, there's a chance that the storage unit may have originally been written by some other part editor. As a result, the content property may contain part kinds that the current part editor doesn't support, or the values may appear in the wrong fidelity order. In this case, the part's Externalize method should remove all existing values from the content property so that it can write out its own content data in proper fidelity order.•

Listing 3. Externalizing a part

```
SOM_Scope   void
SOMLINK     TextEditor__Externalize(SampleCode_TextEditor  *somSelf,
                                    Environment            *ev)
{
   SampleCode_TextEditorData *somThis =
                                SampleCode_TextEditorGetData(somSelf);
   SOMMethodDebug("TextEditor", "Externalize");

   SOM_CATCH return;

   // Ask parent classes to externalize themselves.
   parent_Externalize(somSelf, ev);

   // Check dirty flag.
   if (_fDirty) {
      // Get storage unit.
      ODStorageUnit *storageUnit = somSelf->GetStorageUnit(ev);

      // Verify that the storage unit has the appropriate properties;
      // if not, add them.
      somSelf->CheckAndAddProperties(ev, storageUnit);

      // Validate storage unit's contents and clean up if necessary.
      somSelf->CleanseContentProperty(ev, storageUnit);

      // Write out state information and contents.
      somSelf->ExternalizeStateInfo(ev, storageUnit, 0, kODNULL);
      somSelf->ExternalizeContent(ev, storageUnit, kODNULL);

      // Clear dirty flag.
      _fDirty = kODFalse;
   }
}
```

A standard property named kODPropPreferredKind identifies the part kind that the user chooses to represent the data. If this property already exists, the part editor shouldn't tamper with it; if it doesn't exist, the part editor may create it and give it a value of type kODISOStr containing the name of the highest-fidelity part kind. When writing out the content data, the part editor should be sure to include a value in the format specified by this property.

RECONSTRUCTION

The InitPartFromStorage method is called whenever a part object needs to be reconstructed from external storage. This method should do the following:

1. Call the parent class's InitPartFromStorage method.
2. Save the incoming part wrapper object in an internal field.
3. Set up an internal permissions field to indicate whether writing to the draft is allowed.
4. Set up the part's runtime data structures.

5. Read the content data from the storage unit into the runtime data structures.
6. Clear the part's internal dirty flag to false.

Notice that these are essentially the same steps we listed earlier for the InitPart method, except that the contents of the part's runtime data structures are read in from the storage unit instead of being initialized to standard values, and that the dirty flag is cleared to false instead of true to show that the part's contents agree with those in the external storage unit. Listing 4 shows an example of an InitPartFromStorage method.

Listing 4. Reconstructing a part

```
SOM_Scope   void
SOMLINK     TextEditor__InitPartFromStorage
                                 (SampleCode_TextEditor  *somSelf,
                                 Environment            *ev,
                                 ODStorageUnit          *storageUnit,
                                 ODPart                 *partWrapper)
{
   SampleCode_TextEditorData *somThis =
                                 SampleCode_TextEditorGetData(somSelf);
   SOMMethodDebug("TextEditor", "InitPartFromStorage");

   // Avoid initializing the part twice.
   if (fSelf != kODNULL)
      return;

   SOM_TRY
      // Call the parent class's InitPartFromStorage method. The parent
      // will in turn call its parent, and so on.
      parent_InitPartFromStorage(somSelf, ev, storageUnit, partWrapper);

      // Store part wrapper object in an internal field.
      _fSelf = partWrapper;

      // Set a flag showing whether this draft is read-only.
      _fReadOnlyStorage = (storageUnit->GetDraft(ev)->
                              GetPermissions(ev) == kDPReadOnly);

      // Call common initialization code to set up our initial state.
      somSelf->Initialize(ev);

      // Read in state data from external storage.
      somSelf->InternalizeStateInfo(ev, storageUnit);

      // Read in content data from external storage.
      somSelf->InternalizeContent(ev, storageUnit);

   SOM_CATCH_ALL
      // No explicit code needed here: cleanup will be performed by the
      // destructor, which is called automatically when an error is
      // thrown.

   SOM_ENDTRY
}
```

As we've already noted, the storage unit from which a part is reconstructed may have been created by a part editor other than the one reading it in. The OpenDoc binding subsystem uses the part kinds found in the storage unit's content property to determine which part editor to invoke. If the original part editor cannot be found, the binding subsystem will look for another editor capable of reading the available part kinds. The contents of the storage unit may thus be very different from what the current part editor expects. Here are a few points to note:

- If the storage unit identifies a preferred part kind (that is, if it contains the property kODPropPreferredKind), the part editor should read its content data from the indicated value of the content property. If no preferred kind is specified (or if the part editor cannot handle a value of the specified kind), it should iterate through the available values looking for one it can handle. When it finds such a value, it should read the content data from that value into its runtime data structures.
- The InitPartFromStorage method should not add its own properties to the part's storage unit, but should leave that task to the Externalize method instead. This is because the user may close the document without modifying any of its contents. If the InitPartFromStorage method modifies the storage unit, the user will be prompted to save the document before closing it, even though the document has not been modified.
- The part editor should not alter the part's preferred-kind property (kODPropPreferredKind).

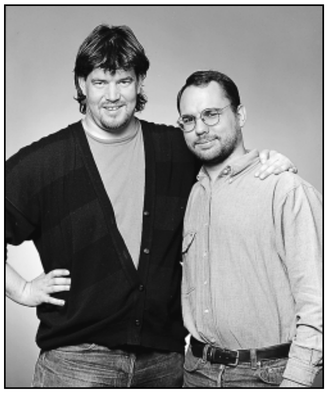
WHAT NEXT?

Needless to say, the only real way to get familiar with OpenDoc programming is to jump in and develop a part editor of your own. The techniques discussed in this article will help you manage your storage needs effectively. The rest is up to you and your imagination.

RELATED READING

- "The OpenDoc User Experience" by Dave Curbow and Elizabeth Dykstra-Erickson, *develop* Issue 22.
- "Getting Started With OpenDoc Graphics" by Kurt Piersol, *develop* Issue 21.
- "Building an OpenDoc Part Handler" by Kurt Piersol, *develop* Issue 19.
- *OpenDoc Programmer's Guide* and *OpenDoc Class Reference*, available as part of the OpenDoc developer releases.
- The latest news on OpenDoc can be found on the World Wide Web at http://www.info.apple.com/opendoc or http://www.cilabs.org.

Thanks to our technical reviewers Dave Bice, Craig Carper, Ed Lai, and Steve Smith.•



GRAPHICAL TRUFFLES

Making the Most of QuickDraw 3D

NICK THOMPSON AND PABLO FERNICOLA

For those of us on Apple's QuickDraw 3D team, the highlight of SIGGRAPH '95 (the annual conference of the ACM's computer graphics interest group) was having the chance to work with developers who were showing QuickDraw 3D products. Considering that we only started working with developers in December 1994, the number of applications already up and running is inspiring. By the time you read this column, 10 or 15 QuickDraw 3D products will be shipping, including modeling and animation software, 3D hardware accelerators, 3D model clip art, and games. More than 50 developers are actively working on products based on QuickDraw 3D, and those will ship in 1996.

If you're not yet a QuickDraw 3D developer and don't want to be left out, take a look at the *develop* articles "QuickDraw 3D: A New Dimension for Macintosh Graphics" in Issue 22 and "The Basics of QuickDraw 3D Geometries" in Issue 23. This column gives a hodgepodge of additional information.

IMPROVING ACCELERATOR PERFORMANCE

One of the things that has attracted developers to QuickDraw 3D is seamless access to hardware acceleration. In addition to Apple's PCI accelerator card, hardware acceleration cards have been announced by Matrox, Yarc, Radius, and Newer Technology. If you really want your application to fly, you need to make sure that you're using the fastest renderer possible and that if a hardware acceleration card is installed, you're using the card. If you use the QuickDraw 3D API, QuickDraw 3D will take care of this for you, but there's something else you can do that might improve your application's performance.

Certain cards, including Apple's accelerator card, will yield better frame rates in some situations if you use what we call *double buffer bypass*, an option enabled by a flag. Double buffering causes all objects to be drawn first into a back buffer; this entire buffer is then copied to the front buffer (the window). If the scene you're rendering is simple and thus takes very little time to redraw — say, less than 1/10 of a second — enabling double buffer bypass is faster because it avoids having to copy memory from the back buffer to the front buffer. On the other hand, if you use this option with a complex scene, tearing may occur. Therefore, you may want to time a frame (and take into account the complexity of your models) before using double buffer bypass. To time a frame, call the Toolbox routine Microseconds, draw the frame, call Q3Renderer_Sync to make sure the frame has been fully drawn, and then call Microseconds again and subtract the start time from the end time.

If you're using QuickDraw 3D's interactive software renderer, all the code you need to turn on double buffer bypass is shown in Listing 1.

The interactive renderer can render using software only or using hardware acceleration. The interactive renderer is set by default to look for the best device possible, so if a hardware accelerator is installed, the accelerator will always be used. On occasion, though, you may want to switch from using hardware to using software (for demos or testing, for example). In this case you must explicitly request the software rasterizer, as follows:

```
Q3InteractiveRenderer_SetPreferences(myRenderer,
   kQAVendor_Apple, kQAEngine_AppleSW);
```

INTERACTING WITH INPUT DEVICES

QuickDraw 3D provides an input device abstraction layer that allows you to interact with different input devices without having to write special code for each of them. The sample application NewEra demonstrates

NICK THOMPSON (eWorld NICKT), transplanted English soccer fan and member of Apple's Developer Technical Support team, thinks that this could be the year for the Arsenal Football Club. With the acquisition of Dutch star Dennis Bergkamp and England striker David Platt, things are looking up at Highbury. By the time you read this, the Premier League standings will tell if this is the dawning of a new era or more of the same "boring, boring, Arsenal," as those charming Spurs fans like to chant.•

PABLO FERNICOLA (eWorld EscherDude), of Apple's Interactive Multimedia Group, is much more relaxed since shipping QuickDraw 3D 1.0. He now has time to eat his dad's great barbecue, dally with his lovely wife, and sleep — although the latter entails the challenge of trying to get his golden retriever, aptly named Mac, to give up some of the space he takes up in the bed. Pablo's latest research project is to find out exactly what the purpose is for those orange balls that one finds on high power transmission lines.•

interaction with tablets and other input devices; this application is available on the CD that comes with the book *3D Graphics Programming With QuickDraw 3D*, and a newer version can be found on this issue's CD.

To take advantage of QuickDraw 3D's input device layer, you need to create a tracker object and associate it with a controller object (created by an input device driver), as Listing 2 does. Once you've set up your tracker, you can poll it to get its new position and orientation, as shown in Listing 3. To reflect the change in your scene, you apply the values returned by the tracker to a transform object, affecting either a particular geometry or group (if an object was selected and being manipulated) or the camera, depending on the interaction model for your application.

QuickDraw 3D's input device abstraction layer also makes writing input device drivers easier. For example, it took us about three days to write a driver for the Magellan device from Logitech, Inc., a 3D input device with six degrees of freedom. As illustrated in Figure 1, this device enables movement along the x, y, and z axes, as well as rotation about the three axes.

SETTING THE CORRECT FILE TYPE

When saving QuickDraw 3D metafiles, you should set the file type as '3DMF', regardless of how the contents of the file are formatted (as plain-text or binary, or any combination of the different types of organization, such as database or stream). This will enable the file to be read or opened by other QuickDraw 3D applications. If you'd like your end users to read a file as text, add an Export As Text option to your application and then set the file type to 'TEXT'. This is helpful for debugging (and for sending questions or bugs to Developer Technical Support).

HAVING FUN WITH CUSTOM ATTRIBUTES

By taking advantage of QuickDraw 3D's custom attributes and extensible metafile format, you can have objects that encapsulate specialized data relevant to

Listing 1. Turning on double buffer bypass

```
// Create the renderer.
if ((myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive)) != nil) {
   if ((myStatus = Q3View_SetRenderer(myView, myRenderer)) == kQ3Failure) {  // Handle the error.
      ...
   }  // Set bypass.
   Q3InteractiveRenderer_SetDoubleBufferBypass(myRenderer, kQ3True);
}
```

Listing 2. Creating a tracker object and attaching it to a controller object

```
theDocument->fPositionSN = 0;
theDocument->fRotationSN = 0;
theDocument->fTracker = Q3Tracker_New(NULL);
myStatus = Q3Controller_Next(NULL, &controllerRef);
while (controllerRef != NULL && myStatus == kQ3Success) {
   Q3Controller_SetTracker(controllerRef, theDocument->fTracker);
   myStatus = Q3Controller_Next(controllerRef, &controllerRef);
}
```

Listing 3. Updating position and orientation

```
// We received a null event; grab a new position and orientation for the model.
TQ3Boolean      positionChanged;
TQ3Boolean      rotationChanged;

Q3Tracker_GetPosition(doc.fTracker, &doc.fPosition, NULL, &positionChanged, &doc.fPositionSN);
Q3Tracker_GetOrientation(doc.fTracker, &doc.fRotation, NULL, &rotationChanged, &doc.fRotationSN);
```

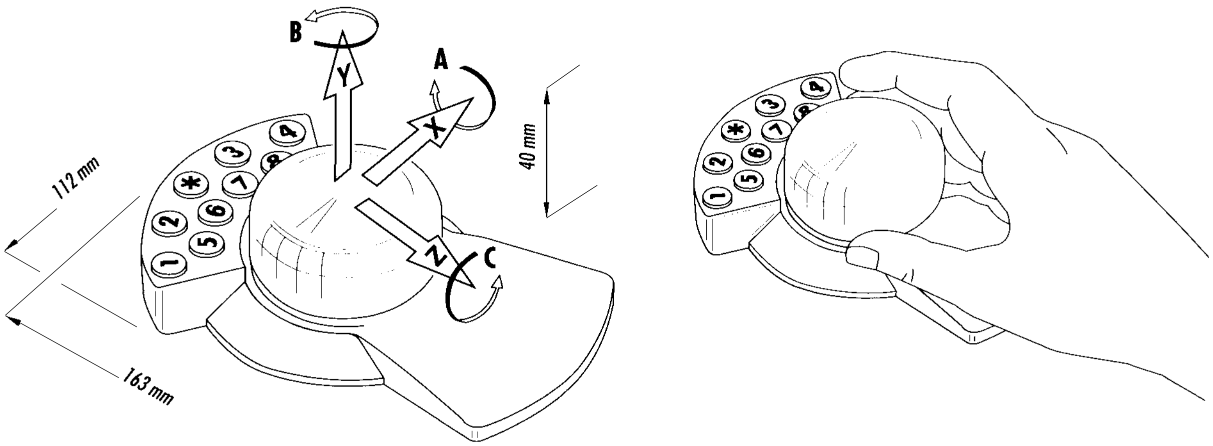


Figure 1. Magellan: a six-degrees-of-freedom input device (courtesy of Logitech)

your application. For instance, to navigate through the World Wide Web in 3D, you can attach Web data (like URLs) to QuickDraw 3D objects as custom attributes. When those objects or scenes are read into one of the many viewers supporting the URL custom attribute, the viewer can communicate through Apple events with applications like Netscape (or your favorite Web browser) to produce 3D navigation. You'll find a sample application that shows how to do this on this issue's CD.

Custom attributes also enable you to associate sound and other data with objects in your 3D scene.

DEBUGGING

There are two really handy techniques that you can use to diagnose problems you may be having with your QuickDraw 3D application. For both of these approaches to debugging your software, you'll want to make sure that you have MacsBug installed on your machine and that you're using the debugging version of the QuickDraw 3D extension supplied with the QuickDraw 3D development software.

The first technique is to install error and warning handlers, described in our article in *develop* Issue 22. Error and warning handlers are particularly useful for telling you of potential problems with your use of the QuickDraw 3D library. If you don't install error and warning handlers, you won't know if you're doing something that the library identifies as erroneous. Although we stated this in our original article, many developers missed its significance and thus have experienced longer debugging times than necessary and a great deal of frustration.

The second technique is to use a software tool, the 3D debugger, included on this issue's CD. This application enables you to examine the QuickDraw 3D heap and look at the different objects, their attributes, and their reference count. Please note that you're looking under the hood, so you may encounter untyped blocks, and the reference count for objects may reflect references internal to the QuickDraw 3D system.

LOOKING AHEAD

We'll continue to release great new QuickDraw 3D features, so bring your applications along for the ride. By early 1996 we expect to have all major existing 3D applications on the Macintosh using QuickDraw 3D, along with applications that developers port from other platforms. Many 2D applications will be making use of the 3D Viewer as well.

Watch *develop* for further articles about other aspects of QuickDraw 3D. Meanwhile, you may want to check out the Addison-Wesley book *3D Graphics Programming With QuickDraw 3D* (which includes the QuickDraw 3D development software) and see this issue's CD for the development software and the latest versions of the sample code and utility applications. And for the latest news on QuickDraw 3D, see our Web page at http://www.info.apple.com/qd3d.

For more information on making your application work with Magellan, contact Stephan Ilberg at Logitech by sending a message to stephan_ilberg@logitech.com.•

Thanks to Robert Dierkes and Fábio Pettinati for reviewing this column, and a special thanks to Dan Venolia and David Vasquez for supplying some of the code and applications discussed.•

Sound Secrets

The Sound Manager is one powerful multimedia tool for the Macintosh, but no one has ever accused it of being too obvious. This article explores some of the more subtle Sound Manager features, showing some simple ways to improve your application's use of sound. A sample application demonstrates features such as volume overdrive and easy continuous sound.

KIP OLSON

The Sound Manager has a long and distinguished career on the Macintosh. First released in 1987, it was completely revised in 1993 with the release of Sound Manager 3.0. The introduction of Sound Manager 3.1 in the summer of 1995 brought native PowerPC performance, making the Sound Manager one of the most powerful multimedia tools around. However, getting the most out of the Sound Manager often means wading through many pages of *Inside Macintosh: Sound.*

This article pulls together valuable information about the Sound Manager, focusing on some of its little-known features that will ease your development of multimedia applications. The tips and techniques come straight from the Sound Manager development team at Apple and cover diverse areas of developer interest, including

- parsing sound resources
- displaying compression names
- maximizing performance
- adjusting volume
- controlling pitch
- playing continuous sounds
- compressing audio

Two of these topics, controlling pitch and compressing audio, require the use of Sound Manager 3.1, which is included on this issue's CD. You'll also find the SoundSecrets application and its source code on the CD. SoundSecrets demonstrates many of the techniques described in the article. To get the most out of this article, you should be familiar with the Sound Manager command interface and concepts such as sound channels, as described in *Inside Macintosh: Sound.*

So, let's get started unlocking some of those sound secrets!

KIP OLSON was recently dispatched to the Copland team at Apple with orders to rewrite the Sound Manager (again). To keep things interesting, he promises to add even more obscure features.•

FIND WHAT YOU'RE LOOKING FOR

On the Macintosh, sounds can be stored in a variety of formats, including 'snd ' resources, AIFF (Audio Interchange File Format) files, and QuickTime movies. Applications often need to read these files directly and extract their sound data, which can be a daunting task, especially when you begin to deal with some of the new compressed sound formats introduced in Sound Manager 3.1 — for example, IMA 4:1.

Fortunately, Sound Manager 3.0 introduced a couple of routines to help you navigate these tricky waters — GetSoundHeaderOffset and GetCompressionInfo. Let's take a look at these routines, and put them to work with an example of parsing an 'snd ' resource taken from the SoundSecrets application.

The 'snd ' resource format is described fully in *Inside Macintosh: Sound*, so we won't go into detail here, except to say that embedded in the resource is a sound header and the audio samples themselves. Finding this embedded sound header is the job of GetSoundHeaderOffset. It takes a handle to an arbitrary 'snd ' resource and returns the offset of the sound header data structure within that handle.

However, once you find the sound header, your work is not complete; you must determine which of the three possible sound header structures it is. In the SoundSecrets application, the sound header is represented as a union of the three structures SoundHeader, ExtSoundHeader, and CmpSoundHeader. The **encode** field in these structures determines which union member to use when examining the header.

After you've extracted the appropriate information from the sound header, you can use the GetCompressionInfo routine to determine the sound format and the compression settings. GetCompressionInfo fills out and returns a CompressionInfo record, which contains the OSType format of the sound, samples per packet, bytes per packet, and bytes per sample. You can use these fields to convert between samples, frames, and bytes.

For a thorough discussion of GetCompressionInfo, see the Macintosh Technical Note "GetCompressionInfo()" (SD 1).•

As shown in Listing 1, the SoundSecrets application uses GetSoundHeaderOffset to find the sound header structure, and then uses a **case** statement based on the **encode** field to extract the useful information from each type of header. The SoundSecrets application calculates the number of samples in the sound using information returned by GetCompressionInfo.

CHOOSE THE RIGHT NAME

Now that you've extracted the sound settings from an 'snd ' resource, the next thing you'll want to do is display this information to the user of your application. Settings like sample rate and sample size are easy to display, but what if the sound is compressed? All you've got is an OSType to describe the compressed sound data format, and not too many users are going to get much out of seeing something like 'MAC3' displayed on their screen.

Fortunately, the Sound Manager makes it easy for you to find a string to display that does make sense. Using the Component Manager, you can look up the name of the audio codec used to expand the compressed sound, and use this name to describe the compression format to the user.

This is done with the Component Manager routine FindNextComponent, which is passed a ComponentDescription record. By setting the componentType field of this

Listing 1. Getting information from the sound header

```
typedef union {
   SoundHeader      s;       // Plain sound header
   CmpSoundHeader   c;       // Compressed sound header
   ExtSoundHeader   e;       // Extended sound header
} CommonSoundHeader, *CommonSoundHeaderPtr;

OSErr ParseSnd(Handle sndH, SoundComponentData *sndInfo,
      CompressionInfo *compInfo, unsigned long *headerOffsetResult,
      unsigned long *dataOffsetResult)
{
   CommonSoundHeaderPtr    sh;
   unsigned long           headerOffset, dataOffset;
   short                   compressionID;
   OSErr                   err;

   // Use GetSoundHeaderOffset to find the offset of the sound header
   // from the beginning of the sound resource handle.
   err = GetSoundHeaderOffset((SndListHandle) sndH,
         (long *) &headerOffset);
   if (err != noErr)
      return (err);

   // Get pointer to the sound header using this offset.
   sh = (CommonSoundHeaderPtr) (*sndH + headerOffset);
   dataOffset = headerOffset;

   // Extract the sound information based on encode type.
   switch (sh->s.encode) {
      case stdSH:    // Standard sound header
         sndInfo->sampleCount = sh->s.length;
         sndInfo->sampleRate = sh->s.sampleRate;
         sndInfo->sampleSize = 8;
         sndInfo->numChannels = 1;
         dataOffset += offsetof(SoundHeader, sampleArea);
         compressionID = notCompressed;
         break;

      case extSH:    // Extended sound header
         sndInfo->sampleCount = sh->e.numFrames;
         sndInfo->sampleRate = sh->e.sampleRate;
         sndInfo->sampleSize = sh->e.sampleSize;
         sndInfo->numChannels = sh->e.numChannels;
         dataOffset += offsetof(ExtSoundHeader, sampleArea);
         compressionID = notCompressed;
         break;

      case cmpSH:    // Compressed sound header
         sndInfo->sampleCount = sh->c.numFrames;
         sndInfo->sampleRate = sh->c.sampleRate;
         sndInfo->sampleSize = sh->c.sampleSize;
         sndInfo->numChannels = sh->c.numChannels;
```

(continued on next page)

Listing 1. Getting information from the sound header *(continued)*

```
         dataOffset += offsetof(CmpSoundHeader, sampleArea);
         compressionID = sh->c.compressionID;
         sndInfo->format = sh->c.format;
         break;

      default:
         return (badFormat);
         break;
   }

   // Use GetCompressionInfo to get the data format of the sound and
   // the compression information.
   compInfo->recordSize = sizeof(CompressionInfo);
   err = GetCompressionInfo(compressionID, sndInfo->format,
         sndInfo->numChannels, sndInfo->sampleSize, compInfo);
   if (err != noErr)
      return (err);

   // Store the sound data format and convert frames to samples.
   sndInfo->format = compInfo->format;
   sndInfo->sampleCount *= compInfo->samplesPerPacket;

   // Return offset of header and audio data.
   *headerOffsetResult = headerOffset;
   *dataOffsetResult = dataOffset;

   return (noErr);
}
```

record to kSoundDecompressor, the componentSubType field to the OSType of the compressed sound data format, and the remaining fields to 0, you can search for the sound component that will decompress the sound. Once you have the component, you can use GetComponentInfo to obtain the component name, which is the descriptive string that makes sense to the user. The routine from SoundSecrets shown in Listing 2 finds the name of any compressed sound format.

MAXIMIZE YOUR POTENTIAL

The Sound Manager is almost always used in conjunction with other operations on the Macintosh. For example, QuickTime uses the Sound Manager to play a sound track while it's drawing the frames of a movie, and games play sound effects and background music while animating the screen. That's why the performance of the Sound Manager is of such great concern to many programmers: if the Sound Manager takes too much time to do its work, QuickTime will begin to drop video frames and games or animations will run slower.

To get the best performance out of the Sound Manager, you first need to understand a little about how it plays a sound. The Sound Manager's major function is to convert the sounds played by an application into the audio format required by the sound hardware on a particular computer. For example, the sound hardware on the Power Macintosh 8100 requires a stream of 16-bit, stereo, 44.1 kHz audio samples, so the Sound Manager must convert all sounds to this format during playback.

Listing 2. Finding the name of a compressed sound format

```
OSErr GetCompressionName(OSType compressionType, Str255 compressionName)
{
   ComponentDescription   cd;
   Component              component;
   Handle                 componentName;
   OSErr                  err;

   // Look for decompressor component.
   cd.componentType = kSoundDecompressor;
   cd.componentSubType = compressionType;
   cd.componentManufacturer = 0;
   cd.componentFlags = 0;
   cd.componentFlagsMask = 0;

   component = FindNextComponent(nil, &cd);
   if (component == nil) {
      err = siInvalidCompression;
      goto FindComponentFailed;
   }

   // Create handle for name.
   componentName = NewHandle(0);
   if (componentName == nil) {
      err = MemError();
      goto NewNameFailed;
   }

   // Get name from the Component Manager.
   err = GetComponentInfo(component, &cd, componentName, nil, nil);
   if (err != noErr)
      goto GetInfoFailed;

   // Return name.
   BlockMoveData(*componentName, compressionName,
         GetHandleSize(componentName));

GetInfoFailed:
   DisposeHandle(componentName);
NewNameFailed:
FindComponentFailed:
   return (err);
}
```

It does this by examining the format of the sound to be played, and setting up the proper conversion steps needed to convert it to the hardware format. These steps might include decompression, sample size adjustment, sample rate conversion, volume adjustment, and mixing, all of which take time away from your application.

Therefore, the best way to maximize Sound Manager performance is to simply supply it with sounds that are already in the format required by the sound hardware. This way, the Sound Manager doesn't have to spend a lot of time processing, and your application will have more time to do other operations. Fortunately, Sound Manager

3.1 provides a new routine, SndGetInfo, that helps you determine the current sound hardware settings, so maximizing performance is a snap. (Of course, this technique applies only to sounds the application generates itself, since otherwise you have no control over their format.)

SndGetInfo is a selector-based routine that returns information about the sound channel. You pass in an OSType selector, and it returns a data structure of information. (This is similar to the operation of the SPBGetDeviceInfo routine in the Sound Input Manager, and in fact they use the same selectors.) Once you know the sound hardware sample rate, sample size, and number of channels, you know the kind of sounds that will be played back most efficiently.

The SoundSecrets application demonstrates how to determine the hardware settings and then find the sound with the correct format. It uses the GetHardwareSettings routine, which determines the hardware settings, and the FindMatchingSound routine, which chooses the right sound to play to maximize performance.

Listing 3 shows how to use SndGetInfo to return the current hardware settings.

Listing 3. Getting the current hardware settings

```
OSErr GetHardwareSettings(SndChannelPtr chan,
      SoundComponentData *hardwareInfo)
{
   OSErr err;

   err = SndGetInfo(chan, siNumberChannels, &hardwareInfo->numChannels);
   if (err != noErr)
      return (err);

   err = SndGetInfo(chan, siSampleRate, &hardwareInfo->sampleRate);
   if (err != noErr)
      return (err);

   err = SndGetInfo(chan, siSampleSize, &hardwareInfo->sampleSize);
   if (err != noErr)
      return (err);

   if (hardwareInfo->sampleSize == 8)
      hardwareInfo->format = kOffsetBinary;
   else
      hardwareInfo->format = kTwosComplement;

   return (noErr);
}
```

PUMP UP THE VOLUME

Most sound programmers have heard (literally) about the venerable ampCmd command, which lets you scale the volume of all sounds on a channel from a minimum of 0 (silence) to 255 (full volume). However, only the truly righteous know that Sound Manager 3.0 added an even more powerful command for manipulating sound volume — volumeCmd.

The volumeCmd command does three things. First, like ampCmd, it allows you to scale the volume from silence to full volume. However, volumeCmd doesn't stop there; like that revolutionary amplifier in the movie *Spinal Tap* that could go all the way to 11, it lets you go beyond full volume to overdrive the sound volume. And finally, it allows you to control the volume of the left and right channels independently, providing complete stereo control over your sounds.

All this is possible because the volumeCmd command represents the sound volume in 16-bit fixed-point notation. By using the most significant 8 bits to represent the integer portion of the volume and the least significant 8 bits for the fractional portion, it provides very precise volume settings. And overdriving the sound is a cinch. By combining the left and right volume settings into one 32-bit quantity, volumeCmd gives you full control over how loud you can blast your speakers. Another command, getVolumeCmd, returns the current volume setting, in case you forgot what you set it to.

> **A new interaction between the volumeCmd and ampCmd** commands was added in Sound Manager 3.1. Previously, ampCmd would clobber the separate left and right settings made by volumeCmd, setting them to the same value. Starting with Sound Manager 3.1, volumeCmd now specifies a base volume for a channel, and ampCmd scales against that base, which lets ampCmd and volumeCmd coexist better when playing the system alert beep.•

Table 1 gives some examples of values you can pass to volumeCmd and their effect. Remember, once you've changed the volume setting with volumeCmd, the setting is applied immediately to the current sound that's playing (if any) and to every subsequent sound played on that channel.

Table 1. Sample values for volumeCmd

volumeCmd	**Right Channel**	**Left Channel**	
Setting	**Decimal Value**	**Decimal Value**	**Effect**
0x01000100	1.0	1.0	Full volume out both channels (the default)
0x00000000	0.0	0.0	Silence out both channels
0x01000000	1.0	0.0	Full volume out right channel; silence out left
0x00000100	0.0	1.0	Silence out right channel; full volume out left
0x02000200	2.0	2.0	Double the full volume out both channels
0x01800040	1.5	0.25	One and a half times full volume out right channel; one quarter out left

The SoundSecrets sample program included on the CD demonstrates the usefulness of volumeCmd by providing a slider control to adjust left and right volume separately, with volume overdrive up to two times the normal full volume.

ACHIEVE PERFECT PITCH

One of the trickiest things to do with the Sound Manager is to play a sound at just the right pitch. While the frequencyCmd command lets you trigger a sound at a given MIDI note value, and the rateCmd command gives you limited control over the pitch of the sound currently playing, before Sound Manager 3.1 there was no good way to just play a sound at an arbitrary pitch, short of generating the samples yourself. So Sound Manager 3.1 introduced the rateMultiplierCmd command, which gives you perfect pitch every time.

The concept behind rateMultiplierCmd is very simple. Using a Fixed value, you can apply a multiplier to the playback rate of all sounds played on a channel. This allows you to vary the sample rate of the sound being played, and thus control its pitch. (Of course, changing the rate also changes the duration of the sound.) You can use getRateMultiplierCmd to return the current rate multiplier setting.

Like any great concept, it's most easily understood with an example, so Table 2 gives some values you can pass to rateMultiplierCmd and their effect. Remember, as with volumeCmd, once you change the rate multiplier with this command, the setting is applied immediately to the current sound that's playing (if any) and to every subsequent sound played on that channel. Our helpful SoundSecrets application demonstrates the rateMultiplierCmd command with a slider control to adjust the playback rate of the sound from 0.0 to 2.0.

Table 2. Sample values for rateMultiplierCmd

rateMultiplierCmd Setting	**Decimal Value**	**Effect**
0x00010000	1.0	Play sounds at the normal pitch setting (the default)
0x00020000	2.0	Play sounds at a pitch shifted up one octave
0x00008000	0.5	Play sounds at a pitch shifted down one octave
0x00018000	1.5	Play sounds at a pitch shifted up half an octave
0x00000000	0.0	Repeat the last audio sample indefinitely, which effectively pauses playback on this channel

PLAYING SOUND THE QUICKTIME WAY

Something that vexes nearly everyone using the Sound Manager is attempting to play continuous sound. Many applications break sounds up into chunks as they're read off the disk, and most games have background music that's continuously generated and mixed with sound effects. After spelunking through *Inside Macintosh: Sound*, you'll eventually come across the SndPlayDoubleBuffer routine, which looks like the answer to your prayers. However, SndPlayDoubleBuffer has some serious limitations that you need to consider.

First of all, SndPlayDoubleBuffer ping-pongs between just two buffers, and the location of those buffers can't be changed once the sound is started, which can be really inconvenient when you're trying to piece together a lot of sound buffers off the disk. In addition, the format of the sound being played can't be changed once the sound is started, and the headers describing the sound must be attached to the sound data itself.

There has got be a better way, right? Well, QuickTime uses a strategy involving sound callbacks that's much more flexible and doesn't make you scratch your head over when to use that lastBuffer flag in SndPlayDoubleBuffer. Once you read about the QuickTime way, you'll probably want to use it too.

With the QuickTime strategy you trigger all your sounds with a plain old bufferCmd command, and set up callBackCmd to call you when that buffer is done playing. This has two big advantages:

- Because bufferCmd takes a pointer to a sound header as its only parameter, you can queue up a different buffer for every callback if you want, freeing you from that pesky two-buffer limit.
- Because the sound header records contain a pointer to the audio data, you have a lot more flexibility in buffer management, and you can dynamically adjust the buffer sizes to any values that make sense to you.

This technique is demonstrated by Listing 4, taken from the SoundSecrets application on the CD. Basically, the interrupt routine just plays the next buffer and then queues up a callback, which keeps the sound playing continuously. The application has a slider that lets you adjust the size of the buffer dynamically.

Listing 4. Playing continuous sound

```
// Issue bufferCmd to play the sound, using SndDoImmediate.
sndCmd.cmd = bufferCmd;
sndCmd.param1 = 0;
sndCmd.param2 = (long) &globals->sndHeader;

err = SndDoImmediate(globals->sndChannel, &sndCmd);
if (err != noErr)
   return (err);

// Issue callBackCmd using SndDoCommand so that we get called back
// when the buffer is done playing.
sndCmd.cmd = callBackCmd;
sndCmd.param1 = 0;
sndCmd.param2 = (long) globals;

err = SndDoCommand(globals->sndChannel, &sndCmd, true);
if (err != noErr)
   return (err);
```

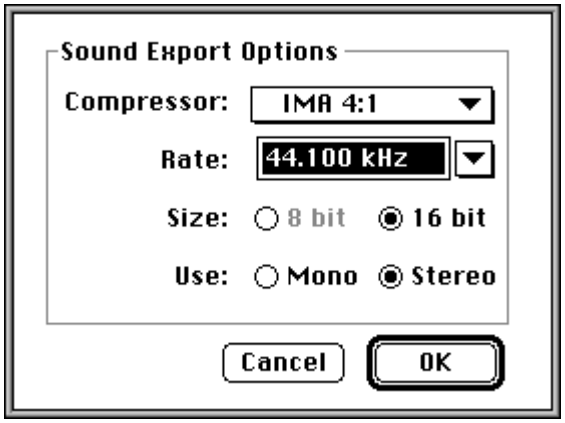
Remember, callBackCmd calls your application at interrupt time, so it's up to you to set up your A5 world if you want to use globals. You can't call Toolbox routines like those in the Memory Manager from within the callback; however, you can call most Sound Manager routines (see *Inside Macintosh: Sound* for information on individual routines). To make things easier, you can pass an application-defined value to the callback routine in param2 of callBackCmd. Also, to ensure correct queue processing, it's very important that you use SndDoImmediate to send bufferCmd, and SndDoCommand to send callBackCmd.

COMPRESS WITH THE BEST

While Sound Manager 3.0 included an architecture for decompressing arbitrary sounds (described in the article "Make Your Own Sound Components" in *develop* Issue 20), no method was provided to compress sounds. However, with the arrival of Sound Manager 3.1 and QuickTime 2.1, creating compressed sound files became as easy as opening a movie.

The compression technique demonstrated here uses the import/export facility built into QuickTime. Movie import components allow you to convert other files into QuickTime movies, while movie export components let you save QuickTime movies in other formats. QuickTime 2.1 provides an export component that works with Sound Manager 3.1 to let you save the audio in a QuickTime movie to an AIFF file in any format you please.

QuickTime does this by calling the Sound Manager to mix all the tracks together, converting them to the sample rate and size you specify, and even compressing the data with any of the compression algorithms provided by Sound Manager 3.1. The resulting AIFF file can then be played by any other Sound Manager routine, or converted back into a movie. The export component provides a dialog to let the user select the sample rate, sample size, and compression format of the AIFF file, as shown in Figure 1.



Sound Export Options

Compressor: IMA 4:1

Rate: 44.100 kHz

Size: ○ 8 bit ● 16 bit

Use: ○ Mono ● Stereo

Figure 1. Sound Export Options dialog

Listing 5 demonstrates the process of converting a movie to an AIFF file, displaying the Sound Export Options dialog to let the user control the conversion process. The SetMovieProgressProc routine displays a progress dialog while the movie is being converted. The code is taken from ExportAIFF on this issue's CD.

Listing 5. Converting a movie to an AIFF file

```
OSErr ConvertMovieToAIFF(FSSpec *inputFile, FSSpec *outputFile)
{
   short    fRef;
   Movie    theMovie;
   OSErr    err;

   err = OpenMovieFile(inputFile, &fRef, fsRdPerm);
   if (err != noErr)
      goto OpenMovieFileFailed;

   err = NewMovieFromFile(&theMovie, fRef, nil, nil, 0, nil);
   if (err != noErr)
      goto NewMovieFromFileFailed;
```

(continued on next page)

Listing 5. Converting a movie to an AIFF file *(continued)*

```
   SetMovieProgressProc(theMovie, (MovieProgressUPP) -1L, 0);

   err = ConvertMovieToFile(theMovie, nil, outputFile, 'AIFF', 'sSnd',
         0, nil, showUserSettingsDialog, nil);

   DisposeMovie(theMovie);

NewMovieFromFileFailed:
   CloseMovieFile(fRef);
OpenMovieFileFailed:
   return (err);
}
```

SOUNDING OFF

Now that this article has revealed some of the best-kept secrets of the Sound Manager, you can go out and create great applications on your own. Consider all your new skills — parsing and displaying sound resources, improving playback performance, adjusting volume and pitch, playing continuous sounds, and compressing audio. Now that the Sound Manager is your friend, you can focus on making your applications insanely great, instead of having the Sound Manager drive you insane!

RELATED READING

- *Inside Macintosh: Sound* (Addison-Wesley, 1994).
- Macintosh Technical Note “GetCompressionInfo()” (SD 1).
- “Make Your Own Sound Components” by Kip Olson, *develop* Issue 20.

Thanks to our technical reviewers Bob Aron, Peter Hoddie, Kevin Mellander, and Jim Reekes.•

DAVE EVANS

BALANCE OF POWER

Advanced Performance Profiling

There's little that compares to diving headfirst toward the ground at 120 miles per hour. I may have been going even faster when I last went skydiving. Tucking my arms in tightly, with my head back and legs even, I heard a deafening roar from the wind as I sped toward terminal velocity. "Terminal" would have been a good word for the situation if it weren't for the advances that have been made in parachute technology.

Parachutes have come a long way since their debut, when they were billowy round disks of silk sewn with simple cords stretching to a harness. They were greatly improved when the square parachute was invented thirty years ago. The square parachutes look like an airplane's wing, and they create lift in much the same way. Until recently, however, square parachutes weren't improved upon much. Perhaps their superiority over round parachutes left everyone satiated. That lack of progress was unfortunate; if recent improvements — like many-celled parachutes and automatic activation devices — had been pursued many years ago, skydiving would be even safer today.

The moral from this is to question satisfaction, and that will be our mantra for this column. In particular, I want you to question the performance gains you've seen by moving to native PowerPC code. In this column we'll look at improved tools for examining PowerPC code performance, and you'll see how such questioning can really enlighten you.

ILLUSIONS

The PowerPC processors can issue multiple instructions at once. You therefore may think they'll tear through your code, executing many instructions per cycle. While this is sometimes true, a number of hurdles keep the PowerPC processors from completing even one instruction per cycle. These hurdles include instruction cache misses, data cache misses, and processor pipeline stalls.

What may surprise you is how often the processor sits idle because of these hurdles. I did some tests and found that while opening new windows in one popular application, a Power Macintosh 8500's processor completed an average of only one instruction for every two cycles. This is not very efficient, considering its PowerPC 604 processor can complete up to four instructions per cycle.

Much of that inefficiency is from instruction and data cache misses. As PowerPC processors reach faster clock rates, these cache misses will have an increasing impact. By minimizing cache misses we could realize a significant performance improvement.

Simply recompiling your 680x0 code to native PowerPC code doesn't typically generate efficient code. Many designs and data structures for the 680x0 architecture work very poorly when ported to PowerPC code. When you port native, you should carefully examine your code. Tuning for a cached RISC architecture is very different than for the 680x0 family. Here are some important things to consider:

- Redesign your data structures. Use long word–sized elements. Keep commonly used elements together, and keep everything aligned on double long word boundaries.
- Keep results in local variables, instead of recomputing or calling subroutines to retrieve global variables.

BETTER PROFILING

Until recently you couldn't measure cache misses unless you had a logic analyzer or other expensive hardware. The PowerPC 604 processor, however, includes an extremely useful performance measurement feature: two special registers (plus a register to control them) that can count most events that occur in the processor. Each of these registers can count about 20 events, and there are five basic events that both registers can count.

Here are just a few examples of what you can count with these registers: integer instructions that have completed; mispredicted branch instructions; data

DAVE EVANS likes to go skydiving when he can get away from his job gluing together the Mac OS software at Apple. He has gone a few times now, but he'll always cherish the memory of his first jump. Friends on the ground that day claim to have clearly heard his scream, although he was nearly a mile above them when he left the plane. On his second leap, if he hadn't opened the chute while upside down and then watched it deploy through his legs, he might have noticed more of the surrounding countryside.•

cache misses; and floating-point instructions that have been issued.

To use the performance profiling that the PowerPC 604 processor provides, you'll need to have one of the newer Macintosh models that include this processor, such as the Power Macintosh 9500 or 8500. This will cost less than a logic analyzer yet allow you to get detailed performance profiles.

Although these registers will show your software's performance only on a 604-based Power Macintosh, your software's cache usage and efficiency should be similar on other PowerPC processors. Use the 604's special abilities to profile your code and you'll benefit on all Power Macintosh models.

For more accurate performance measurements, you may want to use the DR Emulator control panel, which is provided on this issue's CD. With this control panel you can turn off the dynamic recompilation feature of the new emulator; this feaure, which is described in the Balance of Power column in Issue 23, can affect the performance of your tests over time.

Also provided on the CD is the POWER Emulator control panel. This control panel lets you turn off the Mac OS support for RS/6000 POWER instructions and thus check for these instructions in your code (they'll cause a crash).•

THE 4PM PERFORMANCE TOOL

To use the new 604 performance registers, you don't need to program in PowerPC assembly language. On this issue's CD we've included a prototype application called 4PM. This tool, which was developed by engineer Tom Adams in Apple's Performance Evaluation Group, uses the PowerPC 604–specific registers to provide various types of performance data.

4PM is very simple to use. It presents three key menus: Control, Config, and Tests, as shown in Figure 1. You use these menus to select the type of performance measurement and an application you'd like to run the tests on. The application you're testing is launched by 4PM, and you can gather data either continuously or, using a "hot key," exactly when you want.

Once a test completes, 4PM fills a window with the results — a tabular summary with a different test run on each line. The Save command in the File menu will write the results to a file of type 'TEXT'.

The Control menu. Use the Launch command in this menu to select an application and run it, gathering the test data specified with the Config or Tests menu. The default configuration will measure cycles and instructions completed between when the application launches and when it quits. The Launch Again command simply relaunches the last application you tested.

Check Use Hotkey if you'd like to control exactly when data is gathered. With this option, you start and stop collecting data by holding down the Command key while pressing the Power key. (This key combination is the same way to force entry to MacsBug, which you'll be unable to do during the tests.)

The Repeats command is just a shortcut that's handy if you're repeating a test multiple times. If you specify a repeat value with this command, your test application will be relaunched that many times after you quit it.

The Intervals command allows you to collect data points at regular intervals; a dialog box offers the choices 10 milliseconds, 100 milliseconds, 1 second, or Other. Normally just a total is collected, but by specifying an interval time you'll instead receive a spreadsheet of timings. This will show what your code's performance was as the test progressed.

The Config menu. The commands in the Config menu allow you to tailor the test data by specifying exactly which events each register will count. The Count Select command lets you specify the machine states to collect data in; set this to "User Only" since you'll be tuning application code.

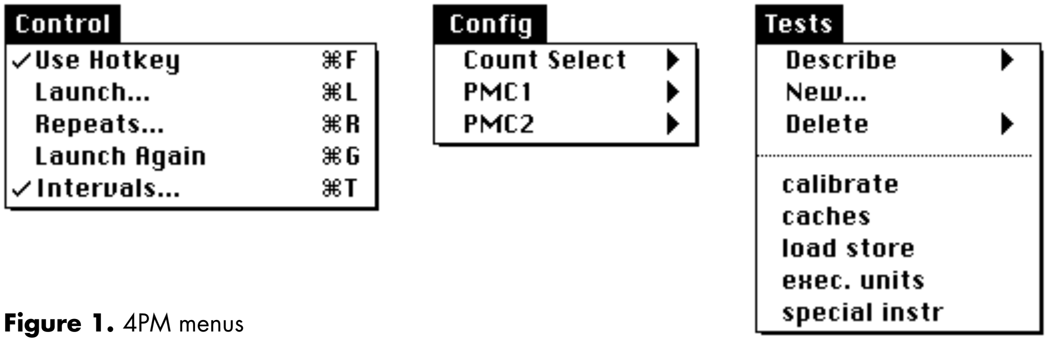


Figure 1. 4PM menus

The Tests menu. The commands in the Tests menu are for generating typical reports. Use the calibrate command to count the five basic events that are common to both 604 performance registers, including cycles and instructions completed; with this test selected, the Launch command will run your application five times, successively counting each of these events. You can use one of the remaining tests to collect more specific measurements. The caches, load/store, execution units, and special instructions tests each generate a report for the corresponding aspect of 604 performance. The Describe command displays a window describing which events are counted in the selected test. Use the New command to create your own tests. These new tests are automatically saved; you can use the Delete command to remove any that you've added.

ASSEMBLY USAGE

If you want finer results, you should read and write to the 604 performance registers directly. This requires writing in PowerPC assembly language, but it allows you complete control over what data you'll collect for your time-critical code.

You'll be accessing three new special-purpose registers: MMCR0, PMC1, and PMC2. MMCR0 controls which events will be recorded and when exactly to record. The performance monitor counter registers, PMC1 and PMC2, are the registers in which you'll read the results. I'll give a brief summary of how to use these registers, but you'll need to read Chapter 9 of the *PowerPC 604 RISC Microprocessor User's Manual* for details.

MMCR0 is a 32-bit register that specifies all the options for performance measurement. Most of these options aren't important to your application profiling, and you should at first leave the high 19 bits of MMCR0 set to 0. The low 13 bits, however, specify which events you want counted in PMC1 and PMC2. Bits 19 through 25 select PMC1, and bits 26 through 31 select PMC2. See Chapter 9 of the 604 user's manual to learn which specific bits to set.

Here's an example of how to measure data cache misses per instruction:

```
.eq PMC1_InstructionsCompleted  2 << 6
.eq PMC2_DataCacheMisses        6
.eq MMCR0_StopAllRecording      $80000000

    li      r0, MMCR0_StopAllRecording
    mtspr   MMCR0, r0   ; stop all recording
    li      r0, 0
    mtspr   PMC1, r0    ; zero PMC1
    mtspr   PMC2, r0    ; zero PMC2
    li      r0, PMC1_InstructionsCompleted +
                PMC2_DataCacheMisses
    mtspr   MMCR0, r0   ; start recording
```

Notice that we load MMCR0 with only the most significant bit set to turn off all recording. This holds PMC1 and PMC2 at their current values and allows us to also zero PMC1 and PMC2 before we start recording. When you're done measuring, follow with this code:

```
    li      r0, MMCR0_StopAllRecording
    mtspr   MMCR0, r0   ; stop all recording
    mfspr   PMC1, r3    ; r3 is number of
                        ; instructions completed
    mfspr   PMC2, r4    ; r4 is data cache misses
```

Notice again that we turn off recording before reading the results. Otherwise the very act of reading the registers would affect the results; it will slow your code slightly, since the **mtspr** and **mfspr** instructions take multiple cycles to complete.

Don't record over very long periods of time, because the PMC1 and PMC2 registers can overflow. To measure over long periods, you should periodically read from the registers, add the result to a 64-bit number in memory, and clear the registers to prevent this overflow.

Don't ship any products that rely on these performance registers. They're supported only in the current 604 processor, and they're not part of the PowerPC architecture specification.

COMPLACENCY

The moral is the same as for my tale of the square parachutes: question satisfaction. Don't become complacent about the performance of your new native PowerPC applications. The profiling tools described here should help you more accurately measure and identify bottlenecks in your PowerPC code. Use that information to tune — especially paying attention to memory usage — and you'll be surprised how much faster your product will run. Macintosh users consistently hunger for faster computers and more responsive software; spend some serious time tuning, and they'll thank you for it.

Thanks to Tom Adams, Geoff Chatterton, Mike Crawford, and Dave Lyons for reviewing this column.•

Guidelines for Effective Alerts

This article expands on the Macintosh Human Interface Guidelines for making attractive, helpful alerts (and dialogs) with a standard appearance and behavior. Standardization is important, because the more familiar an alert looks to users, the more easily they can concentrate on the message. Using the Finder as a source of good alerts, we provide examples of different alert types and discuss how to make alerts user-friendly.

PAIGE K. PARSONS

Alerts are an in-your-face way of getting the user's attention. It's hard for a user to ignore alerts because they block all other input to the application until the user dismisses them. These little windows are powerful stuff. When used correctly, alerts are a helpful way to inform the user of a serious condition that requires immediate attention. When used incorrectly or capriciously, alerts are annoying and disruptive; since they must constantly be swatted out of the way, their content is often ignored.

This article discusses when to use alerts, describes the different types of alerts, and gives tips for designing alert boxes. It elaborates and expands on alert guidelines in the *Macintosh Human Interface Guidelines*. At the end of the article, you're encouraged to try your hand at evaluating some real-life alerts.

Though not implemented as such in the system, alert boxes are essentially a type of modal dialog box. This article focuses on alerts, but the guidelines can be applied to other dialog boxes as well. We specifically cover status dialogs here because there are guidelines that are unique to that type of dialog.

For information on implementing alerts and dialogs in your application, see *Inside Macintosh: Macintosh Toolbox Essentials.*•

ALERTS IN GENERAL

Alerts provide information about error conditions and warn users about potentially hazardous situations. They should be used only when the user's participation is essential; in all other cases, try using another mechanism to get your point across. For example, consider an error or output log if the messages are something that the user may want to save.

PAIGE K. PARSONS (parsons@apple.com) is a Human Interface Specialist at Apple. For two years she worked on the user interface of the Apple Dylan Development Environment. She recently began working at Apple's Human Interface Design Center, where she is responsible for software user interface issues in the PowerBook division. Favorite diversions include maintaining a Web site for the House Rabbit Society (http://www.psg.lcs.mit.edu/~carl/paige/HRS-home.html) and trolling used record shops in Berkeley for vintage vinyl.•

The *Macintosh Human Interface Guidelines* haven't caught up yet with the main recommendation in this article: that alerts be movable and application modal. The current interface guidelines and system software don't allow alerts to be movable, but this may change in future versions of the Mac OS. Until then, you can implement your alerts as movable modal dialogs.

Making alerts movable is helpful in case an alert is covering information on the screen that the user would like to see before responding to the alert. Another advantage to movable alerts is that they have a title, which gives the user a context for the error.

Application modal means the alert is modal in the current application only: the user can't interact with this application while the alert is on the screen, but can switch to another application. This is especially useful when the user needs to get information from another application in order to respond to the alert. (*System modal*, on the other hand, means the user can't interact with the system at all except within the alert box.)

TYPES OF ALERTS

Alerts come in three varieties, each of which is geared to a different situation. This section provides a few examples of each type, and also takes a look at status dialogs.

NOW HEAR THIS: NOTE ALERTS

A *note alert* simply conveys information, informing the user about a situation that has no drastic effects and requires no further action. For example, if a user selects a word and executes a spell check, an alert saying that the word is spelled correctly would be a note alert. Rather than provide a smorgasbord of options, a note alert contains a single button to dismiss the alert.

Don't use an alert to signify completion of a task; use alerts only for situations that require the user to acknowledge what has occurred. For example, the following note alerts are inappropriate and get in the user's way:

- The Trash has finished emptying.
- The 3,432 files you selected have been copied.

WATCH OUT! CAUTION ALERTS

Caution alerts warn users of potentially dangerous or unexpected situations. You should use them, for example, to be sure the user wants to proceed with a task that might have undesirable results. In this case the alert normally contains only two buttons — one that cancels the operation and one that confirms it. Here are two caution alert messages:

- An item named "READ ME" already exists in this location. Do you want to replace it with the one you're moving?
- The Trash contains 1 item. It uses 102K of disk space. Are you sure you want to permanently remove it?

Don't use alerts to confirm operations that would cause only a minor inconvenience if performed by mistake. Here are two examples of unnecessary caution alerts:

- Do you really want to eject the disk "Installer"?
- Do you really want to duplicate the selected item?

Caution alerts are also used when an unexpected situation occurs and the user needs to decide what to do next. The following examples contain only two buttons, for canceling or confirming the operation, but such an alert may present several choices if appropriate.

- The document “Calendar” is locked, so you will not be able to save any changes. Do you want to open it anyway?
- The item “Calendar” could not be deleted, because it contains items that are in use. Do you want to continue?

Before deciding to use this type of alert, double-check to see if it’s really needed; superfluous alerts are a bad idea because users will get in the habit of dismissing alerts and possibly let an important one go by. It’s better to have a user make choices with commands instead of alerts. For example, the Finder has separate Shut Down and Restart commands (in its Special menu) instead of having only a Shut Down command with an alert asking “Restart after shutting down?”

HOLD IT: STOP ALERTS

Use a *stop alert* when calling attention to a serious problem that prevents an action from being completed. They typically have only one button, to dismiss the alert. Here are two good examples of stop alerts:

- You cannot copy “Calendar” onto the shared disk “Zippy” because the disk is locked.
- The alias “Calendar” could not be opened, because the original item could not be found.

It’s especially important in stop alerts to give enough detail about the problem to help the user prevent it in the future. The following alert message doesn’t convey much useful information:

- You cannot rename the item “Zowie”.

This alternative is more helpful:

- The name “Zowie” is already taken. Please use a different name.

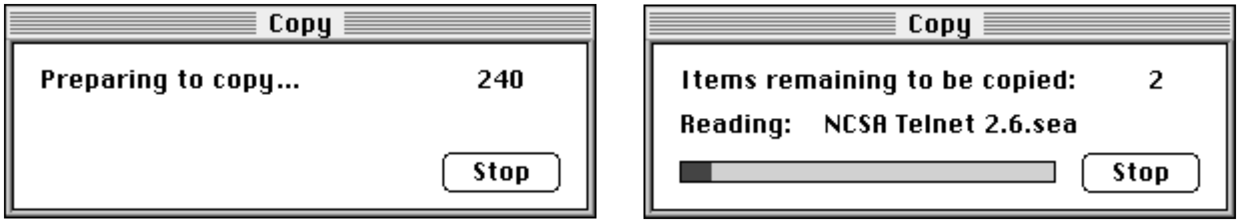
Similarly, if the chosen name is too long, it’s more helpful for the message to state the maximum number of characters a filename can have.

EVERYTHING IS OK: STATUS DIALOGS

Status dialogs inform the user when an application is busy and the user cannot continue working in the application until the operation finishes. In the Finder, these operations include copying, moving, and deleting files. Status dialogs should be displayed whenever the application is busy for more than about five seconds (unless posting and updating the dialog would take most of that time). During this time the application should also change the pointer to the standard wristwatch.

A status dialog differs from an alert in that the user doesn’t need to explicitly dismiss the dialog; it goes away on its own once the task has completed. The dialog should contain a message that describes the status of the operation and a progress indicator to show how much of the job has been completed. A status dialog may change messages depending on the stage of operation. Figure 1 shows a status dialog at two stages of a copy operation.

A sense of completion is important, so the application should be sure not to remove the status dialog until the progress indicator shows that the operation is done (such as by completely filling up the status bar in the example in Figure 1).


Copy

Preparing to copy... 240

Copy

Items remaining to be copied: 2

Reading: NCSA Telnet 2.6.sea

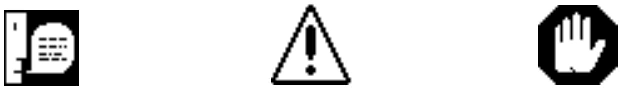
Before copying

During copying

Figure 1. Status dialog during a copy operation

ICONS IN ALERT BOXES

Alert boxes always contain an icon that identifies the type of alert, as shown in Figure 2. (Status dialogs contain no icon.) If you implement your alerts as movable modal dialogs, there's no Toolbox infrastructure set up for getting the correct icon automatically, so you'll need to remember which one to use.


Note alert

Caution alert

Stop alert

Figure 2. Icons for specific alert types

> **A note on OpenDoc and alert icons:** OpenDoc part editors aren't as visible to the user as today's applications, but at times it may be important for users to make the connection between a running editor and its stored representation on disk. One such time is when the editor is reporting an error about itself, such as an incompatible version; for these errors, the alert should contain the icon of the editor instead of a note, caution, or stop icon.•

WRITING ALERT MESSAGES

The alert message is the most important component of an alert. You want users to read and respond to your alerts easily and then continue smoothly with their work. This section gives tips on structure, content, tone, and other important factors in writing effective alert messages.

SITUATION, REASON, SOLUTION

Every alert message should start by describing the situation that led to the alert, letting the user know what's wrong. This is usually followed by the reason the problem occurred and a proposed solution to the problem.

When describing the situation that caused an alert, be as specific as possible, to help the user understand the problem.

Giving the reason the alert occurred is especially helpful when the application can't do something because it's dependent on some other operation that it can't control. For example, compare these messages:

- The alias "Warne" could not be opened.
- The alias "Warne" could not be opened because the shared disk "Beatrix" could not be found on the network.

The first message doesn't give the user any information about why the problem occurred. Is the application that created the document missing? Is the file corrupted? The second message is much better because it tells the user why the operation could not be completed.

Whenever possible, alerts should indicate a solution for the user. Users become extremely frustrated when an alert says something is wrong but doesn't offer a remedy to the problem. Even worse is an alert that tells the user something is wrong when the application could have fixed the problem itself. The following would be a bad message because the Finder is capable of quitting all the applications on its own:

- You must quit all running applications before shutting down your Macintosh.

In cases where the application can perform the action itself, consider whether doing so may surprise the user; if so, presenting a caution alert may be more appropriate. For example, if the user attempts to shut down a Macintosh while other users have it mounted as a server, the Finder could just disconnect the other users automatically; however, in this case it's more helpful to present an alert confirming the shutdown.

BE CONSISTENT

Be sure your alert messages are consistent in tone, content, and structure with each other as well as with other messages your software presents to the user. Are your application's alerts consistent with its status messages, for example? Do all your alerts refer to the application in a consistent manner? Users pick up on small inconsistencies, and even subtle differences can cause confusion.

BE BRIEF

Alert messages should be brief and to the point, to keep the user's attention. If you need to give a lot of information, consider writing it to an error log or providing a brief message in the alert along with a button to get to the application's help system.

If you absolutely have to put a long message in an alert, keep in mind that many people have PowerBook computers or "classic" Macintosh computers with small screens. A good rule of thumb is that an alert message must consist of no more than 150 characters to fit on a small screen. Also note that translation from English to other languages tends to expand the length of the message. Even translations into languages that use Roman characters can cause the message length to double or triple in size.

BE ENCOURAGING

Use a positive and constructive tone. After encountering a problem and being presented with an alert, the last thing the user wants is an overly negative response from the application.

Avoid assigning blame or offending users. Don't accuse them of doing something wrong or stupid. Instead, give the reason an action cannot be performed, or offer to perform the action. Which message would you rather see?

- You forgot to save your changes!
- Save changes to access privileges for "Zippy"?

PHRASING AND TERMINOLOGY

Don't use double negatives, such as "No items are not used." They're difficult for users to understand and just bad English. Double negatives can be especially confusing when combined with a Cancel button; the user rarely gets the expected outcome.

Keep the situation and action in the present. This is clearer and usually requires fewer words. For example, compare these two messages:

- An item named “READ ME” already existed in this location. Did you want to replace it with the one you moved?
- An item named “READ ME” already exists in this location. Do you want to replace it with the one you’re moving?

If there’s an implied subject of a message, it should be the application. For example, if the user tries to open a document that the application can’t open (as when it runs out of memory), the alert message might begin “Cannot open document.” Messages in which the user or some other noun could be the implied subject are more likely to be confusing — for example, “Have exceeded allotted network time. Try again later.”

Use terms that are familiar to the user. This often means avoiding computer jargon at all costs. Remember, terms that seem common to you may be unfamiliar to many Macintosh users. It depends on what type of user will be working with your application. For example, the expression *establishing a connection* may be clearer than *handshaking* to many users.

Use *invalid* instead of *illegal*. The user hasn’t broken the law, but has simply given the application some information that it can’t handle.

PUNCTUATION AND CAPITALIZATION

Alert messages should always be complete sentences, beginning with a capital letter and ending with a period or question mark. The closing punctuation gives a sense of completion and lets the user know that the message hasn’t been truncated.

Don’t use colons when requesting that the user supply information; instead, use a period. This makes your alerts consistent with other dialogs and user interface elements in the system software.

Use an apostrophe (’), typed with Option-Shift-], rather than a single straight quotation mark ('), and use curly (“ ”) rather than straight (") double quotation marks — that is, Option-[and Option-Shift-[, rather than Shift-'.

Use double quotation marks around any names in the message that are variable, such as names of documents, folders, and search strings. This lets the user know exactly what part of the message is the name. Remember that Macintosh filenames can contain spaces, which can make things really confusing without the quotes. Commas, periods, and other punctuation characters should be placed outside the quotation marks:

- You cannot duplicate the shared disk “Warne”, because the disk is locked.

Never use an exclamation point or all uppercase letters. It makes users feel as if they’re being shouted at, as in this example:

- Revert to the saved version of “Map”? WARNING! All changes will be lost!

STATUS MESSAGES

In status dialogs, use an ellipsis (Option-semicolon, a character that looks like three periods) to indicate that an intermediate process is under way:

- Preparing to copy…
- Scanning “My Document”…

For describing the status of a task, the terms *canceled*, *failed*, *in progress*, and *complete* are good choices. Avoid computer jargon such as *aborted*, *killed*, *died*, or *ack'ed*.

ALERT TITLES

Every movable alert should have an informative title, to provide a context for the alert. Users may be working on several tasks at the same time and may not remember what action generated the alert. A well-chosen title helps the user figure out not only which application caused the alert to appear, but also which action.

The title of the alert should be the same as, or closely related to, the command or action that generated the alert. (If the command has an ellipsis in it, don't include the ellipsis in the alert title.) For example, when a user copies or duplicates an item in the Finder, the associated status dialog has the title "Copy"; when the user chooses Empty Trash, the title of the Finder's status dialog is "Trash."

Like menu commands, alert titles are capitalized like book titles. Capitalize every word except articles (*a*, *an*, *the*), coordinating conjunctions (for example, *and*, *or*), and prepositions of three or fewer characters (except when the preposition is part of a verb phrase, as in "Turn Off").

ALERT BUTTONS

Alerts contain buttons that dismiss the alert or allow the user to make choices regarding how to proceed. The standard button height is 20 pixels.

Try to limit the number of buttons that appear in an alert. The more buttons, the more difficult it is for the user to decide which is the "right" option. In addition, screen size often limits the number of buttons. As a general rule, about three buttons of ten or fewer characters will fit on a small screen. Button names should be simple, concise, and unambiguous.

Capitalize button names like book titles (for example, Connect to Server). Never capitalize all letters in the name (except for the OK button, which should always be named OK and never ok, Ok, Okay, okay, OKAY, or any other strange variation).

On the Macintosh, ellipses are used after command names when the user needs to provide additional information to complete the command. An ellipsis after a button name indicates that the button leads to other dialogs, a rare but occasional occurrence.

THE ACTION BUTTON

Alert boxes that provide the user with a choice should be worded as a short question to which there is an unambiguous, affirmative response. The button for this affirmative response is called the *action button*.

Whenever possible, label the action button with the action that it performs. Button names such as Save, Quit, or Erase Disk allow experienced users to click the correct button without reading the text of a familiar dialog. These labels are often clearer than words like OK or Yes. Phrase the question to match the action that the user is trying to perform.

If the action can't be condensed conveniently into a word or two, use OK. Also use OK when the alert is simply giving the user information without providing any choices.

THE CANCEL BUTTON

Whenever possible, caution alerts should provide a button that allows the user to back out of the operation that caused the alert. This button should be labeled "Cancel" so that users can easily identify the safe escape hatch. Cancel means "dismiss this operation with no side effects"; it doesn't mean "done with the alert," "stop no matter what," or anything else. Pressing Command-period or the Escape key should have the same effect as clicking the Cancel button.

Don't label the button Cancel when it's impossible to return to the state that existed before an operation began; instead, use Stop. Stop halts the operation before its normal completion, accepting the possible side effects. Stop may leave the results of partially completed tasks around, but Cancel never does. For example, a Cancel button would be inappropriate for a copy operation in which some of the items may have already been copied. Figure 1 (earlier in this article) illustrates using Stop in a status dialog for a copy operation.

THE DEFAULT BUTTON

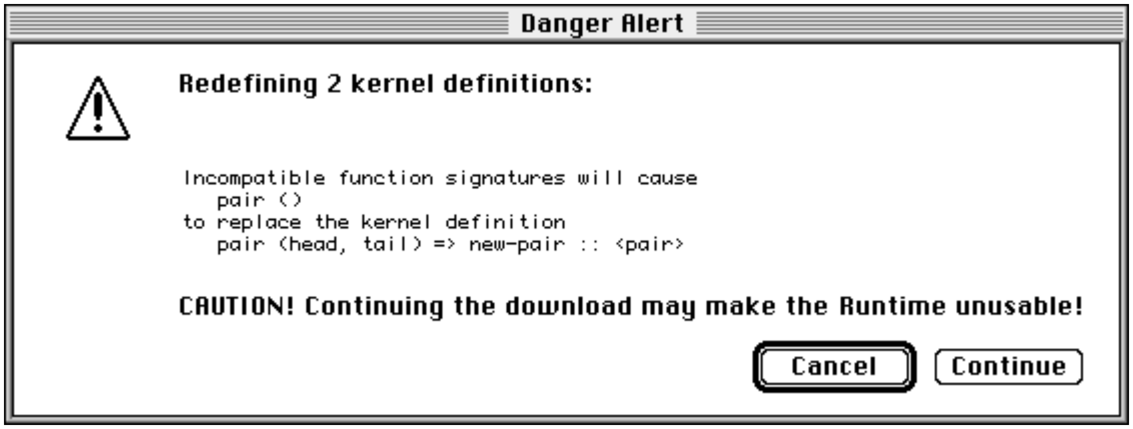
The default button represents the action performed when the user presses the Return or Enter key. This button should perform the most likely action (if that can be determined). In most cases, this means completing the action that the user started, so the default button is usually the same as the action button.

The default button's distinctive bold outline appears automatically around the default button in alerts, but remember that in dialog boxes you need to outline the button yourself.•

If the most likely action is dangerous (for example, it erases the hard disk), the default should be a safe button, typically the Cancel button. If none of the choices are dangerous and there isn't a likely choice, there should be no default button; by requiring users to select a button explicitly, you protect them from accidentally damaging their work by pressing the Return or Enter key out of habit.

POP QUIZ

Now, for a bit of fun. I've been collecting some alerts that need improvement (Figures 3, 4, and 5). Based on the information in this article, can you find the flaws in each, and suggest improvements?



Redefining 2 kernel definitions:

```
Incompatible function signatures will cause
   pair ()
to replace the kernel definition
   pair (head, tail) => new-pair :: <pair>
```

CAUTION! Continuing the download may make the Runtime unusable!

Figure 3. Poorly designed "danger alert"

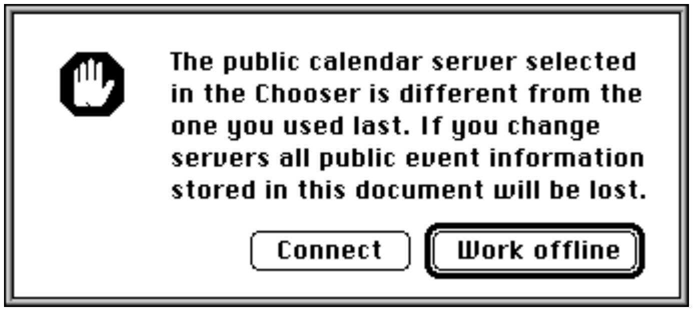

The public calendar server selected in the Chooser is different from the one you used last. If you change servers all public event information stored in this document will be lost.

Figure 4. Poorly designed "server alert"

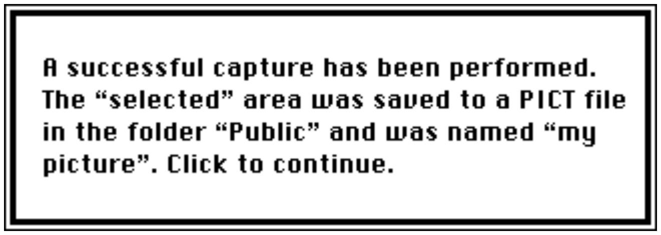

A successful capture has been performed. The "selected" area was saved to a PICT file in the folder "Public" and was named "my picture". Click to continue.

Figure 5. Poorly designed "finished alert"

The main problems with the alert in Figure 3 are as follows:

- Its title isn't descriptive (and is overly alarming).
- The implied subject of the message is the user instead of the application.
- The word "caution" is in all uppercase letters, and the punctuation includes an exclamation point.

Also, the buttons are slightly shorter than the standard height. Since the audience in this case is programmers, the words *kernel* and *runtime* are acceptable, though the use of *runtime* in this context is colloquial and can be more clearly stated with a simpler word. To improve this alert, you could change the title to "Download" and the message to "The code you are downloading redefines one or more kernel definitions. Continuing the download may make the application unusable." Also, the buttons should be made 20 pixels high.

The alert in Figure 4 isn't movable, so it has no title and can't be repositioned. The message, with its "If . . ." clause, isn't direct and clear enough. Also, it's not clear which button provides a safe escape mechanism. Finally, the "Work offline" button title has incorrect capitalization. To improve the alert, you could make it movable and give it the title "Connect to Server." The message should be "The public calendar server selected in the Chooser is different from the one you used last. Connecting to the new server will cause all public event information in your document to be lost." You could add a Quit button as an escape mechanism, giving the alert three buttons — Quit, Connect, and Work Offline (the default).

The alert in Figure 5 doesn't contain any title, icon, or buttons. Because there are no buttons, it's not clear how to get rid of the message without reading to the end of it. Also, its message should be stated in the present (for example, *is named*). But the biggest problem is that this is a nuisance alert: the success of the capture could have been confirmed in an earlier step, when the user was asked to pick the filename. The solution is to get rid of the alert altogether.

THE PAYOFF

Spending some time thinking about the design of your application's alerts makes sense because it results in a better product. If you follow the simple guidelines presented in this article, your alerts should be in really good shape. Your users will have an easier time recovering from errors, adding to their positive experience with your software.

RECOMMENDED READING

- *Electronic Guide to Human Interface Design* (Addison-Wesley, 1994). This CD (available from APDA) combines the *Macintosh Human Interface Guidelines* and its companion CD, *Making It Macintosh*.
- *Macintosh Human Interface Guidelines*, (Addison-Wesley, 1993). Available separately from APDA in book form.
- *Inside Macintosh: Macintosh Toolbox Essentials* (Addison-Wesley, 1992), Chapter 6, "Dialog Manager."

Thanks to our technical reviewers Pete Bickford, Sharon Everson, Chris Forden, Elizabeth Moller, and Mark Stern.•

Macintosh Programmer's Toolbox Assistant

by Apple Computer, Inc.

Have you ever needed to find current information about using a crucial Macintosh Toolbox call while in the middle of writing your application? Now with just a click of your mouse, you can get instant access to more than 5,000 Toolbox calls that are at the heart of Macintosh system software.

Macintosh Programmer's Toolbox Assistant is an invaluable CD-ROM database that will help you find key data structures, resources, constants, and functions when you need them the most.

Available from a bookstore near you, or through APDA:
1-800-282-2732 in the U.S.   1-800-637-0029 in Canada   (716)871-6555 elsewhere

For more information or product updates, see our site on the World Wide Web
at http://www.info.apple.com/dev/mpta.html
Updates also available on the develop Bookmark CD.

TIM MAROONEY

MPW TIPS AND TRICKS

ToolServer Caveats and Carping

MPW comes with dozens of useful tools and scripts. They're handy for a lot of things besides programming — or would be, if you were willing to keep the MPW Shell open all the time, and if they weren't based on command lines. Fortunately, the Shell is not the only way to use them: a small application known as ToolServer makes it possible to run MPW commands in a standalone mode. You can write double-clickable MPW scripts, give MPW commands from AppleScript, and write front ends to tools in high-level programming languages.

Using ToolServer isn't exactly like using the MPW Shell. There are caveats if you want to write scripts and tools that will work in both environments. We'll first take a look at these issues and then explore how to package commands for use with ToolServer.

MODULARITY AND FACTORING

Shell scripting languages such as **sh** and **csh** in UNIX®, as well as MPW, have always taken a rather cavalier approach to code organization. Most configuration is achieved with a global namespace of environment variables. This is a problem with ToolServer, because you don't want to load your entire set of MPW startup scripts every time you run a command. Even if you wanted to, you couldn't — ToolServer doesn't have text editing, menu bar customization, or other user interface elements of the MPW Shell, so it's missing several built-in commands. Your existing startup scripts won't work, and some utility commands may also fail.

Three principles from structured software design are useful here:

- Separate user interface code from core code.
- Use the "include" mechanism to provide modularity.
- Reduce dependencies between modules.

Let's take a concrete example. Many of us cut our teeth as programmers on UNIX. Initiates of this brilliant but byzantine operating system tend to grow fond of its command set, in much the same way that cabalists become attached to bizarre metaphysical formulas purporting to explain the universe. A UNIX wizard's MPW startup script usually contains a list of aliases to translate between UNIX and MPW: Alias ls Files, Alias cp Duplicate, and so on. These commands are then used in all the wizard's utility scripts as well. This creates a problem with ToolServer: it can't use these startup scripts because they also customize the user interface with commands like AddMenu and SetKey. Without the aliases, though, the utility scripts won't run.

One solution to this problem combines the first two principles listed above. First, separate the aliases from the user interface setup code, yielding two different startup files. Both files are invoked by the MPW Shell startup process but neither is invoked at ToolServer startup. Second, instead of assuming a particular global configuration, make each utility script explicitly include whatever setup files it may require. MPW's analog of the **#include** directive of C is Execute, which executes a file in the current namespace.

We can apply common C bracketing conventions to avoid multiple inclusion of the same file. Assuming that our UNIX wizard has split off his or her aliases into a file named UNIXAliases, a script using these aliases would start — after the header comment — as follows:

```
if {__UNIXALIASES__} == ""
   execute UNIXAliases
end
```

The script file UNIXAliases would set the variable __UNIXALIASES__ to something other than the empty string, and decline to execute itself again if it had already been executed, like so:

```
if {__UNIXALIASES__} == ""
   set __UNIXALIASES__ "true"
   ... # the aliases go here
end # __UNIXALIASES__
```

TIM MAROONEY was discovered on the Isle of Wight by seal farmers in the Year of Our Lord 1394, and again seventy years later by Tasmanian basket twirlers out for a stroll in the Yukon. The little tyke pursued a happy life of fun, freedom, and quantum mechanics. He resurfaced in 1961, in the town of Holyoke, Massachusetts. Tim played a magician in bondage in a class play in the second grade, which may have prepared him for the contract work he's now doing at Apple.•

A different solution to the same problem involves the third principle, reducing dependencies between modules. Utility scripts don't really *need* to use **csh** commands, after all: the aliases are there mostly so that the wizard can type them into the MPW Shell, his or her fingers having long ago locked into an inflexible pattern of TTY interaction. If scripts don't assume the availability of a different command set — that is, if they stick with the MPW command names — the aliases need not be included at all.

Independence is a good idea for another reason: you may give your ToolServer scripts to other people at some point in your long and happy life. The more your commands depend on the global environment, including ToolServer startup files, the more likely they are to conflict with another user's environment.

INPUT AND OUTPUT

ToolServer implements most of the MPW Shell's I/O system, which is based on the **stdio** library and UNIX-style redirection. However, it doesn't read keyboard input or display text output. All of its I/O channels are ultimately files, pipes, or the pseudodevice Dev:Null.

The only mechanisms for interacting with the user in ToolServer are commands like Alert and Confirm that display dialog boxes, and interface tools you write yourself. Even these must be used with caution, since ToolServer can run remotely over a network, and hanging a server machine by bringing up a dialog box is often regarded as undesirable.

It helps to separate user interface code from core code, as already discussed. Commands you intend to run with ToolServer should not have a user interface: they should perform an action that's completely specified by their command line. An outermost user interface script can present choices to the user, then invoke an innermost command that has no user interface. The outermost script is just for ToolServer; the inner script or tool is suitable for both ToolServer and the MPW Shell.

You can detect when a command is running under ToolServer and squelch its user interface by looking at the environment variable BackgroundShell. This is the empty string when running under the MPW Shell, but it's nonempty under ToolServer. Most user interactions in MPW commands are just confirmation alerts, so if execution reaches a Confirm command and BackgroundShell is set, assume that the user would answer "no." All commands that require confirmation should support the **-y** and **-n** options, which provide answers on the command line, and these options should be provided when the commands are used from ToolServer.

Some MPW commands, such as Make and Backup, write output to the Worksheet, and the user then selects and executes the output. This model doesn't apply to the ToolServer environment since it has no Worksheet. The easiest solution is to redirect the command output to a temporary file, execute that file, and then delete it. This is less selective than using the Worksheet, which allows the user to decide which lines to execute. If selectivity is important, you can write a command that presents the lines of output to the user and allows them to be independently accepted or rejected.

We don't live in the best of all possible worlds, St. Thomas Aquinas and Dr. Pangloss to the contrary, and so commands often return errors. These generate text that's directed to the standard error channel. In the MPW Shell, error text goes to the frontmost window by default, but in ToolServer, the default is a file named *command*.err in the folder containing the command file. This is very antisocial behavior, especially since commands invoke other commands and the error file could wind up buried at some arbitrary-seeming place in your folder tree. Redirect the standard error channel to save yourself from Sisyphean levels of frustration whenever something goes just a little bit wrong.

There are two ways to redirect errors. First, you can use the standard MPW error redirection characters ≥, ≥≥, ∑, and ∑∑ in your outermost user interface script. For instance, the script line

```
Veeblefetzer ≥ "{Boot}"Veeblefetzer.Errors
```

would redirect errors to the file Veeblefetzer.Errors at the top level of your startup disk. This does little or nothing to bring the errors to your attention, though, so your outermost script should look something like this:

```
Set ErrorFile "{TempFolder}"MyUtility.Errors
Delete -i "{ErrorFile}"
Set Exit 0 # Don't bomb quietly on errors
Potrzebie ≥≥ "{ErrorFile}"
if {Status} == 0
   Veeblefetzer ≥≥ "{ErrorFile}"
end
if `Exists "{ErrorFile}"`
   Alert `Catenate "{ErrorFile}"`
   Delete -i "{ErrorFile}"
end
```

The other way to redirect errors is to set the ToolServer built-in variable BackgroundErr to the name of a file. This will create that file whenever there's an error. This is somewhat less flexible than redirection, but it can be set once and for all in a ToolServer startup script. That would make the script above read like this:

```
Delete -i "{BackgroundErr}"
Set Exit 0 # Don't bomb quietly on errors
Potrzebie
if {Status} == 0
   Veeblefetzer
end
if `Exists "{BackgroundErr}"`
   Alert `Catenate "{BackgroundErr}"`
   Delete -i "{BackgroundErr}"
end
```

Standard output can be controlled similarly, using either redirection characters or the environment variable BackgroundOut.

FORMS OF TOOLS

There are several ways to package commands for use with ToolServer. The most basic and boring ways are:

- Use the Execute Script command in ToolServer's File menu to select and execute a script file.
- Drop a script file on the ToolServer application icon.
- Give the "ToolServer [*script* …]" command in the MPW Shell.

There are more interesting deployment modes, but they require a bit more explanation.

Standalone scripts. If you change the creator of a script file to 'MPSX', double-clicking it in the Finder will launch ToolServer and send it an Open Document event, causing it to be executed. Use this approach for your outermost user interface scripts. To change the creator, use MPW's "SetFile -c 'MPSX' *file*" command.

There is, alas, no such thing as a standalone tool, but you can write a one-line script that invokes a tool with any parameters or none.

AppleScript. ToolServer is fully scriptable. Aside from the four required commands, it has only one scripting command, the dreaded DoScript. This takes a command written in another scripting language — MPW command language in this case — and passes the command to its script interpreter. DoScript is discouraged in new applications because it's unstructured, but it's very useful for pre-AppleScript applications that have their own languages.

A simple AppleScript script confers few benefits over a standalone ToolServer script. In fact, it's better to avoid mixing scripting languages if you can. However, using FaceSpan or another AppleScript authoring tool, you can use AppleScript to set up a conventional application that relies on ToolServer as a workhorse. Simply pass DoScript commands in response to user actions, redirecting errors and output to temporary files that you interpret in your AppleScript code.

Apple events. Finally, you can take AppleScript one step further, driving ToolServer directly with Apple events generated from compiled software. This delivers the maximum in flexibility and performance. You could even write a project-file development system based on MPW compilers. Another possibility would be HyperCard XCMDs, allowing MPW commands to be invoked from HyperTalk. An Apple event front end could be created for a particular MPW tool, allowing it to be cleanly invoked from other scripts or compiled software; this might also provide a simple user interface for controlling it with dialogs and menus.

ToolServer accepts the required Apple events, as well as DoScript and some special-purpose events related to status checking, command canceling, and error and output redirection. These are documented in Chapter 4 of the *ToolServer Reference* manual that comes with MPW. In this column in the last issue of *develop*, I provided sample code for interacting with SourceServer (another Apple event–driven MPW Shell subset), and that code can easily be adapted for ToolServer.

TOOLS FOR THE FUTURE?

Because it's tied to a command-line interface, the MPW toolset has come to seem rather archaic, but there's life in the old girl yet. ToolServer's support for Apple events and AppleScript allows innumerable improvements in its interface. In the future, we may see friendly front ends for various MPW tools, as well as deeper support for compilation and other kinds of file processing with MPW tools in third-party development systems.

Ultimately, MPW's command-line interface is destined to become a fading memory. Although it confers some advantages in power, it must give way to friendlier approaches in the end. However, if we fail to move its toolset forward into the post-command-line world, we will be poorer for the loss.

Thanks to Deeje Cooley, Arno Gourdol, and Rick Mann for reviewing this column.•

Printing Images Faster With Data Compression

Using JPEG image compression techniques can dramatically improve performance during printing to PostScript™ Level 2 printers; compressed images are significantly smaller and take much less time to print. You don't need to write PostScript code or special-case your code for PostScript printing; QuickTime and the printer driver do most of the work for you. You don't have to wait to get started, either. If you implement JPEG image data compression techniques in your application, users printing to PostScript Level 2 printers with the current LaserWriter 8.3 driver will see improvements in printing performance right away.

DAVID GELPHMAN

Many applications compress image data for storage and transmission, but compressing images for printing is relatively uncommon. With the techniques presented in this article, you can start printing with image data compression and realize significant performance gains without a lot of effort. First we'll explore the concepts behind using image data compression for printing, and then go through three sample applications that show you how to do it.

The first two samples demonstrate how to print existing compressed image data. Applications that already deal with JPEG compressed data, such as Web browsers and JPEG viewing applications, can benefit immediately from these techniques. Developers whose applications handle other kinds of compressed data (such as fax) can see how they might benefit in the future as printing software is enhanced to handle other types of compressed data.

Some applications don't already have compressed data to print. Painting applications, for example, handle image data that may not be in a standard compressed format. The third sample application shows you how to compress your data as you do your print-time imaging.

To give you an idea of the performance gains you might expect with these techniques, I printed the same images with and without JPEG image data compression and compared print times and data sizes. The improvements are notable — compressed

DAVID GELPHMAN (gelphman@rbi.com) seems to specialize in backwards-reading programming languages. From FORTH he moved into PostScript at Adobe Systems and then to Telescript at General Magic. He does do most other things in a more or less forward direction, although he has been known to fall off a horse backwards. David, together with his colleague Richard Blanchard, co-designed Apple's LaserWriter 8 PostScript printer driver while working at Adobe Systems. After a stint at General Magic, David now works at RBI Software Systems (http://www.rbi.com) as a contractor to Apple and Adobe on their PostScript printer drivers. He does other contracting work as well, primarily in the area of PostScript printing.•

color images, for example, can print in less than half the time. You may find the results so compelling that you'll want to implement these techniques in your own application.

This issue's CD contains the sample applications as well as some images you can use with them. It also contains a prerelease version of LaserWriter 8.3.1, which you may find useful for testing your application as you implement printing with compression.

THE BASICS

Realistic images can be quite large, resulting in slow print times. Compression algorithms such as JPEG, fax, and LZW are used to reduce the size of these images for storage and transmission. Image data compressed in these formats can be decompressed on PostScript Level 2 printers.

While many applications can handle compressed image data, at print time they usually decompress the data and use CopyBits to draw the decompressed images. Only a few applications use custom PostScript code to take advantage of the image decompression available in PostScript Level 2 output devices.

QuickTime's Image Compression Manager provides an API for applications to compress and decompress still image data. By using the Image Compression Manager functions, applications can draw JPEG compressed image data. If this drawing takes place at print time, the application is effectively passing compressed image data to the printer driver; this allows the driver to handle the compressed data appropriately for the target output device, as described in the next section. The application doesn't need to know whether that device is a QuickDraw, PostScript Level 1, or PostScript Level 2 device.

If your application handles only QuickDraw pictures, it doesn't need to perform any special action to take advantage of image data compression. QuickDraw pictures containing JPEG compressed image data are available from various sources; QuickTime can compress QuickDraw pictures transparently, and applications such as Adobe™ Photoshop can create QuickDraw pictures containing JPEG compressed image data. Applications that use DrawPicture to draw such pictures automatically take advantage of printer drivers that have special handling of compressed image data. All they need to do is let the QuickDraw low-level drawing routines do their normal thing.

LaserWriter drivers starting with version 8.3 are savvy about JPEG compressed images that are drawn with QuickTime. When the driver receives data that's compressed with JPEG compression and the PostScript output is destined for a PostScript Level 2 device, the driver sends the compressed data directly to the printer. Since JPEG compressed images can be as much as 1/10 to 1/40 the size of uncompressed images, the amount of data sent to the printer is much smaller, which drastically reduces print times.

HOW THE PRINTER DRIVER HANDLES COMPRESSED IMAGE DATA

In general, printer drivers intercept QuickDraw drawing through the QuickDraw low-level bottleneck routines. When an application draws compressed image data with the Image Compression Manager functions (or draws a compressed QuickDraw picture with DrawPicture), QuickTime passes the compressed data to the low-level QuickDraw drawing routines through the StdPix bottleneck routine. Normally, StdPix decompresses the data and passes the decompressed data to the bitsProc bottleneck routine for drawing.

StdPix is described in detail in *Inside Macintosh: QuickTime*, pages 3-137 to 3-139.•

The LaserWriter 8.3 driver installs custom bottleneck routines as replacements for the standard bottlenecks, including bitsProc and StdPix. The custom StdPix bottleneck is key to the special handling of compressed image data, as shown in Figure 1. The driver installs the custom StdPix bottleneck in the printing graphics port so that it can intercept calls to StdPix and examine the compressed data. If the data is compressed with a compression type that the driver recognizes and knows the printer is capable of receiving, the driver sends the data directly to the printer. Otherwise, it calls the standard StdPix, which, as described above, sends the decompressed data to the bitsProc bottleneck. Drivers that don't have a custom StdPix bottleneck (such as QuickDraw printer drivers and LaserWriter drivers previous to version 8.3) will always have decompressed data passed to their bitsProc bottleneck.

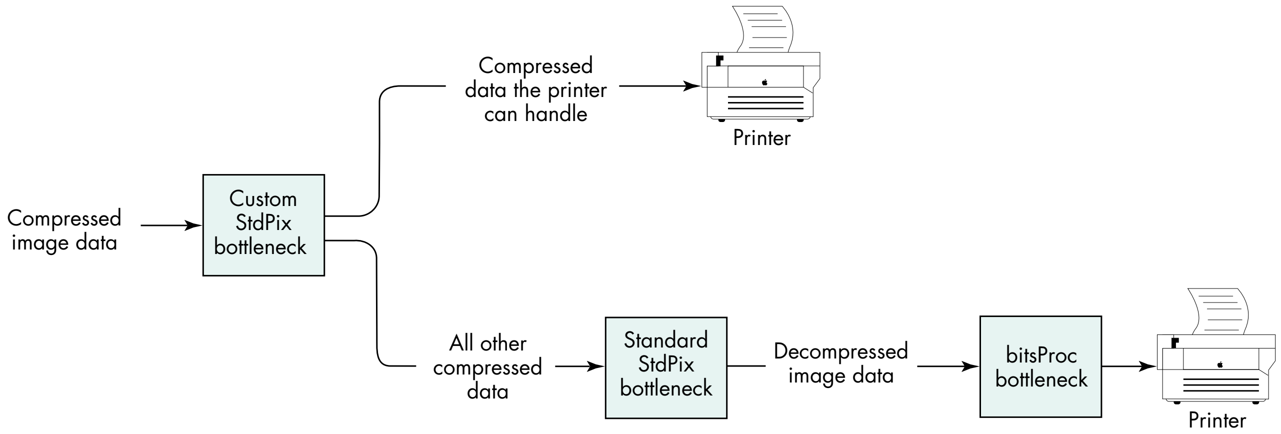


Figure 1. Special handling of compressed image data in the LaserWriter 8.3 driver

Using a custom StdPix bottleneck lets a printer driver handle different compression types appropriately. It also allows for the generation of correct output both for PostScript Level 2 output devices, all of which support JPEG, fax, and LZW decompression, and for PostScript Level 1 devices, which don't support any decompression. For drivers like LaserWriter 8.3 that spool (for background printing or as part of foreground printing), there's another advantage: since the spool file can contain compressed images instead of uncompressed images, users benefit from smaller disk space requirements.

The techniques described here for handling compressed image data will work correctly with any printer driver, not just PostScript drivers with this special compressed image data handling. Of course, the performance benefits will be seen only with drivers that do have it. Most QuickDraw printer drivers will not gain a performance benefit because they ultimately render decompressed data on the host system and send the rendered results to the printer. In fact, if the data is being compressed on the host specifically for printing, there will a performance penalty. A few QuickDraw drivers, such as Adobe's Acrobat™ PDFWriter, create data files that could potentially take advantage of image compression done by your application.

Note that this technique of using a custom StdPix bottleneck applies to printing to a color graphics port on a Macintosh system that has Color QuickDraw built in (most do). Black-and-white ports don't have StdPix bottlenecks; later we'll look at what to do if you're printing compressed data to a black-and-white port.

WHY THE DRIVER DOESN'T DO COMPRESSION FOR YOU

You might be wondering: "If using image data compression for printing is so great, why doesn't the driver do it for me automatically?" It's a good question and one that deserves a good answer.

Different kinds of images, such as fax images, photographic images, and synthetic images, have different characteristics. The best type of compression to apply depends on the type of image. Printer drivers operate at too low a level to make good decisions about image data compression. On the other hand, applications typically have a good idea about the kind of data they handle.

Additionally, some compression algorithms, such as JPEG, can be "lossy" (that is, they throw away information), and it would be inappropriate for the driver to apply them without user control. The driver user interface isn't well suited to specifying compression preferences, particularly since such decisions should be on a document by document basis or even on a per image basis within a document. The LaserWriter 8.x drivers do use PackBits compression for all image data passed to their low-level bitsProc bottleneck, but that's the only active compression done by the drivers and it isn't very effective for many types of image data.

PRINTING EXISTING COMPRESSED IMAGE DATA THAT FITS IN MEMORY

As mentioned earlier, applications that use DrawPicture to draw QuickDraw pictures containing JPEG data don't need to do anything special to print the images. In this section we'll look at how applications can print compressed image data that is not in a QuickDraw picture.

The JPEG Print sample application reads an existing compressed JPEG data file for display and printing. In this application, the JPEG data must fit completely in memory before it can be imaged. This is not a requirement for using compressed data, but is the simplest approach to describe initially. Later we'll talk about the case where the data doesn't all fit in memory at once.

At application startup, the JPEG Print sample code checks that QuickTime is installed. The code also tests to make sure there's a compression-decompression codec that can handle the decompression of JPEG data; the codec is used to decompress the data on the host if the data can't be sent to the printer in a compressed form. Applications that can already print compressed data without QuickTime and an appropriate codec should continue using their existing code to print when QuickTime and the codec aren't present.

FILLING IN THE IMAGEDESCRIPTION DATA STRUCTURE

The QuickTime image decompression functions require a handle to an ImageDescription data structure. This structure contains information about an image, such as the compression type used, the number of bytes in the compressed image, and the image height, width, and depth. QuickTime needs this data separate from the compressed data itself.

In the case of JPEG compressed data, much of the information required in the ImageDescription data structure is contained in the compressed JPEG data stream. The JPEG Print application reads the JPEG data stream and extracts the width, height, horizontal resolution, vertical resolution, and depth of the image. It then uses this data to build up an ImageDescription structure for use with the Image Compression Manager functions. The specifics of parsing a JPEG data stream for image description information aren't discussed here; this part of the sample code comes almost directly from the sample JFIF Translator application in the Macintosh OS Software Developer's Kit, with little modification.

CHOOSING THE APPROPRIATE DECOMPRESSION ROUTINE

To draw compressed still images with QuickTime, you can use one of three functions: DecompressImage, FDecompressImage, or the StdPix bottleneck routine. However, the DecompressImage and FDecompressImage functions always call the standard StdPix bottleneck; they do not call any custom StdPix bottleneck (including LaserWriter 8's) in the graphics port. Since we want our compressed image data to pass through the driver's StdPix bottleneck, we'll just call the StdPix bottleneck directly, as described in the next section.

For drawing to a black-and-white port, you'll need to use DecompressImage or FDecompressImage since a black-and-white port doesn't have a StdPix bottleneck. One of the arguments to DecompressImage and FDecompressImage (as specified in the QuickTime documentation) is a handle to the pixel map in which the decompressed image is to be displayed. In a black-and-white graphics port there is no PixMapHandle available; instead, there is a BitMap data structure. DecompressImage and FDecompressImage *can* accept a BitMap instead of a PixMapHandle as the destination to draw to, and that's what we pass to DecompressImage when drawing to a black-and-white graphics port.

CALLING THE STDPIX BOTTLENECK DIRECTLY

The StdPix bottleneck is declared as follows:

```
pascal void StdPix(PixMapPtr src, Rect *srcRect, MatrixRecordPtr matrix,
   short mode, RgnHandle mask, PixMapPtr matte, Rect *matteRect, short flags);
```

The first argument is a pointer to a PixMap "containing" the compressed image data. This isn't a PixMap in the normal QuickDraw sense; instead, it's a PixMap data structure that has compressed data "attached" to it with the QuickTime call SetCompressedPixMapInfo. This call associates an ImageDescription data structure and the corresponding compressed image data with a PixMap data structure. It's important that the compressed data not move in memory after you've associated it with the PixMap. If you use a handle to your compressed data, as we do in the sample code, you should lock the handle before your call to SetCompressedPixMapInfo and keep it locked until after you're done with the PixMap.

The next two arguments to StdPix specify a source rectangle and a transformation matrix that describes the mapping between the source rectangle of the image data and the destination rectangle. By specifying a source rectangle and a matrix rather than a source and a destination rectangle, the StdPix interface allows for more general coordinate transformations than just scaling and translation. Currently, however, QuickTime supports only scaling and translation.

The mode argument specifies which QuickDraw transfer mode to use when drawing the image. JPEG Print uses the ditherCopy mode. When printing to PostScript printers, ditherCopy mode is treated by the LaserWriter 8.x driver exactly like srcCopy mode, and the PostScript interpreter does any halftoning or dithering appropriate for the PostScript output device. When imaging to QuickDraw output devices, ditherCopy causes QuickDraw to dither the image, which usually yields better results than using srcCopy.

StdPix also accepts mask and matte arguments to obtain special effects. The mask argument has the same effect as clipping to a mask region as part of the imaging call.

The matte arguments allow for effects similar to those of Color QuickDraw's CopyDeepMask. Current LaserWriter 8.x drivers do not support clipping to bitmap regions, or the CopyDeepMask-like effects available with the matte arguments. Consequently, the mask and matte arguments are ignored by LaserWriter 8.x drivers.

The final argument to StdPix is a flags parameter. The relevant flags are callOldBits and callStdBits; they work together to specify whether a call to StdPix results in a call to the bitsProc bottleneck with decompressed data. When the callOldBits and callStdBits flags are both set, StdPix will always call the bitsProc bottleneck with decompressed data. If callOldBits is set and callStdBits is not, StdPix will call the bitsProc bottleneck with the decompressed data only if the bitsProc bottleneck is not StdBits, but a custom bitsProc routine.

The JPEG Print sample code uses a flags value of (callOldBits | callStdBits) to specify the most conservative handling of compressed image data during printing. Printer drivers that know how to handle compressed image data, such as LaserWriter 8.3, will have a custom StdPix bottleneck to intercept the call and adjust the flags appropriately. Drivers that don't know how to handle compressed image data will always receive decompressed image data via their bitsProc bottleneck.

Once we're ready to call the StdPix bottleneck, we don't want to just call the function StdPix; instead, we must be careful to use any custom StdPix bottleneck that has been installed. To do this, the code must check the current graphics port for custom QuickDraw bottlenecks, as shown in Listing 1. If there aren't any, the code gets the standard bottlenecks; otherwise, it gets the pointer to the CQDProcs record stored in the graphics port. Once it has the appropriate bottlenecks, the code uses the procedure pointer stored in the newProc1 field of the CQDProcs record; this is the StdPix bottleneck.

Listing 1. Calling the QuickDraw StdPix bottleneck directly

```
// Look to see if there are custom QuickDraw bottlenecks in the
// current graphics port.
if ((((CGrafPtr)qd.thePort)->grafProcs) == NULL) {
   // Get the standard bottleneck procs.
   SetStdCProcs(&myStdProcs);
   // The newProc1 field is the StdPix bottleneck.
   MyProcPtr = (StdPixProcPtr)myStdProcs.newProc1;
} else {
   // Use the grafProcs record in the current port to obtain the custom
   // bottleneck procs. The newProc1 field is the StdPix bottleneck.
   MyProcPtr =
      (StdPixProcPtr) ((CGrafPtr)qd.thePort)->grafProcs->newProc1;
}
// Now call the bottleneck.
CallStdPixProc(MyProcPtr, SpecialPixMapP, &srcRect, &theMatrix,
   ditherCopy, NULL, NULL, NULL, flags);
```

USING DATA-LOADING TECHNIQUES TO PRINT LARGE COMPRESSED IMAGES

The compressed image data you're working with may not fit completely in memory. QuickTime supports this case through the use of a data-loading function, which you supply. QuickTime calls this function as needed to obtain data during image decompression. Data loading eliminates the need to have the full image in memory, greatly reducing memory usage.

The use of a data-loading function is described in somewhat sketchy terms in *Inside Macintosh: QuickTime*, pages 3-148 to 3-150. Basically, your application creates a buffer that your data-loading function uses for passing data to QuickTime. In preparation for the StdPix call, you call SetCompressedPixMapInfo with a pointer to the beginning of the buffer, the buffer length, and your data-loading function. When you call the StdPix bottleneck, QuickTime calls the data-loading function as necessary to obtain the compressed image data.

The data-loading function is declared as follows:

```
pascal OSErr MyDataLoadingProc(Ptr *dataP, long bytesNeeded, long refcon);
```

The first argument is a pointer to a pointer into your data buffer (the one you supplied in the call to SetCompressedPixMapInfo as described earlier). The bytesNeeded argument tells your function how many bytes need to be available in the data buffer pointed to by the pointer in *dataP *after* the function call returns. The refcon argument lets you pass additional information to your data-loading function.

EXTENDING JPEG PRINT WITH A DATA-LOADING FUNCTION

The sample application JPEG Print with Dataload, an extended version of JPEG Print, uses the function MyDataLoadingProc, shown in Listing 2. Code not included here fills up the buffer with the first chunk of compressed data and sets up the data-loading function so that the refcon passed to it is a pointer to our application-defined DataLoad structure.

The data-loading function's job is to ensure that when it's called with a request for bytesNeeded bytes of data, at least that many bytes are available in the buffer pointed to by *dataP *after* the data-loading function returns. When MyDataLoadingProc is called with dataP not NULL, the code first computes how many bytes remain in the buffer from *dataP to the end of the buffer. If that number of bytes is greater than or equal to bytesNeeded, there are enough bytes available and the function returns. Otherwise, the data from *dataP to the end of the buffer is copied to the beginning of the buffer, and the remainder of the buffer is filled up with new data. Once the buffer is refilled, *dataP is set to point to the beginning of the buffer so that the caller starts getting its data there.

TECHNIQUES FOR COMPRESSING AND PRINTING UNCOMPRESSED DATA

Your application may not have compressed data to print. The third sample application on this issue's CD, PrintPICTtoJPEG, compresses 32-bit-deep image data and prints it. To obtain a source of bits to compress, PrintPICTtoJPEG takes a PICT file and images it into a 32-bit-deep offscreen bitmap. It then draws from this bitmap into the current graphics port. During printing, the data in the offscreen bitmap is (optionally) compressed using JPEG compression, and then printed using the techniques for printing compressed data as discussed above for the JPEG Print application.

The PrintPICTtoJPEG application uses PICT data solely as a source of bits to use to demonstrate compression. *By no means* are we advocating this technique as the proper way to print QuickDraw pictures. QuickDraw pictures may contain line art, text, custom PostScript code, and images of varying depths that will image and print much better if you just use DrawPicture. A good portion of the PrintPICTtoJPEG

Listing 2. The data-loading function

```
static pascal OSErr MyDataLoadingProc(Ptr *dataP, long bytesNeeded,
      long refcon)
{
   OSErr theErr = noErr;

   if (dataP != NULL) {
      DataLoadPtr theDataLoadPtr = (DataLoadPtr) refcon;
      // refcon is a pointer to a structure that contains the locked
      // handle to our buffer, a field with the buffer size, and a field
      // with the file reference number of the image data file we are
      // decompressing.
      Ptr theDataBufferP =
         StripAddress(*(theDataLoadPtr->theDataBufferH));
      long theBufferSize = theDataLoadPtr->theBufferSize;
      short theRefNum = theDataLoadPtr->theRefNum;

      // Calculate the number of bytes left in our existing data buffer.
      long bytesAvail = theBufferSize - (*dataP - theDataBufferP);

      // Are there enough bytes in our buffer for this call? If so, we
      // don't need to read any more data.
      if (bytesNeeded > bytesAvail) {
         // We don't have enough bytes of data in our buffer. Figure
         // out how many bytes we should read to refill the buffer.
         long bytesToRead = theBufferSize - bytesAvail;

         // If there are bytes available at the end of our buffer, move
         // them to the beginning of the buffer.
         if (bytesAvail) BlockMove(*dataP, theDataBufferP, bytesAvail);

         // Go ahead and fill up the rest of the buffer, starting just
         // after the last valid byte in the buffer.
         theErr = FSRead(theRefNum, &bytesToRead, theDataBufferP +
                     bytesAvail);
         // Ignore end of file errors.
         if (theErr == eofErr) theErr = noErr;

         // Reset the data pointer used by the caller of the data-
         // loading function so that it points to the first byte of
         // valid data, which is now at the beginning of our buffer.
         *dataP = theDataBufferP;
      }
   } else {
      // The data mark reset case. This implementation doesn't know how
      // to reset the stream, so we return an error. We haven't seen
      // a data mark reset as part of JPEG decoding. (Note that not
      // handling this case slows down PhotoCD significantly.)
      theErr = -1;
   }
   return theErr;
}
```

application is devoted to getting a QuickDraw picture and drawing it into the offscreen bitmap as a source of bits. The interesting part of the application is the compression and imaging of the bits once we have them, and that's what we'll discuss here.

The PrintPICTtoJPEG application compresses data only as part of printing it. Of course, it isn't necessarily true that you would compress data only during printing; it's very likely that you would maintain the data in a compressed form. Only you know for sure how you want to handle it.

PrintPICTtoJPEG also does image compression on the data only if the printing port is a color graphics port; otherwise, it just does the usual CopyBits. (If you already have compressed image data, you can use FDecompressImage as in the JPEG Print application to draw already compressed images to a black-and-white graphics port. If you're compressing strictly for printing, there's no obvious benefit to do so for a black-and-white port.)

USING COMPRESSIMAGE

The simplest way to compress image data is to use the QuickTime functions CompressImage and FCompressImage. You call GetMaxCompressionSize to determine the maximum compression size of your image, and then allocate a handle of that size and pass it to CompressImage or FCompressImage, as shown in Listing 3.

GetMaxCompressionSize is likely to return a large size for full color images, perhaps a larger amount of memory than the application can allocate out of its application heap. To allow for this, PrintPICTtoJPEG first tries to allocate a handle in its application heap by using NewHandle. If that fails, it attempts to allocate temporary memory using the TempNewHandle function. In this way, the application can compress images when temporary memory is available without requiring a large application heap. If there isn't enough memory available, you can use the FCompressImage function with an application-supplied data-*unloading* function to write the data to disk as it's being compressed by QuickTime.

The sample code directly chooses JPEG image compression with any codec that supports JPEG compression with a quality value of codecNormalQuality. The other available constants for compression quality values are codecLosslessQuality, codecMaxQuality, codecMinQuality, codecLowQuality, and codecHighQuality. These constants give varying compression ratios and corresponding image fidelity.

PROVIDING A USER INTERFACE FOR COMPRESSION PREFERENCES

Although PrintPICTtoJPEG doesn't do this, your application should provide the user a way to specify compression parameters when using JPEG compression. This is especially important when you're applying a lossy compression method such as JPEG, since there's a tradeoff between compression size and image fidelity. Such a decision is appropriate on a per document or even a per image basis.

The PrintPICTtoJPEG application knows that the data it's working with is best suited for JPEG compression. If your application has a good idea of what kind of image data it's working with, it can make the choice of which compression scheme to apply to the data. If not, you should probably use the standard image-compression dialog to let the user choose both the compression scheme and the compression parameters.

PERFORMANCE MEASUREMENTS

As part of developing the sample applications, I did some stopwatch time measurements to see what kind of performance improvements we'd get with JPEG image data

Listing 3. Compressing image data with CompressImage

```
CodecType theCodecType = 'jpeg';
CodecComponent theCodec = (CodecComponent) anyCodec;
CodecQ spatialQuality = codecNormalQuality;
short depth = 32;

// sPixMap is a handle to the pixel map to be compressed.
// bounds is a pointer to a rectangle specifying the portion of the
// image to compress.
if (theErr == noErr)
   theErr = GetMaxCompressionSize(sPixMap, bounds, depth,
      spatialQuality, theCodecType, theCodec, &maxCompressionSize);
if (theErr == noErr) {
   // This allocation should be no problem.
   theDescH =
     (ImageDescriptionHandle) NewHandle(sizeof(ImageDescriptionHandle));
   // This allocation is probably for a lot of memory.
   compressedDataH = NewHandle(maxCompressionSize);
   theErr = MemError();

   // See if we allocated the ImageDescriptionHandle but not the memory
   // to receive the compressed image.
   if ((theDescH != NULL) && (theErr != noErr)) {
      // See if we can get temporary memory instead. Since we're going
      // to use the temporary memory as a real handle, we require
      // System 7.0 or later.
      compressedDataH = TempNewHandle(maxCompressionSize, &theErr);
      // This probably can't happen, but just in case...
      if (compressedDataH == NULL && theErr == noErr)
         theErr = iMemFullErr;
   }
}
if ((theErr == noErr) && (compressedDataH != NULL)
      && (theDescH != NULL)) {
   MoveHHi(compressedDataH);
   HLock(compressedDataH);
   theErr = CompressImage(sPixMap, bounds, spatialQuality, theCodecType,
            theDescH, StripAddress(*compressedDataH));
   HUnlock(compressedDataH);
}
```

compression. (The image files I used are included on this issue's CD.) The results, while carefully obtained, are obviously not comprehensive, but they'll give you an idea of what you can expect. All measurements were taken using a Power Macintosh 6100/66 as the computing host on relatively unloaded LocalTalk and EtherTalk networks. Unless the application uses JPEG image compression, the LaserWriter 8.3 driver compresses the data using PackBits compression.

For comparison purposes, I used LaserWriter 8.3, which has the special support for JPEG images described in this article, and LaserWriter 8.2.2, which does not. In both cases, the application printing code was identical. LaserWriter 8.3 sends the compressed JPEG data directly to a PostScript Level 2 printer; with LaserWriter 8.2.2, the data is decompressed on the host Macintosh by QuickTime and passed to the driver's bitsProc bottleneck. Since the LaserWriter 8.2.2 driver is seeing uncompressed data, it compresses the data with PackBits compression before sending it to the printer.

To measure print times for already existing compressed data, I used the JPEG Print application to take an already compressed 186K JPEG image of a jaguar and print it to a PostScript Level 2 printer. Table 1 shows the results.

Table 1. Jaguar JPEG image print times for already compressed data

Printer	Network	Print Time, PackBits	Print Time, JPEG
LaserWriter 320	LocalTalk	289 seconds	125 seconds
LaserWriter 16/600	EtherTalk	121 seconds	42 seconds

Next I used the PrintPICTtoJPEG sample application to measure and compare printing times both with and without compression on the host (Table 2). I used the same jaguar image as before but saved as a PICT file, and a smaller PICT file I already had on hand. Doing image compression on the host is time intensive: it routinely took 2 to 4 seconds to compress the large jaguar image. Even so, overall performance is better because the data transfer times to the printer are so much smaller.

Table 2. PICT image print times when compressing data on the host

Image File	Printer	Network	Print Time, PackBits	Print Time, JPEG Normal Quality
Jaguar (as PICT)	LaserWriter 320 LaserWriter 16/600	LocalTalk EtherTalk	288 seconds 116 seconds	129 seconds 44 seconds
Portrait.pict	LaserWriter 320 LaserWriter 16/600	LocalTalk EtherTalk	54 seconds 22 seconds	37 seconds 18 seconds

Table 3 compares the data sizes for JPEG and PackBits compression.

Table 3. Image compression data sizes

Image File	Image Size, PackBits	Image Size, JPEG Normal Quality
Jaguar	2955K bytes	186K bytes
Portrait.pict	399K bytes	44K bytes

PRINTING WITH DATA COMPRESSION: CURRENT AND FUTURE DRIVERS

Today's LaserWriter 8.3 driver has direct support for handling JPEG compressed images as described in this article. LaserWriter 8.3 supports JPEG compression only when printing to Apple's PostScript Level 2 printers. When printing to other PostScript printers or to PostScript files on disk, the driver uses the JPEG decompressor on the host to decompress the data, regardless of user settings.

LaserWriter 8.3.1 and future LaserWriter 8.x drivers will take advantage of JPEG compression when printing to all PostScript Level 2 printers as well as when saving to disk with Level 2 Only selected in the standard file dialog. Adobe's PostScript printer driver for the Macintosh, PSPrinter, will soon take advantage of JPEG compression, as will a future version of the PostScript printing system for QuickDraw GX.

The prerelease version of LaserWriter 8.3.1 on this issue's CD will enable you to test your application with JPEG compression when printing to non-Apple printers or to disk. Remember that JPEG compressed data will be written into the data stream only when your application prints JPEG compressed data and the printer is a PostScript Level 2 printer. If you're saving PostScript files to disk, be sure to choose the Level 2 Only setting in the standard file dialog. Choosing the Level 1 Compatible setting causes the driver to write uncompressed data into the output file. When you print 24-bit photo-realistic images using JPEG compression, files saved with the Level 1 Compatible setting will be about 10 to 40 times larger than files saved with the Level 2 Only setting.

Since PostScript Level 2 output devices also have fax and LZW decompression filters available, Apple is considering adding support for these compression formats to a future LaserWriter 8.x driver so that applications handling these types of data can take advantage of the techniques described here. If you would take advantage of fax or LZW support in the LaserWriter driver, let us know at AppleLink DEVFEEDBACK or devfeedback@applelink.apple.com on the Internet.

LET'S GET STARTED!

JPEG images are now abundant, especially on the Internet where more and more people encounter them each day. Let's start printing these as compressed images! By implementing the techniques presented here for printing JPEG compressed image data, you can give your users immediate and substantial gains in printing performance. Plus you'll be well on your way to printing other kinds of compressed data when printing software is enhanced to support it.

Thanks to Richard Blanchard, Paul Danbold, Peter Hoddie, Kent Sandvik, and Nick Thompson for reviewing this article.•

The New Device Drivers: Memory Matters

If you're writing a device driver for the new PCI-based Macintosh computers, you need to understand the relationship of the memory an application sees to the memory the hardware device sees. The support for these drivers (which will also run under Copland, the next generation of the Mac OS) includes the PrepareMemoryForIO function, as discussed in my article in Issue 22. This single coherent facility connects the application's logical view of memory to the hardware device's physical view. PrepareMemoryForIO has proven difficult to understand; this article should help clarify its use.

MARTIN MINOW

If you managed to struggle through my article "Creating PCI Device Drivers" in *develop* Issue 22, you probably noticed that it got rather vague toward the end when I tried to describe how the PrepareMemoryForIO function works. There are a few reasons for this: the article was getting pretty long and significantly overdue (the excuse), and I really didn't understand the function that well myself (the reason). Things are a bit better now, thanks to the enforced boredom of a very long trip, the need to teach this algorithm to a group of developers, and some related work I'm doing on the SCSI interface for Copland.

My previous article showed the simple process of preparing a permanent data area that might be used by a device driver to share microcode or other permanent information with a device. This article attacks a number of more complex problems that appear when a device performs *direct memory access* (DMA) transfers to or from a user data area. It also explores issues that arise if data transfers are needed in situations where the device's hardware cannot use DMA.

A later version of the sample device driver that accompanied the Issue 22 article is included in its entirety on this issue's CD. Of course, you'll need a hardware device to use the driver and updated headers and libraries to recompile it. Included is the source code for the DMA support library (files DMATransfer.c and DMATransfer.h), which consists of several functions I've written that interact with PrepareMemoryForIO; the revised sample device driver shows how this library can be incorporated into a complete device driver for PCI-based Power Macintosh computers.

I'll assume that you've read my earlier article (which you can find on the CD if you don't have it in print). That article gives an overview of the new device driver architecture and touches on the PrepareMemoryForIO function, but for a

MARTIN MINOW is writing the SCSI plug-in for Copland on a computer named "There must be a pony here" and competes with his boss to see who is more cynical about Apple management. During the few moments he can escape from meetings, he runs with the Hash House Harriers.•

comprehensive description of the architecture and details about the function, see *Designing PCI Cards and Drivers for Power Macintosh Computers* (available from APDA). I'll also assume that you're reasonably familiar with the basic concepts of a virtual memory operating system, including memory pages and logical and physical addresses; for a brief review, see "Virtual Memory on the Macintosh."

PREPARING MEMORY FOR A USER DATA TRANSFER

At the beginning of a user data transfer (a data transfer on behalf of a program that's calling into your driver), the device driver calls PrepareMemoryForIO to determine the physical addresses of the data and to ensure the coherency of memory caches. At the end of the transfer, the driver calls the CheckpointIO function to release system resources and adjust caches, if necessary. PrepareMemoryForIO performs three functions that are necessary for DMA transfers: it locates data in physical memory; it ensures that the data locations contain the actual data needed or provided by the device; and, with the help of CheckpointIO, it maintains cache coherence.

Your device driver can call PrepareMemoryForIO from task level, from a software interrupt, or from the mainline driver function (that is, DoDriverIO). CheckpointIO can be called from task level, from a software interrupt, or from a secondary interrupt handler. (For more on the available levels of execution, see "Execution Levels for Code on the PCI-Based Macintosh.") In a short while, we'll see how the fact that these functions must be called from particular points affects the transfer process.

If the data is currently in physical memory, PrepareMemoryForIO locks the memory page containing the data so that it cannot be relocated. If the data isn't in physical

VIRTUAL MEMORY ON THE MACINTOSH

BY DAVE SMITH

Virtual memory on the Macintosh has two major functions: it increases the apparent size of RAM transparently by moving data back and forth from a disk file, and it remaps addresses. Of the two, remapping addresses is more relevant to device driver developers (and, incidentally, much more of a headache).

When Macintosh virtual memory is turned on, the processor and the code running on the processor always access *logical addresses*. A logical address is used the same way as a physical address; however, the Memory Management Unit (MMU) integrated into the processor remaps the logical address on the fly to a physical address if the data is resident in memory. If the data isn't resident in memory, a *page fault* occurs; this requires reading the desired data into memory from the disk and possibly writing other, unneeded data from memory to the disk to free up space in memory. (This explanation is slightly simplified, of course.)

Since it would be impractical to have a mapping for each byte address, memory is subdivided into blocks called *pages*. A page is the smallest unit that can be remapped. Memory is broken into pages on *page boundaries*, which are page-size intervals starting at 0. The remapping allows physical pages that are not actually contiguous in physical memory to appear contiguous in the logical address space.

The Macintosh currently uses a page size of 4096 bytes; however, future hardware may use a different page size. You should call the GetLogicalPageSize function in the Driver Services Library to determine the page size if you need it.

DMA is performed on physical addresses since the MMU of the processor is not on the address bus that devices use. One of the functions of PrepareMemoryForIO is to translate logical addresses into physical addresses so that devices can copy data directly to and from the appropriate buffers.

Many virtual memory systems provide multiple logical address spaces to prevent applications from interfering with each other. It appears to each application that it has its own memory system, not shared with any other application. The Macintosh currently has only one logical address space, but future releases of the Mac OS will support multiple logical address spaces.

memory, PrepareMemoryForIO calls the virtual memory subsystem and a page fault occurs, reorganizing physical memory to make space in it for the data. After the transfer finishes, CheckpointIO releases the memory page locks.

PrepareMemoryForIO and CheckpointIO perform an important function related to the use of caches. A *cache* is a private, very fast memory area that the CPU can access at full speed. The processor runs much faster than its memory runs; to keep the processor running at its best speed, the CPU copies data from main memory to a cache. Both the PowerPC and the Motorola 68040 processors support caching, although their implementation details differ. The important point is that a value of a data item in memory can differ from the value for the same data item in the cache

EXECUTION LEVELS FOR CODE ON THE PCI-BASED MACINTOSH

BY TOM SAULPAUGH

Native code on PCI-based Macintosh computers may run in any of four execution contexts: software interrupt, secondary interrupt, primary interrupt, or task. All driver code contexts have access to a driver's global data. No special work (such as calling the SetA5 function on any of the 680x0 processors) is needed to access globals from any of these contexts.

SOFTWARE INTERRUPT

A *software interrupt routine* runs within the execution environment of a particular task. Running a software interrupt routine in a task is like forcing the task to call a specific subroutine asynchronously. When the software interrupt routine exits, the task resumes its activities. A software interrupt routine affects only the task in which it's run; the task can still be preempted so that other tasks can run. Those tasks, in turn, can run their own software interrupt routines, and a task running a software interrupt routine can be interrupted by a primary or secondary interrupt handler.

All software interrupt routines for a particular task are serialized; they don't interrupt each other, so there's no equivalent to the 680x0 model of nested primary interrupt handlers.

Page faults are allowed from software interrupt routines. A software interrupt routine is analogous to a Posix signal or a Windows NT asynchronous procedure call. A software interrupt routine running in the context of an application, INIT, or **cdev** doesn't have access to a driver's global data.

SECONDARY INTERRUPT

The *secondary interrupt level* is the execution context provided to a device driver's *secondary interrupt handler*. In this context, hardware interrupts are enabled and additional interrupts may occur. A secondary interrupt handler is a routine that runs in privileged mode with primary interrupts enabled but task switching disabled.

All secondary interrupt handlers are serialized, and they never interrupt primary interrupt handlers; in other words, they resemble primary interrupt handlers but have a lower priority. Thus, a secondary interrupt handler queued from a primary interrupt handler doesn't execute until the primary interrupt handler exits, while a secondary interrupt handler queued from a task executes immediately.

Page faults are not allowed at primary or secondary interrupt level. A secondary interrupt handler is analogous to a deferred task in Mac OS System 7 or a Windows NT deferred procedure call. Secondary interrupt handlers, like primary interrupt handlers, should be used only by device drivers. Never attempt to run application, INIT, or **cdev** code in this context or at primary interrupt level.

PRIMARY INTERRUPT

The *primary interrupt level* (also called *hardware interrupt level*) is the execution context in which a device's *primary interrupt handler* runs. Here, primary interrupts of the same or lower priority are disabled, the immediate needs of the device that caused the interrupt are serviced, and any actions that must be synchronized with the interrupt are performed. The primary interrupt handler is the routine that responds directly to a hardware interrupt. It usually satisfies the source of the interrupt and queues a secondary interrupt handler to perform the bulk of the servicing.

TASK (NON-INTERRUPT)

The *task level* (also called *non-interrupt level*) is the execution environment for applications and other programs that don't service interrupts. Page faults are allowed in this context.

(called *cache incoherence*). Furthermore, you have to explicitly tell the PowerPC or 680x0 processor to synchronize the cache with memory.

Normally, the processor hardware prevents cache incoherence from causing data value problems. However, for some processor architectures, DMA transfers access main memory independently of the processor cache. PrepareMemoryForIO (for write operations) and CheckpointIO (for read operations) synchronize the processor cache with main memory. This means that DMA write operations write the valid contents of memory, and the processor uses the valid data just read from the external device.

As noted earlier, some devices cannot perform DMA transfers; instead, they use *programmed I/O*, in which the CPU moves data between logical addresses and the device. PrepareMemoryForIO also returns the logical address that such devices must use.

A SIMPLE MEMORY PREPARATION EXAMPLE

Listing 1 presents a very simple example that shows how a memory area may be prepared for I/O.

> **To simplify listings,** I've often omitted data type casting. Think of all data types as unsigned 32-bit integers. Because of this omission, you can't implement these listings as written, but should base your code on the sample on this issue's CD.•

PrepareMemoryForIO is called with one parameter, an IOPreparationTable. Among other things, this table specifies one or more address ranges to prepare (only one, in this example). Each address range is indicated by a starting logical address and a count of the number of bytes in the range.

The IOPreparationTable also points to a *logical mapping table* and a *physical mapping table* (gLogicalMapping and gPhysicalMapping in our example). The physical mapping table is where PrepareMemoryForIO returns the page addresses that the driver can use to access the client's buffer during DMA. The logical mapping table is the list of addresses that the driver must use for doing programmed I/O.

The simplest IOPreparationTable options — kIOMinimalLogicalMapping and kIOLogicalRanges — are set in this example. The kIOMinimalLogicalMapping flag indicates that only the first and last logical pages need to be mapped, while the kIOLogicalRanges flag indicates that the data (here, the gMyBuffer vector) consists of logical addresses.

Because kIOMinimalLogicalMapping is set, the logical mapping table requires two entries for each address range; we have only one range, so our logical mapping table needs a total of two entries. The physical mapping table requires one entry per page; we set this to two entries because our 512-byte buffer may cross a page boundary. When writing your driver, you can use the GetMapEntryCount function in the DMA support library to compute the actual number of physical mapping table entries needed for an address range.

If the preparation is successful, the driver performs the DMA transfer and calls CheckpointIO to release internal operating system structures that were used by PrepareMemoryForIO. PrepareMemoryForIO sets the kIOStateDone flag in the IOPreparationTable's state field if the entire area has been prepared.

If PrepareMemoryForIO can't prepare the entire area, it doesn't set the kIOStateDone flag, and your driver needs to call PrepareMemoryForIO again with the firstPrepared

Listing 1. Simplified memory preparation

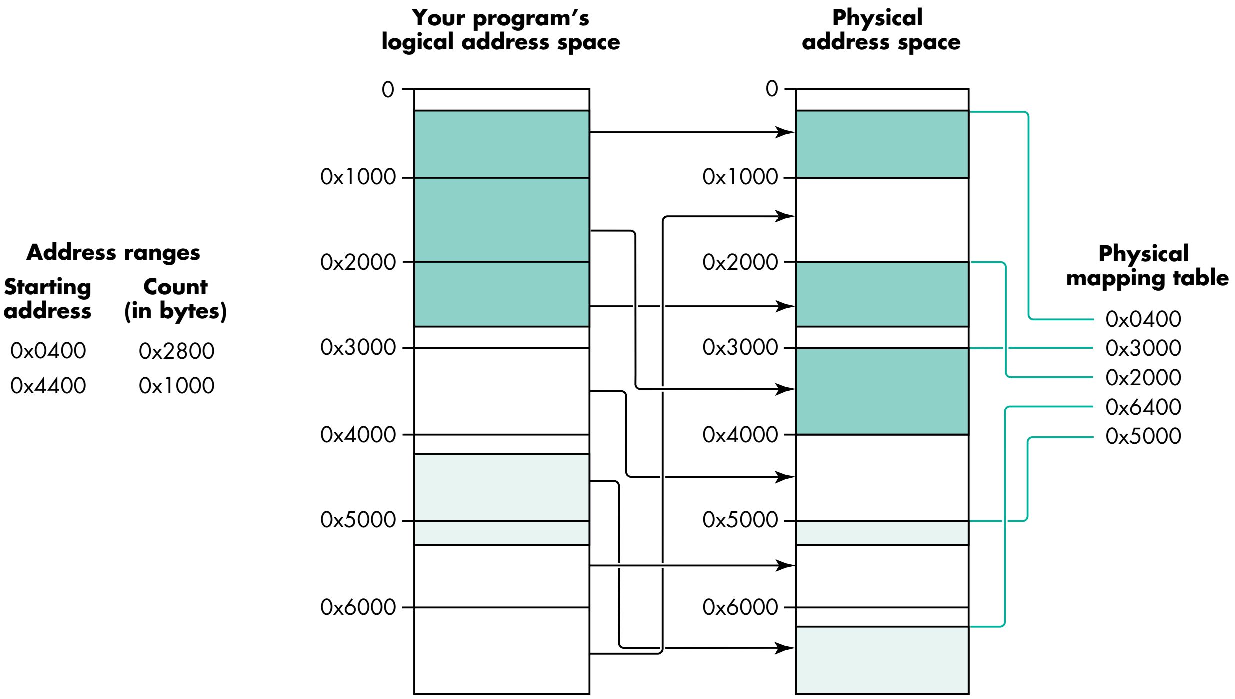
```
#define kBufferSize  512
#define kMapCount    2
/* The buffer your driver or application is preparing */
UInt8                gMyBuffer[kBufferSize];
IOPreparationTable   gIOTable;
/* Logical & physical mapping tables, filled in by PrepareMemoryForIO */
LogicalAddress       gLogicalMapping[2];
PhysicalAddress      gPhysicalMapping[kMapCount];

void SimpleMemoryPreparation(void)
{
   OSStatus    osStatus;

   gIOTable.options = 
      (kIOMinimalLogicalMapping | kIOLogicalRanges | kIOIsInput);
   gIOTable.state = 0;
   gIOTable.addressSpace = kCurrentAddressSpaceID;
   gIOTable.granularity = 0;
   gIOTable.firstPrepared = 0;
   gIOTable.lengthPrepared = 0;
   gIOTable.mappingEntryCount = kMapCount;
   gIOTable.logicalMapping = gLogicalMapping;
   gIOTable.physicalMapping = gPhysicalMapping;
   /* Set the logical address to be mapped and the length of the area
      to be mapped. */
   gIOTable.rangeInfo.range.base = (LogicalAddress) gMyBuffer;
   gIOTable.rangeInfo.range.length = sizeof gMyBuffer;
   /* Call PrepareMemoryForIO and process the preparation. */
   do {
      osStatus = PrepareMemoryForIO(&gIOTable);
      if (osStatus != noErr)
         break;
      MyDriverDMARoutine(...);
      CheckpointIO(gIOTable.preparationID, kNilOptions);
      gIOTable.firstPrepared += gIOTable.lengthPrepared;
   } while ((gIOTable.state & kIOStateDone) == 0);
}
```

field updated to reflect the number of bytes prepared in this range of memory. The recall must be done from a software interrupt routine; it cannot be performed from an interrupt handler.

MORE ABOUT MAPPING

Address ranges to be prepared by PrepareMemoryForIO may cross one or more page boundaries and thus may take up two or more pages in physical memory. Figure 1 shows what the physical mapping might look like for two address ranges: the first is more than two pages long and crosses two page boundaries, while the second is an even page long and crosses one page boundary.

Each address range maps to an area in physical memory that can be thought of as having up to three sections: the beginning page, the middle pages, and the ending page.



Address ranges

Starting address	Count (in bytes)
0x0400	0x2800
0x4400	0x1000

Figure 1. Mapping to multiple pages

- Every address range produces a beginning page. Your data may start at an offset into this page, depending on the starting address of the range. This is true for both address ranges in Figure 1. The address in the mapping table for the beginning page points to the beginning of your data in the page. Notice that for the second address range in our example, the logical address for the start of the data, 0x4400, maps to the physical address 0x6400.
- If your address range maps to three or more pages, some number of middle pages are completely filled with your data. The first address range in Figure 1 illustrates this.
- If your address range maps to two or more pages, the data on the ending page begins at the beginning of the page, but it may cover only part of the page, depending on the count in your address range.

Unfortunately, there's no simple one-to-one correspondence between entries in the physical and logical mapping tables and the address range (or ranges) that a driver or application specifies when it calls PrepareMemoryForIO. Because of this, the function that controls a driver's DMA or programmed I/O process must iterate through the input address ranges and output mapping tables to compute the address and size of each data transfer segment. As you'll see when you look at the DMA support library on this issue's CD, this turns out to be an extremely complex process.

The DMA support library functions iterate through the address ranges and mapping tables, matching the two together to provide each data transfer segment in order. The library recognizes when two physical pages are contiguous and extends the data transfer length as far as possible.

When called for the example in Figure 1, the DMA support library returns five physical transfer segments (this example doesn't demonstrate logical alignment problems). To learn how PrepareMemoryForIO's algorithm works, I'd recommend that you work out the actual addresses and segment transfer lengths using pencil and paper. (When you look at the DMA support library in DMATransfer.c, you'll see a more mechanized approach that I strongly recommend if you're developing complex software.)

THE DATA TRANSFER PROCESS

Figure 2 illustrates how a data transfer might proceed through the system. It shows the five steps involved in a transfer that requires partial preparation of a large chunk of data that can't be prepared in one gulp. The diagram also shows the proper execution levels for each step. As we'll see later, the process is considerably simpler without partial preparation.

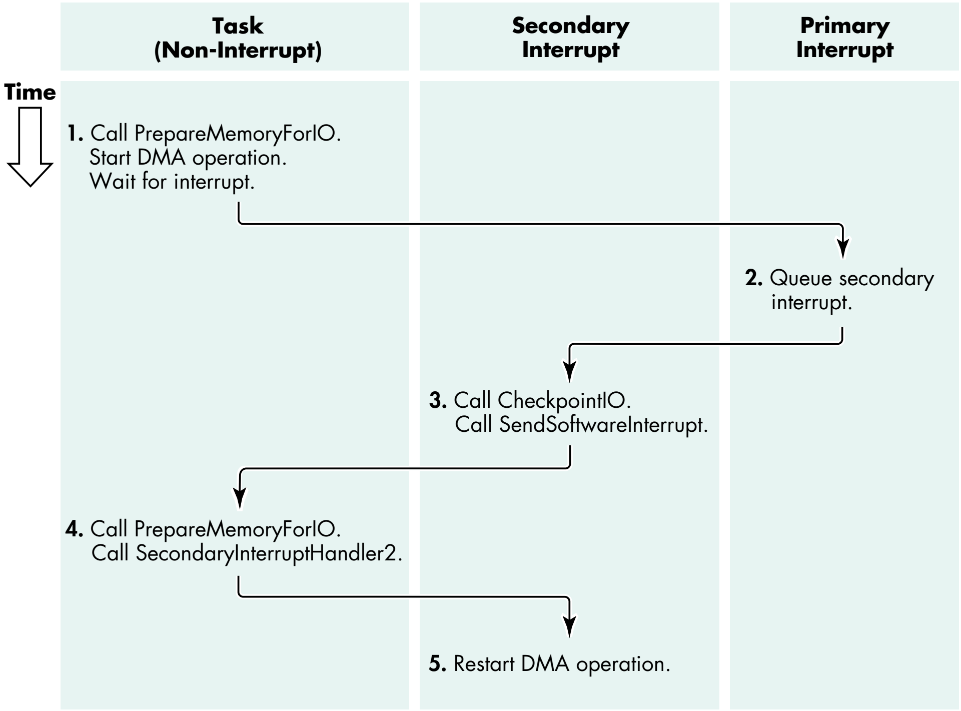

Task (Non-Interrupt)
Secondary Interrupt
Primary Interrupt
Time
1. Call PrepareMemoryForIO. Start DMA operation. Wait for interrupt.
2. Queue secondary interrupt.
3. Call CheckpointIO. Call SendSoftwareInterrupt.
4. Call PrepareMemoryForIO. Call SecondaryInterruptHandler2.
5. Restart DMA operation.

Figure 2. The progress of a data transfer with partial preparation

Here's a breakdown of the steps in the data transfer:

1. The transfer starts at task (application or driver mainline) level. The driver must call PrepareMemoryForIO from task level because PrepareMemoryForIO may require virtual memory page faults and has to reserve system memory for its own tables. After memory is prepared, the driver examines the logical and physical mapping tables and starts the DMA operation. It then waits for an interrupt. (Of course, the actual driver behavior depends on your hardware.)
2. When the driver's primary interrupt handler runs, it determines that another DMA transfer is needed, but that no more data is prepared (because the number of bytes transferred equals the value in the lengthPrepared field in the IOPreparationTable). Since another partial preparation must be performed, the primary interrupt handler queues a secondary interrupt and exits the primary interrupt. The device is in a "frozen" state: it either has data available (to read) or needs more data (to write) but cannot proceed at this time. I'll talk more about this problem later.

3. The driver's secondary interrupt handler starts. It examines its internal state and determines that a DMA transfer has been completed. It calls CheckpointIO with the kMoreIOTransfers flag to complete the current partial transfer. Since another data transfer will be needed, it begins the process of calling PrepareMemoryForIO again, by calling SendSoftwareInterrupt to queue a software interrupt routine. Then, with nothing more to do, the secondary interrupt handler exits. The device is still frozen.
4. The software interrupt routine runs. It updates the firstPrepared field and calls PrepareMemoryForIO to prepare the next segment (range of memory). This may require a page fault, causing the virtual memory subsystem to move data between main memory and the virtual memory disk file. When PrepareMemoryForIO finishes, the logical and physical mapping tables are updated and the lengthPrepared field contains the number of bytes that can be transferred in the next segment. The software interrupt routine calls a secondary interrupt handler (which is equivalent to queuing the handler).
5. The sequence returns to the secondary interrupt handler, and the DMA operation is restarted. The partial preparation algorithm continues at step 2, progressing through steps 2 to 5 until all data is transferred.

The device is frozen in steps 2 to 5; it cannot proceed on the current I/O request until the partial preparation completes. But note that the page fault handler in step 4 may require disk I/O; consequently, any device that can service the page fault device (such as the SCSI bus manager) cannot support partial preparation. Writers of disk drivers and other SCSI-based interface software must understand these restrictions.

A CLOSER LOOK: SOME EXAMPLES

Unfortunately, as a result of some necessary constraints of PrepareMemoryForIO, the code in Listing 1 isn't usable in an actual device driver when the data transfer results in the interruption of the hardware device by the CPU. In this section, I'll return to the five-step transfer process outlined above, with more detail on the way that a driver interacts with memory preparation. I'll illustrate the process with three different examples: the simple case of a single DMA transfer; the more complicated case where more than one DMA transfer is needed because the physical mapping entries are discontiguous; and finally the full five-step transfer process, complete with partial preparation.

A SIMPLE TRANSFER

Our first example uses the sample preparation shown in Figure 3. Here your application or driver created a simple IOPreparationTable for an application data buffer that's 512 bytes long and begins at logical address 0x01B89F80.

In this case the transfer process consists of only three steps:

1. The buffer in our example crosses a physical page boundary, so two mapping entries are needed. PrepareMemoryForIO fills in the logical and physical mapping tables and sets the lengthPrepared field. Since it has successfully prepared the entire buffer, it sets the kIOStateDone flag in the state field. After your driver uses the NextPageIsContiguous macro (in DMATransfer.h) to determine that the two physical mapping entries are contiguous, it puts the first physical address, 0x0077EF80, and the entire byte count into the DMA registers and starts the device.
2. When the transfer finishes, the driver's primary interrupt handler runs. It determines that the transfer has finished and queues a secondary interrupt to complete processing.

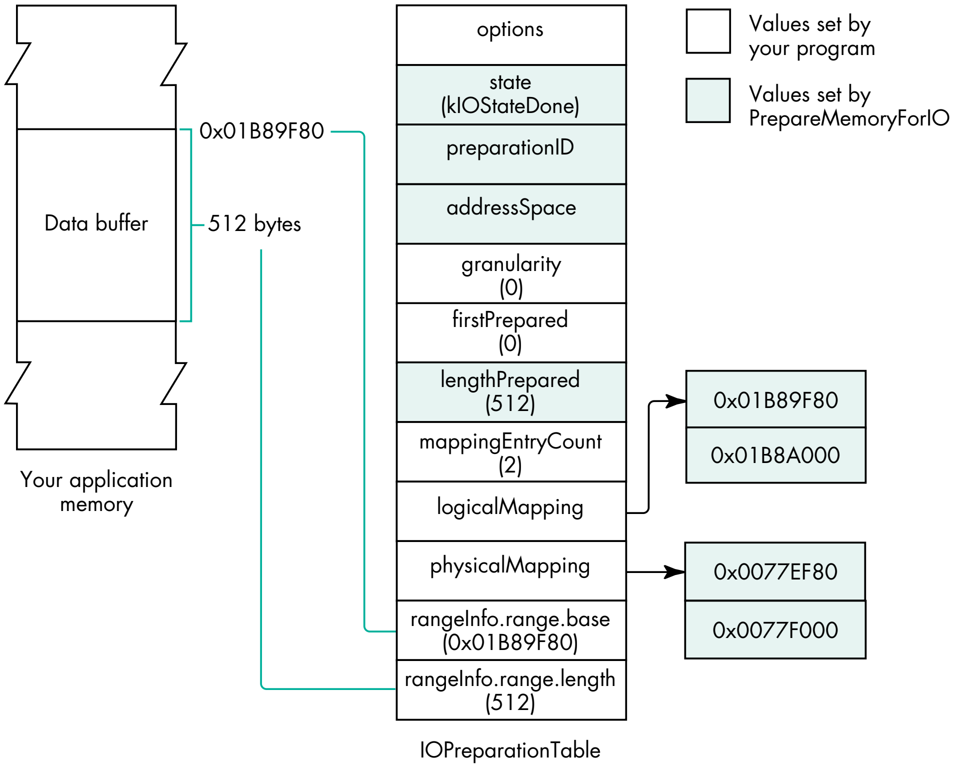


Figure 3. A simple IOPreparationTable

3. The driver's secondary interrupt handler calls CheckpointIO to complete the transfer. It then completes the entire device driver operation by calling IOCommandIsComplete.

DISCONTIGUOUS PHYSICAL MAPPING

The above example requires a single DMA transfer; however, if the physical mapping entries are discontiguous, the first two steps of the process become more complicated:

1. After preparation, your driver determines that the two physical mapping entries are not contiguous. Therefore, it puts the first physical address, 0x0077EF80, and the first byte count (128 bytes in this case) into the DMA registers and starts the DMA operation.
2. When the transfer finishes, the driver's primary interrupt handler runs. It determines that the transfer has finished; however, another physical transfer is needed and can be performed, so it loads the DMA registers with the new physical address and the remaining byte count (384 bytes in this case), restarts the DMA operation, and exits the primary interrupt handler.

   After this DMA operation finishes, the operating system reenters the primary interrupt handler. Upon the completion of the entire transfer, the primary interrupt handler queues the secondary interrupt handler to finish the entire operation.

PARTIAL PREPARATION

The example in Figure 3 requires only a single preparation, but in some cases PrepareMemoryForIO cannot prepare the entire area at once and so requires partial preparation. To illustrate this, I'll change a few parameters in the IOPreparationTable.

- The logical address of the buffer is 0x01B89F80.
- The transfer length is 20480 bytes.
- The transfer granularity is 8192 bytes. This value limits the length of the longest preparation.

PrepareMemoryForIO performs partial preparation of the data three times, as shown in Table 1.

Table 1. Three partial preparations

	Logical Mapping	Physical Mapping	Byte Count
First Preparation	0x01B9F80 0x01BA000	0x0077EF80 0x0077F000	4224
Second Preparation	0x01B8B000 0x01B8C000	0x00780000 0x00782000	8192
Third Preparation	0x01B8D000 0x01B8E000	0x00783000 0x00784000	8064

The entire transfer requires these three repetitions of the five-step transfer process:

1. The driver prepares the first DMA operation for physical address 0x0077EF80, length 4224. After it interrupts, the primary interrupt handler queues a secondary interrupt that, when run, calls CheckpointIO and causes a software interrupt routine to run. This software interrupt routine updates the firstPrepared field from 0 to 4224 (the amount previously prepared) and calls PrepareMemoryForIO for the next partial preparation. When PrepareMemoryForIO finishes, the software interrupt routine calls the secondary interrupt handler.
2. The secondary interrupt starts the next transfer for physical address 0x00780000, length 8192. When this transfer finishes, the primary interrupt queues the secondary interrupt, which, in turn, calls CheckpointIO and causes the software interrupt routine to run a second time. This task calls PrepareMemoryForIO for the next preparation and calls the secondary interrupt handler again.
3. The secondary interrupt handler starts the final transfer. When it finishes, the driver completes the entire preparation.

LOGICAL DATA TRANSFER: PROGRAMMED I/O

Some hardware devices do not support DMA but rather use programmed I/O, in which the main processor moves data between program logical addresses and the device. Programmed I/O is also needed when the device's DMA hardware cannot use DMA in a particular situation or context — for example, a one-byte transfer.

Some hardware devices cannot transfer data that isn't properly aligned to some hardware-specific address value. For example, the DMA controller on the Power Macintosh 8100 requires addresses to be aligned to an 8-byte boundary; it can only transfer to physical addresses in which the low-order three bits are set to 0. Also, data transfers must be a multiple of 8 bytes. To handle such cases, the DMA support library returns the logical addresses of unaligned segments so that a device driver can transfer them with programmed I/O operations.

This restriction on logical alignment means that before starting a DMA transfer, the driver must look at the low-order bits of the physical address and the low-order bits of the count. The actual data transfer process is illustrated by the code in Listing 2, which presumes 8-byte alignment and ignores a few additional complications. The ugly stuff is in the ComputeThisSegment function, which examines the global IOPreparationTable and handles multiple address ranges. The DMA support library simplifies the procedure, as we'll see in the next section.

Listing 2. Data transfer with logical alignment

```
LogicalAddress    thisLogicalAddress;
PhysicalAddress   thisPhysicalAddress;
ByteCount         thisByteCount, segmentByteCount;

ComputeThisSegment(&thisLogicalAddress, &thisPhysicalAddress,
   &thisByteCount);
if ((thisPhysicalAddress & 0x07) != 0) {
   /* Pre-alignment logical transfer */
   segmentByteCount = 8 - (thisPhysicalAddress & 0x07);
   if (segmentByteCount > thisByteCount)
      segmentByteCount = thisByteCount;
   DoLogicalTransfer(thisLogicalAddress, segmentByteCount);
   thisByteCount -= segmentByteCount;
   thisLogicalAddress += segmentByteCount;
   thisPhysicalAddress += segmentByteCount;
}
if (thisByteCount > 0) {
   /* Aligned physical transfer */
   segmentByteCount = thisByteCount & ~0x07;
   if (segmentByteCount != 0) {
      DoPhysicalTransfer(thisPhysicalAddress, segmentByteCount);
      thisByteCount -= segmentByteCount;
      thisLogicalAddress += segmentByteCount;
   }
}
if (thisByteCount != 0) {
   /* Post-alignment logical transfer */
   DoLogicalTransfer(thisLogicalAddress, thisByteCount);
}
```

PUTTING IT ALL TOGETHER

Here we'll take a look at how your driver can use several of the functions in the DMA support library to simplify dealing with PrepareMemoryForIO.

Before you can call any of the functions in the DMA support library to make a partial preparation, you need to create the system context for a software interrupt. This context is created by the CreateSoftwareInterrupt system routine, as shown in the InitializePrepareMemoryGlobals function in Listing 3. CreateSoftwareInterrupt must be called from your driver's intialization routine because it allocates memory. Your driver's interrupt handler uses a software interrupt to start a task that can call PrepareMemoryForIO (as described earlier in step 4 of the data transfer process).

Listing 3. Initialization for DMA

```
SoftwareInterruptID    gNextDMAInterruptID;

/* This function is called once, when your driver starts. */
OSErr InitializePrepareMemoryGlobals(void)
{
   OSErr    status;

   gLogicalPageSize = GetLogicalPageSize();
   gPageMask = gLogicalPageSize - 1;
   status = CreateSoftwareInterrupt(
         PrepareNextDMATask,    /* Software interrupt routine */
         CurrentTaskID(),       /* For my device driver */
         NULL,                  /* Becomes the p1 parameter */
         TRUE,                  /* Persistent software interrupt */
         &gNextDMAInterruptID); /* Result is the task ID. */
   return (status);
}
```

The DMA support library contains two functions that a driver can use to simplify processing the output from PrepareMemoryForIO: InitializeDMATransfer, which is called once to configure the overall transfer operation, and PrepareDMATransfer, which is called to set up each individual transfer.

The MyConfigureDMATransfer function in Listing 4 calls PrepareMemoryIO and InitializeDMATransfer to configure the transfer. This function is called by the mainline driver function (and by a software interrupt routine for partial preparation, as we'll see later).

If MyConfigureDMATransfer is successful, the driver initializes the hardware to begin processing. I assume here that the hardware interrupts the process when it requires a data transfer. The primary interrupt handler is shown in Listing 5.

Listing 4. MyConfigureDMATransfer

```
/* In a production system, kPageCount should be retrieved from the
   operating system by calling GetLogicalPageSize. */
#define kPageCount        4096
#define kLongestDMA       65536
#define kLogicalAlignment 8
#define kMappingEntries ((kLongestDMA + (kPageCount - 1)) / kPageCount)

DMATransferInfo           gDMATransferInfo;
IOPreparationTable        gIOTable;
LogicalAddress            gLogicalMapping[2];
PhysicalAddress           gPhysicalMapping[kMappingEntries];
AddressRange              gThisTransfer;
Boolean                   gIsLogical;
```

(continued on next page)

Listing 4. MyConfigureDMATransfer *(continued)*

```
OSErr MyConfigureDMATransfer(
         IOCommandCode  ioCommandCode,   /* Parameter to DoDriverIO */
         ByteCount      firstPrepared    /* Zero at first call */
   )
{
   OSErr    status;

   gThisTransfer.base = NULL;      /* Setup for programmed I/O */
   gThisTransfer.length = 0;       /* Interrupt handler */
   gIsLogical = FALSE;

   if (firstPrepared == 0) {
      /* This is an initial preparation for the transfer. */
      gIOTable.preparationID = kInvalidID;     /* Error exit marker */
      switch (ioCommandCode) {
         case kReadCommand:  gIOTable.options = kIOIsInput;    break;
         case kWriteCommand: gIOTable.options = kIOIsOutput;   break;
         default:            return (paramErr);
      }
      ioTable.ioOptions |=
         ( kIOLogicalRanges              /* Logical input area */
         | kIOShareMappingTables         /* Share with OS kernel */
         | kIOMinimalLogicalMapping      /* Minimal table output */
         );
      gIOTable.state = 0;
      gIOTable.addressSpace = CurrentTaskID();
      gIOTable.granularity = kLongestDMA;
      gIOTable.firstPrepared = 0;
      gIOTable.lengthPrepared = 0;
      gIOTable.mappingEntryCount = kMappingEntries;
      gIOTable.logicalMapping = gLogicalMapping;
      gIOTable.physicalMapping = gPhysicalMapping;
      gIOTable.rangeInfo.range.base = pb->ioBuffer;
      gIOTable.rangeInfo.range.length = pb->ioReqCount;
   }
   else  {     /* We were called to continue a partial preparation. */
      gIOTable.firstPrepared = firstPrepared;
   }

   status = PrepareMemoryForIO(&gIOTable);
   if (status != noErr)
      return (status);
   status = InitializeDMATransfer(&gIOTable, kLogicalAlignment,
      &gDMATransferInfo);
   return (status);
}
```

When the primary interrupt handler determines that a data transfer is needed, it calls the function MySetupForDataTransfer, which tries to continue a logical (programmed I/O) transfer. If no logical transfer is appropriate, it calls PrepareDMATransfer, to configure the next data transfer segment. This will be either a logical or a DMA transfer, depending on the interaction between the user's data transfer parameters and

Listing 5. The primary interrupt handler

```
InterruptMemberNumber MyInterruptHandler(InterruptSetMember  member,
                                         void                *refCon,
                                         UInt32              theIntCount)
{
   OSErr    status;

   if (<device has or requires more data> == FALSE)
      status = noErr;              /* Presume I/O completion. */
   else
      status = MySetupForDataTransfer();
   if (status != kIOBusyStatus)
      /* This partial transfer (or device operation) is complete. */
      QueueSecondaryInterruptHandler(DriverSecondaryInterruptHandler,
            NULL, NULL, (void *) status);
   return (kIsrIsComplete);
}

OSErr MySetupForDataTransfer(void)
{
   OSErr    status;

   if (gIsLogical && gThisTransfer.length > 0) {
      /* Continue a programmed I/O transfer. */
      DoOneProgrammedIOByte(* ((UInt8 *) gThisTransfer.base));
      gThisTransfer.base += 1;
      gThisTransfer.length -= 1;
      status = kIOBusyStatus;
   }
   else {      /* We need another preparation segment. */
      status = PrepareDMATransfer(&gDMATransferInfo, &gThisTransfer,
            &gIsLogical);
      if (status == noErr) {      /* Do we have more data? */
         status = kIOBusyStatus;  /* Don't queue secondary task. */
         if (gIsLogical) {        /* Start a programmed I/O transfer. */
            DoOneProgrammedIOByte(* ((UInt8 *) gThisTransfer.base));
            gThisTransfer.base += 1;
            gThisTransfer.length -= 1;
         }
         else  /* Start a DMA transfer segment. */
            StartProgrammedIOToDevice(&gThisTransfer);
      }
      else     /* This preparation is done. Can we start another? */
         status = kPrepareMemoryStartTask;
   }
   return (status);
}
```

the device's logical alignment restrictions. If more data remains to be transferred, MySetupForDataTransfer starts either a DMA transfer or another logical transfer; otherwise, it returns a private status value that will eventually cause a software interrupt routine to call PrepareMemoryForIO again to continue a partial preparation.

Listing 6 shows the secondary interrupt handler — at least the part that handles the DMA operation. The primary interrupt handler provides the operation status in the p2 parameter; the secondary interrupt handler uses this parameter to determine whether the operation is complete (in which case this is the final status), or whether some intermediate operation is required.

Finally, Listing 7 shows the software interrupt routine that's called when the driver must call PrepareMemoryForIO again to perform a partial preparation.

Listing 6. The secondary interrupt handler

```
OSStatus DriverSecondaryInterruptHandler(void  *p1,
                                         void  *p2)
{
   OSStatus    osStatus;

   osStatus = (OSErr) p2;
   switch (osStatus) {
      case kPrepareMemoryStartTask:      /* Need more preparation */
         CancelDeviceWatchdogTimer();
         osStatus = SendSoftwareInterrupt(gNextDMAInterruptID, 0);
         if (osStatus != noErr) {
            /* Handle error status by stopping the device. */
            ...
         }
         break;
      case kPrepareMemoryRestart:        /* Preparation completed */
         osStatus = MySetupForDataTransfer();
         break;
   }
   if (osStatus != kIOBusyStatus) {      /* If I/O is complete */
      CancelDeviceWatchdogTimer();
      CheckpointIO(&ioTable, kNilOptions);
      IOCommandIsComplete(ioCommandID, (OSErr) osStatus);
   }
   return (noErr);
}
```

Listing 7. A software interrupt routine for partial preparation

```
void PrepareNextDMATask(void *p1,
                        void *p2)
{
   OSErr       status;
   ByteCount   newFirstPrepared;

   if ((gIOTable.state & kIOStateDone) != 0)
      status = eofErr;          /* Data overrun or underrun error */
   else {                       /* Do the next partial preparation. */
      newFirstPrepared =
            gIOTable.firstPrepared + gIOTable.lengthPrepared;
```

(continued on next page)

Listing 7. A software interrupt routine for partial preparation *(continued)*

```
      status = MyConfigureDMATransfer(0, newFirstPrepared);
                                /* ioCommandCode is not used. */
   }
   QueueSecondaryInterruptHandler(DriverSecondaryInterruptHandler,
         NULL, NULL, (void *) status);
}
```

YOUR TURN IN THE BARREL

At times, working through the complexity of this problem felt like going off Niagara Falls in a barrel. There used to be a joke among the developers of the UNIX operating system: "We never document our code: if it was hard to write, it should be hard to understand." The algorithms I've described here were hard to write, but I hope I was able to document and clarify the most important features of the library well enough that you don't have to go through the same struggle I did.

Thanks to our technical reviewers David Harrison, Tom Saulpaugh, Dave Smith, and George Towner.•

DEVELOPER
DU
UNIVERSITY

Developer University
Apple Computer, Inc.
1 Infinite Loop, M/S 305-1TU
Cupertino, CA 95014

It's not just the basics anymore!

Advanced courses from Developer University get you up to speed quickly on new Apple technologies.

❐ *OpenDoc*
❐ *PowerPC*
❐ *Newton*
❐ *Graphics/Imaging*

Courses are available as:

Self-Paced *Classroom* *Lecture* *Online*

For detailed information, check out http://www.info.apple.com/dev on the World Wide Web, or contact the Apple Developer University Registrar at (408)974-4897 or fax (408)974-0544

Macintosh Q & A

Q *How do I determine whether a Power Macintosh has PCI expansion slots?*

A If there's a Name Registry, you can use it to determine whether a PCI bus exists. To determine whether the Name Registry exists, use the new Gestalt selector gestaltNameRegistryVersion ('nreg'). If the Name Registry exists, the value returned is the version number of the Registry; otherwise, gestaltUndefSelectorErr is returned, and you can assume that the machine doesn't have PCI slots.

If the Name Registry exists, call RegistryEntrySearch to look for an entry having a property name of **device_type** and a propertyValue of **pci**. If an entry is found, there is a PCI bus on the machine.

Q *Our software doesn't awaken properly on a PowerBook that has come out of sleep mode. Are there any special handling requirements to recover from sleep mode?*

A The changes to the system state when a PowerBook goes to sleep include the following:

- All AppleTalk connections are lost, because the AppleTalk driver is turned off.
- The serial ports are entirely shut down to conserve power.

There are two Macintosh Technical Notes that relate to your situation: "Little PowerBook in Slumberland" (HW 24), which provides a brief overview of the sleep process, and "Sleep Queue Tasks" (HW 31), which presents additional material regarding the sleep process. The second one includes sample code that demonstrates a sleep queue task implementation. The sleep queue task enables your program to save state information that otherwise might be lost. Typically, this is important for a networked process that needs to reestablish a connection upon awakening.

Q *Can we define our own extensions to QuickTime's ImageDescription structure? In other words, can we just attach any kind of data to the end of the ImageDescription structure? Our codec would use this data only on the Macintosh.*

A Yes, you can add any extended data you like, with the utility routines provided for this purpose (described in *Inside Macintosh: QuickTime Components*, starting on page 4-65). You have complete control over how your codec interprets the extensions. Therefore, as long as the default image description handle remains intact (for the benefit of the various Movie Toolbox calls that depend on the documented structure being there), you can add whatever information you like. Note that Apple reserves all extension types consisting entirely of lowercase letters.

Q *We're trying to write a QuickTime codec, but we're having trouble because Inside Macintosh: QuickTime Components was written before the universal headers, and the sample codec source doesn't build at all with the latest headers. Where can we get a QuickTime codec that builds for PowerPC under the current universal headers?*

A Until a PowerPC-native codec example becomes available, you can get the information you need from the Macintosh Technical Note "Component Manager version 3.0" (QT 5), which provides details on creating native components. Note that you have to use Resorcerer or Rez to create the component templates; ResEdit won't suffice.

Q *Our codec needs to provide more options to the user than the normal image-compression dialog contains. The documentation suggests that it's possible to provide an extra Options button in the dialog, and I've seen some applications that do provide an Options button for certain codecs. Is this a function of the application? How does the application know to do this?*

A If your codec component has an exported function named CDRequestSettings, the standard image-compression dialog will automatically provide the specific button. In other words, QuickTime checks the codec component, adds the button (provided it's available), tracks clicks in the button, and calls your CDRequestSettings routine appropriately. For further details, see the Macintosh Technical Note "QuickTime 1.6.1 Features" (QT 4) where CDRequestSettings is documented.

Q *We have a non-Macintosh device that creates and reads QuickTime movies, and we need to pass additional information about the images between the non-Macintosh device and our QuickTime codec. It seems that the logical place to put this information is in an ImageDescription extension (within the sample description atom), since this is about all that's accessible to a codec. Is the format of this extension documented anywhere? We've looked at the extension created by SetImageDescriptionExtension, and the format seems simple, but it would be nice to know what the "official" format is.*

A Chapter 4 of *Inside Macintosh: QuickTime* has a listing of the atoms and their formats. Sample description atoms are described on page 4-35. Note that each media format has its own sample description tables, which are not directly accessible.

The official guideline is to use, if possible, the provided APIs for creating sample description atoms. If you're working on a platform for which there are no Toolbox APIs, you'll have to obtain a source-code license agreement to get real source code showing how the atoms are constructed. (For details regarding licensing part or all of the QuickTime source code, contact Apple Software Licensing at AppleLink SW.LICENSE or (512)919-2645.)

Q *Our application plays QuickTime movies. Some older movies played well in System 7.1, but they don't play properly in System 7.5 or 7.5.1. We happened to find the Apple Multimedia Tuner, and it solves the problem. What is the Apple Multimedia Tuner, who needs it, how does a customer get it, and can we distribute it?*

A The real solution to your problem is just to preroll the movie before playing it, which is what the Apple Multimedia Tuner is doing for you. QuickTime 2.1 incorporates all the Tuner improvements, so there's no longer any need to distribute the Tuner separately.

Q *We have a problem when we draw to an offscreen GWorld under low-memory conditions (when the system heap can't grow) on a Power Macintosh. The GWorld drawn contains digital noise. The same code works just fine in an 680x0 environment. Any idea what's happening?*

A It sounds as if the Code Fragment Manager is unable to load the code from the PowerPlug library into temporary memory. This will cause QuickTime to issue a noCodecErr error. You should always try to catch QuickTime-generated errors, checking, for instance, for playback errors after each MoviesTask call like this:

```
anErr = GetMoviesStatus(Movie theMovie, Track *problemTrack);
```

Here's a possible workaround to your problem: Launch a small application that has the QuickTimeLib (PowerPlug) library statically linked in, so that it's loaded. This application should launch the main application and then kill itself. The second application could try to grow to a predefined size and handle low-memory conditions in whatever way it wants, but the CFM libraries are already in memory by then.

The Code Fragment Manager will never load fragments into an application heap, because there's a global registry of CFM libraries present. If another application registers to use a CFM library that's in an application heap that subsequently goes away, this will obviously be a Bad Thing. In the 680x0 environment, the codecs are components, and the Component Manager will always try to load components into the application heap if the system heap doesn't have any available space.

Q *I need to add print items to a QuickDraw GX dialog box. In attempting to use the Experiment no.9 sample, I found what appears to be a bug. This example uses GXGetMessageHandlerResFile when it calls GXSetupDialogPanel, but it should call CurResFile.*

A You're right. Applications should call CurResFile. GXGetMessageHandlerResFile is reserved for extensions and drivers.

For additional code examples that add print items to a QuickDraw GX dialog box, see the Worldwide Developers Conference 1995 Technology CD (or the Mac OS Software Developer's Kit). The Extension Shell, UserItems, and Additions samples provide the basic item adding/handling code that you require.

Q *Where can I find some good sample code that demonstrates the techniques required for a "panel" with QuickDraw GX printing (as an application — not an extension)?*

A There are two sample applications ("Experiment no.9" and "Banana Jr.") that show how to do this. In both of these applications, the panels appear in the Custom Page Setup dialog. However, the sample code can easily be modified to add panels to the Page Setup and Print dialogs.

Q *How can I draw and print hairlines with QuickDraw GX? We use a picture comment in the normal print code, but this seems to make QuickDraw GX fail. We get a -51 error (reference number invalid) when we call GXGetJobError after calling GXFinishJob, and we sometimes get this error without the picture comment code.*

We also tried calling GXSetShapePen in our spool procedure. When we set it to a fractional value, we get a wide line, but when we set it to a wide value, such as 8, it works properly. What do we need to do to print fractional widths?

A Here are two ways to get QuickDraw GX to draw hairlines when printing:

- Call GXSetShapePen(myShape, 0). This sets your pen width to 0, meaning as thin as possible on the output device. QuickDraw GX always draws hairlines at the resolution of the output device — one pixel wide.
- Call GXSetStylePen(myStyle, 0). This also sets your pen width to 0, with the same result.

When using GXSetShapePen and GXSetStylePen, don't specify the pen width as an integer: remember that it's a fixed-point value. GXSetStylePen(myStyle, 1) sets the pen width to 1/65536; GXSetStylePen(myStyle, ff(1)) sets it to 1.0.

QuickDraw GX uses a backing store file (an invisible file within the System Folder) to send QuickDraw GX objects to disk when additional space is needed within the QuickDraw GX heap. Almost all -51 errors from within QuickDraw GX or an application using QuickDraw GX are caused by double-disposing of a QuickDraw GX object (that is, a shape, ink, style, or transform). The -51 error occurs because the double dispose causes QuickDraw GX to set the shape attributes, which indicates that it has sent the object to disk. When it needs this object, it goes to the backing store and tries to get it, but it's not there. We've found a few cases where QuickDraw GX itself was double-disposing of objects, and these were fixed in QuickDraw GX version 1.1.

Before calling GXDrawShape, call GXValidateShape on the shape or shapes you're trying to print. This ensures that a shape is valid before it's drawn or printed. It slows things down a little, but you'll be able to determine whether a shape is still available before you attempt to draw it (you might be disposing of a shape before you draw it). If you have an error handler installed, you usually receive the "shape_already_disposed" message, but you may not receive this message if something is wrong with the QuickDraw GX backing store.

It's also possible that the hairline drawing problems you're encountering are related to the translation options you're using. A translator takes your QuickDraw drawing commands and converts them to QuickDraw GX objects, based on options you provide. If you use the gxDefaultOptionsTranslation setting, a QuickDraw line turns into a six-sided filled polygon. When your object is a polygon, changing the pen width has no effect.

To avoid translation problems, call GXInstallQDTranslator with the gxSimpleGeometryTranslation or the gxReplaceLineWidthTranslation option.

- gxSimpleGeometryTranslation turns on both the simple-lines and simple-scaling translation options, and it translates QuickDraw lines into QuickDraw GX lines with flat endcaps. The QuickDraw GX line shape runs along the center of the original QuickDraw line, and it covers all the pixels of the QuickDraw line and more.
- gxReplaceLineWidthTranslation turns a QuickDraw line into a QuickDraw GX line with a width that is the average of the original pen's width and height. This option also affects the way the SetLineWidth picture comment is interpreted.

Once you set the translation option, your calls to GXSetShapePen or GXSetStylePen should behave as you expect them to, because they're acting on QuickDraw GX lines, not polygons. When you've installed a translator, be sure to remove it with GXRemoveQDTranslator. To learn more about the translation options, see Chapter 1 of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

Q *I'm trying to send messages from within a QuickDraw GX message override. I want to send GXWriteData to flush the buffer so that I can send the GXGetDeviceStatus message. I override the GXHandlePanelEvent message. In my override, sending messages causes the system to crash. What would cause this to happen?*

A The crash is occurring because there's no connection to the printer at the time you're sending the message. You have to send the GXOpenConnection, GXWriteData, and GXCloseConnection messages. Note that when you send GXOpenConnection, QuickDraw GX puts up the default job status dialog for a short time. If you don't want this dialog to appear, you can override the GXJobStatus message to prevent it from being shown. See also Dave Hersey's Print Hints column, "Writing QuickDraw GX Drivers With Custom I/O and Buffering," in *develop* Issue 21.

Q *I used the sample driver showing how to do custom dialogs as the basis for the compatibility part of our QuickDraw GX PostScript driver, and I added an Options dialog to it for our printer-specific features. I have two problems with it when using applications that aren't QuickDraw GX–aware. First, the paper-type always defaults to the fifth paper-type listed in the resource file, so whichever paper-type is the fifth one listed becomes the default paper-type in the QuickDraw GX compatibility driver. This is, of course, reflected in the Page Setup dialog. Second, the driver always defaults to having the "Print to File" checkbox on. What can I do about these problems?*

A Both the quirks you describe (improper default paper-type and the "Print to File" checkbox defaulting to on) can be fixed by modifying the 'PREC' 0 resource in the driver.

When an application using old-style printing calls PrintDefault to request the default print record from the current printer driver, the driver gives it the contents of the 'PREC' 0 resource. Then, when the application calls PrJobDialog or PrStlDialog, it passes in that print record. In its overrides, the QuickDraw GX printer driver interprets the contents of the old-style print record to set up the states of the dialog's buttons, checkboxes, and so on.

To determine which paper-type radio button to select in the Page Setup dialog, QuickDraw GX compares the page rectangle specified in the old-style print record to the rectangles of all the paper-types in the driver (or paper-type extensions, such as "3-Hole Punch"), and tries to find the best match. Because of the way that the old-style print record in the sample is defined, that best match turns out to be the fifth paper-type in your list. So, to fix this quirk, all you have to do is change the bounds setting in the 'PREC' 0 resource so that it matches the bounds of the US Letter paper-type in the driver.

To determine the state of the "Print to File" checkbox, the driver looks at the UlOffset field of the old-style print record. (You might not think to look here, but old-style print records are limited to 120 bytes, and there was no better place to store this information.) Because the 'PREC' 0 resource in this driver has this field set to 1, the checkbox defaults to on. So, to fix this, all you have to do is set the field to 0.

Q *I want to create an extension for the Page Setup/Format dialog that performs "flipping" functions. Is it feasible to create an extension for the Page Setup/Format dialog rather than the Print dialog?*

A There's nothing to prevent you from creating an extension that adds a panel to the Page Setup dialog. Most printing extensions add to the Print dialog because in most cases this is the proper place to add a panel that affects the entire output, and because what extensions usually do is best suited for the Print dialog. Drivers and applications, on the other hand, typically add to the Format dialog. Note that if you're trying to modify an existing sample extension so that it adds to the Page Setup dialog, you have a bit of work to do.

There's a way that you can test your flipping code without writing a new extension, by the way. Applications can override the GXJobDefaultFormatDialog or GXFormatDialog message. There are two examples ("Experiment no.9" and "Banana Jr.") that demonstrate overriding GXFormatDialog. You might try adding your flipping code to one of these.

Q *A car passed me the other day with one of those round white country stickers that said WAL. Where was it from?*

A Sierra Leone.

Q *In QuickDraw 3D, when we have the interactive renderer on and we try to turn off the draw context's clearImageMethod (setting it to kQ3ClearMethodNone), it still clears. This works properly with the wireframe renderer, but we need this feature in the interactive renderer, since we're pasting in background pictures that we want to act as a backdrop to our 3D models. The interactive renderer always obliterates the background with the clearImageColor. What can we do?*

A Unfortunately, this is a renderer-dependent feature that's supported by the wireframe renderer, but not the interactive renderer. We intend to provide a "Clear with picture" method in the next major release of QuickDraw 3D (version 1.1).

Q *The interactive renderer doesn't draw flat surfaces that are parallel to the camera view direction with the orthographic camera, but the wireframe renderer does. We put in a "floor" of polygons, and when we look along the edge of the floor with the orthographic camera, it totally disappears. With the wireframe renderer, we see a line where the floor is, which is as expected. What gives?*

A Filled primitives have no thickness, so when you look at them edge-on, they do not appear. Lines, however, are a mathematical abstraction, so they always appear to be one pixel thick (when you zoom in on a line, its thickness doesn't increase). While this may seem somewhat odd, it's the way many libraries work. To achieve the effect you want, make the floor a thin box, and texture-shade the top surface. If the depth of the box is nonzero, it appears to be a slab-like structure, and it won't disappear when viewed edge-on.

Q *If we iterate through the vertices in a mesh, will the vertices still be in the same order as they were when they were added?*

A Yes. The ordering of the vertices doesn't change until you duplicate the mesh or write it out. A duplicated mesh (or one that was written out and read back in) doesn't necessarily have the vertices in the same order as when they were added.

Q *When I try to render models with different types of lights, the point light and the directional light work correctly, but the spot light doesn't. Any idea why?*

A The spot light's cone of light needs to touch a number of vertices for any effect to be seen. If the light is attenuated, it may have insufficient intensity when it strikes the surface. The cone of light also needs to be wide enough to cover a significant area of the object being modeled for the renderer to draw a reasonable effect.

Q *What effect does the TQ3ViewObject parameter have in the bounding box calculating routines (Q3View_StartBoundingBox and Q3View_EndBoundingBox)? The old geometric-object routine descriptions refer to world space, but if this is so, there's no need for a view parameter. However, if the view's camera is used, the bounding box is returned in camera coordinates rather than view coordinates. Since both are useful, would it be possible to have both sets of routines available? I can apply a rotation/translation matrix to all of the items to be drawn to generate camera coordinates from a world coordinate routine, but I need to find out if I need to do this or if this has already been accomplished.*

A The QuickDraw 3D routines return the bounding box or bounding sphere in local coordinates. Part of the reason that the API was modified to use submit calls, rather than having separate picking, rendering, and writing calls, is that the transformations that are applied matter more than the camera. Since these modifications were made, the submit calls for everything (including transformations, if they're not stored in the group) can be in one submission function that's called from inside the picking, rendering, or writing loop. If you need the bounding box for a single geometry in its own coordinate space, this is also easy to do — you can write a simple routine that performs bounds calculations on a single object. For example:

```
Q3View_StartBoundingBox;
Q3xxx_Submit;
Q3View_EndBoundingBox;
```

Q *Does QuickDraw 3D prefer meshes or NURB patches? Which kind of data yields better performance?*

A Meshes are convenient for editing, but they take quite a bit of memory, so the tradeoff is time versus space. NURB patches are more convenient for dealing with surfaces as a whole and for representing surfaces at different tessellations.

Although meshes exhibit better performance than NURB patches in the first version of QuickDraw 3D, later versions may have improved patch performance. In the meantime, consider experimenting with the tessellation factor for your patches, since overtessellating reduces performance.

Q *I'd like to make sure that I'm running under version 1.0.2 of QuickDraw 3D. When I get the version from Q3GetVersion the major version is 1 and the minor version is 0, but I can't get the revision (the third number). Is there a Gestalt selector for this?*

A Starting with version 1.0.2, there is a Gestalt selector to get the version of QuickDraw 3D: gestaltQD3DVersion. The return value has two bytes for the major version, a byte for the minor version, and a byte for the revision. So for version 1.0.2 Gestalt will return 0x00010002. Note that this Gestalt selector works only with QuickDraw 3D 1.0.2 and later.

Q *The ColorSync documentation (in the reference section of Inside Macintosh: Advanced Color Imaging) states that each color component in the L*a*b* color space is within the* *range of 0 to 65,280. Shouldn't this be 0 to 65,535, since this is the value for the other spaces and the value in the ICC Profile Format Specification?*

A No. The correct maximum value for this particular color space is 65,280 (0xFF00). Note that the final documentation is now available as *Advanced Color Imaging on the Mac OS*, published by Addison-Wesley.

Q *What exactly are the internal parameters for the ColorSync quality settings? That is, how large a lookup table is built for "draft" versus "normal" versus "best"?*

A The quality flag bits provide a place in the profile for an application to indicate the desired quality of a color match (potentially at the expense of speed and memory). In ColorSync 2.0, these bits do not mandate the use of one algorithm over another, or one lookup table size over another; they're just recommendations that a particular CMM may choose to ignore.

Let's look at how the default Apple CMM uses the quality recommendations specified in the flag bits. Other CMMs, of course, will have different implementations.

When Apple's CMM builds a color world from two or more profiles, and one or more of these profiles contain TRC curves or A2Bx tables, the CMM also builds a private, multidimensional lookup table. The quality flag bits determine the resolution of this private table. Draft quality is treated the same as Normal quality, so there are really only two effective settings, Normal/Draft and Best. In most cases, the quality is only slightly better in Best mode, so the difference is difficult to see, unless one of the profiles has a high gamma value. For high gamma values, the extra resolution in the lookup table is helpful.

Best mode typically takes twice as long to build a color world (about two seconds, versus one second in Normal/Draft mode). However, once the color world is built, the time to use it is the same for either mode (approximately 1.5 MB/second on a Power Macintosh 8100/110).

Best mode also requires significantly more memory than Normal/Draft mode. A color world typically requires 120K of heap space in Best mode versus 25K in Normal mode, and the "high-water" memory requirement while a color world is being built is typically 300K for Best mode versus 90K for Normal mode.

Note again that these guidelines apply only to the default Apple CMM. The tradeoffs between speed, quality, and resources may be quite different for other CMMs.

Q *I want to go directly from an input CMYK space to an output CMYK space (without going through an intermediate three-component space) to preserve the original GCR/UCR settings. Can I create a "link" profile for this purpose? If I do, will I have to write my own CMM to use it?*

A You can build a CMYK-to-CMYK device-link profile for this purpose, and you can use it without writing your own CMM.

Q *I'm using the ColorSync call CWCheckBitMap to do gamut checking in a plug-in for Photoshop. The result bitmap is not what I expected, and seems to be different every time I try it. Any idea what could be going on?*

A CWCheckBitMap sets each pixel in the result bitmap to black if the corresponding pixel in the source bitmap is out of the gamut. It doesn't, however, set each pixel in the result bitmap to white if the pixel in the source bitmap is in the gamut. If you aren't erasing the bitmap before calling CWCheckBitMap, that would explain what you're seeing. Always erase the result bitmap to white before calling CWCheckBitMap. (This is also true of CWCheckPixMap and CWCheckColors.)

Q *If I have a physical drive ID, how can I determine whether that drive is a network volume? I'm not sure where to look, and I need to know whether the information is dependable and not subject to change.*

A Under the current Macintosh file system, there's no completely dependable way to determine whether a volume originates over a network or is implemented on a local disk. This is the result of the way external file systems are implemented — a third party can build a network file system in a variety of ways.

You can, however, easily determine whether a volume uses the AFP (AppleShare) file system, which in many cases is adequate. To make this determination, compare the driver refNum in the drive queue entry to the AppleShare client's refNum.

The following code enumerates the drive queue and displays the relevant information:

```
main()
{
   QHdrPtr     drvQHdr = GetDrvQHdr();
   DrvQElPtr   dqeP;
   short       afpRefNum = 0;
   OSErr       errNo;

   // Get the driver refNum for AFP.
   errNo = OpenDriver("\p.AFPTranslator", &afpRefNum);
   if (errNo != noErr)
      return

   // Scan each drive in the drive table.
   dqeP = (DrvQElPtr) drvQHdr->qHead;
   do {
      // Is it an AFP volume or SCSI device?
      if (dqeP->dQRefNum == afpRefNum) printf("AFP");
   } while (dqeP =(DrvQElPtr) dqeP->qLink);
}
```

For other third-party file systems, such as DECNET and NFS, you have to determine the name of the driver and then compare it to the AppleShare client's refNum.

Q *I need to get a list of files in a particular directory. Should I use PBCatSearch, or should I use indexed PBGetCatInfo or PBGetFInfo requests?*

A The "Cat" in PBCatSearch stands for "Catalog" and that's what PBCatSearch searches: the whole volume catalog. You can specify that matches found by

PBCatSearch be limited to a specific directory by setting the fsSBFlParID bit in the ioSearchBits field of the parameter block, and then specify the directory to match on by setting ioFlParID in ioSearchInfo1 and ioSearchInfo2 to the directory ID you're interested in. However, PBCatSearch may not be what you want to use, for a couple of reasons:

- The matches PBCatSearch finds by matching based on ioFlParID are only in that one directory, not in any of that directory's subdirectories.
- Because the whole catalog file is searched, this is usually not the fastest way to look through a specific directory's contents.

If you need matches in both the directory and its subdirectories and you don't want to search the whole volume, there's a routine in the MoreFiles sample code named IndexedSearch that's compatible with PBCatSearch's parameter blocks, except that IndexedSearch lets you specify what directory you want to search. It uses indexed PBGetCatInfo calls to search a directory and its subdirectories.

If you need matches from only a single directory (and not from that directory's subdirectories), you can use the MoreFiles routine named GetDirItems. This routine uses PBGetCatInfo to index through a directory's entries and returns FSSpecs to the entries found. In this case, making indexed PBGetCatInfo calls is much faster than searching the whole catalog with PBCatSearch.

Q *I need to nest two CustomGetFile dialogs, but I'm running into trouble. Under some circumstances after the user dismisses the second dialog (usually via the Cancel button), I lose all of the custom controls in the first dialog. What's happening?*

A The Standard File Package is not reentrant, so there really isn't a way to nest standard file dialogs that will work right. The real problem is in the resources that the Standard File Package uses for the dialog items. When the second, nested dialog closes, it releases resources that the first dialog is still using; that's why your items are getting messed up.

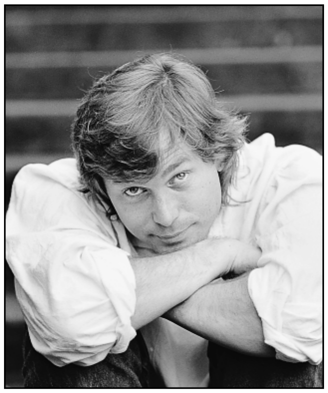
There's a kludgy workaround, but it will break under future systems. You could, however, use sequential calls to the Standard File Package instead of nesting them. This is a bit of a pain, but should accomplish what you want. Here's how: Put up the first dialog. In your filter routine, when the user clicks the control that is to bring up the nested dialog, set a flag in your application signifying "bring up other," and tell the Standard File Package that you're done with the first dialog by passing item 1 or 2 back. After you put up the second dialog and process it, bring the original dialog back. This will be a little messy cosmetically as the dialogs open and close, but it's the only way to do it in a manner that will remain compatible.

Q *What's the best way to remove an attached leech?*

A The best way we know of is to rub a freshly cut lemon or lime on it. Most leeches will detach immediately, and die a writhing, horrible death shortly afterward. Fire and salt are also said to be effective.

These answers are supplied by the technical gurus in Apple's Developer Support Center.•

Have more questions? See the Macintosh Technical Q&As on this issue's CD. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)•



DAVE JOHNSON

THE VETERAN NEOPHYTE

The Right Tool for the Job

Dynamic programming languages are cool. Once you've tasted dynamic programming, it's hard to go back to the old, crusty, static way of doing things. But the fact remains that almost all commercial software is still written with static languages. Why?

Recently I took a class in Newton programming. For me personally the Newton isn't a very useful device, only because I never carry around a notepad or calendar or address book or to-do list and I don't have a need to collect any sort of data out in the field. But even though it's not terribly useful to me, it *is* very useful to a lot of people — and useful or not, it's a really *cool* device. Programming the Newton, for those of you who haven't had the pleasure, is very, very different from programming the Macintosh in C or C++ or Pascal, and is incredibly attractive in a lot of ways.

The language that you use to program the Newton, NewtonScript, is an example of an object-oriented dynamic language, or OODL. (See? Even the acronym is cool.) This means a number of things, but the upshot is that it's very programmer-friendly and very flexible. Now, I don't pretend to be an expert in languages, not by a long shot, so I can't offer any incisive comparisons with other "modern" languages, but I *can* tell you what it feels like for a dyed-in-the-wool C programmer to leap into this new and different world. It feels *great.*

One well-known feature of dynamic languages is garbage collection, the automatic management of memory. Objects in memory that are no longer needed are automatically freed, and in fact there is no way to explicitly free them other than making sure that there are no references to them any more, so that the garbage collector can do its thing. I didn't fully realize how much time and effort and code it takes to deal with memory management until I didn't have to do it anymore. There's something almost naughty about it, going around cavalierly creating objects in memory without worrying about what to do with them later. After a lifetime of living in mortal fear of memory leaks, it feels deliciously irresponsible. I like it. I like it a lot.

NewtonScript's object model is refreshingly simple and consistent. There are the usual "simple" data types — integers, real numbers, Booleans, strings, and so on — and only two kinds of compound objects: *arrays* and *frames*. An array, as you might expect, is simply a linear, ordered group of objects, and the individual objects are referenced by their index (their position in the array). Frames are an unordered collection of items in named *slots*; you refer to a particular item by the name of its slot. Frames are also the only NewtonScript objects that can be sent messages, and the message is simply the name of a slot that contains a function.

Because NewtonScript is dynamic, variables or frame slots or array members can hold any kind of data, including other arrays or frames, or even functions, and the kind of data can be changed at any time. The size of the array or frame can be changed anytime, too; you can add or delete items as needed, without worrying about managing the changing memory requirements. This kind of flexibility is a big chunk of what makes dynamic languages so, well, dynamic. Such a thing is of course unimaginable in a static language, where each byte must be explicitly allocated *before* it's needed, carefully tracked while used, and explicitly deallocated when you're done with it.

NewtonScript is also *introspective*, meaning that all objects "know" all about themselves. (Isn't that a nice term? I like the idea of a language being introspective — sitting there, chin in hand, pondering itself.) The type of a piece of data is stored with the data, and named items keep their names. In fact, everything in memory is coherent, with a well-defined identity; there is no possibility of undifferentiated bits getting schlepped around, no possibility of a dangling pointer or a string being interpreted as a real number. In static languages,

DAVE JOHNSON recently enrolled his smallest dog — named Io (eye-oh) but affectionately called The Stinklet — in an agility class. Dog agility is a sort of obstacle course for dogs, with ramps and jumps and tunnels and poles to climb and leap over and crawl and weave through. Dave got so involved that he started building agility courses in the living room. He came to his senses, thankfully, before creating any permanent installations.•

Dave is easing up on his working life: beginning with the next issue, he'll be working 3/4 time. He had to give up some things, and it was decided (reasonably enough) that helping to edit the rest of *develop* was more important than writing this column. Look for guest Neophytes in coming issues, with perhaps the occasional installment from Dave.•

of course, all that design-level information is thrown out at compile time, and doesn't exist in the running program at all. There's nothing *but* undifferentiated bits, really. What a mess.

And that means that debugging, for the most part, has to take place at the machine level. By the time the program is running, it's just a maze of pointers and bytes and instructions, fine for a machine but nasty for humans. Of course, to combat this we have elaborate, complex programs called source-level debuggers. They give you the sense that the names still exist, thank goodness, but it's just a trick, and depends on an external file that correlates symbols with locations in memory. If you don't have the symbol file, you're out of luck. (Confession time: In my regular C programming I avoid low-level debugging like the plague. Usually I'd rather spend an hour in a source-level debugger than spend five minutes in MacsBug — I know, I know, I'm a wimp — precisely because all the information that helps me to *think* about my program, the names and so on, still "exist" in the source-level debugger. In NewtonScript, there isn't even such a thing as low-level debugging! All that design information is right there in the guts of the running program. Hallelujah!)

With dynamic languages like NewtonScript, you can let go of the details of the machine's operation, and deal with your program's operation instead — you can think at the design level, not the machine level. And it's an incredible relief to float free of the bits and bytes and pointers and handles and memory leaks and messy bookkeeping. Most of the ponderous baggage that comes along with writing a computer program goes away. I mean really, how much longer must we approach the machine on *its* terms when we want to build something on it? Users were released from that kind of bondage to the machine's way of doing things long ago. So what are we waiting for? Obviously we can't program the Macintosh in NewtonScript (more's the pity) but why aren't we all chucking our C++ compilers in exchange for Prograph or Lisp or Smalltalk or Dylan? Well, some of us are. But I think there are two major hurdles to overcome before dynamic languages become mainstream: the need for speed, and inertia.

Dynamic languages carry their own baggage, of course. In the same way that making the Macintosh easier for people to use made it harder to program because the complexity and bookkeeping were shunted behind the scenes, making programming languages easier to use also requires new behind-the-scenes infrastructure and complexity. (*Somebody* has to do the memory management, after all.) This usually results in a bigger memory footprint and slower execution. For "normal" operations, we're long past the point where that mattered: the hardware is beefy enough to handle it without blinking. But software *always* pushes the limits of the hardware. Consequently, there are still times when it's important to squeeze every drop of performance out of the machine. And dynamic languages are just not very good at that. (I don't think you'd want to write your QuickDraw 3D renderer in Lisp.) So any dynamic language that hopes for mainstream commercial acceptance had better have a facility for running hunks of "external" code. That way you could write the bulk of your program in a dynamic language, but still be able to write any time-critical parts in your favorite static language and plug them in. You'd lose the protection of the dynamic language when running the external code, but that's a reasonable tradeoff.

Inertia is the other big problem. People, once they know one way to do something, are often loath to change it, especially if they've been doing it that way for a long time. I'm guilty of this in my own small way: every time I learn a spiffy, liberating new way to program I think I'll never go back to the "old" way. But the next time I set off to write some code I automatically reach for the *familiar* tools, not the new ones. (Lucky for me, the *only* way to program the Newton is in NewtonScript.)

Fortunately, neither one of these hurdles will stop the evolution of our tools. It's unstoppable, if perhaps slower than we might like. There's already a whole spectrum of tools available. From Assembler to AppleScript, Pascal to Prograph, there are tools that allow anyone with enough interest to teach their computers to do new things. The line between users and programmers continues to blur, and dynamic languages can only help that process. I love the thought of putting programming tools into the hands of "nonprogrammers" — kids, artists, hobbyists — and seeing what they come up with. You can bet it will be something new, something that people tied to the machine would never have thought of. I can't wait.

RECOMMENDED READING

- *Unleashed: Poems by Writers' Dogs,* edited by Amy Hempel and Jim Shepard (Crown, 1995).

Thanks to Lorraine Anderson, Jeff Barbose, Paul Dreyfus, 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 or eWorld, or dkj@apple.com on the Internet.

Newton Q & A: Ask the Llama

Q *The on-line discussion groups for Newton developers have a lot of references to compatibility these days. My application works fine on the 120, 110, and 100 models. Does that mean I'm compatible?*

A Good question. Compatibility doesn't mean your application works now, but that it's written in such a way that it will work on future Newton devices and operating systems. There are several APIs and methods for doing things on the 120, 110, and 100 models that will work with them but are not necessarily compatible with future releases of the operating system.

There are two main points to observe for the sake of compatibility:

- If it's not documented, don't use it.
- Catch exceptions; they *can* occur (especially if you ignore the first point).

Since compatibility is such an important question, it will be the focus of this column. The rest of the column will cover the most common breaches of compatibility. Where applicable, there will be an example of the incompatible and compatible ways of doing things. After reading this and making copious notes (especially where you find yourself saying "Oh dear" and "Oh no!"), you'll be in a position to make your code compatible. We also recommend that you try out your application with the Compatibility App Package (which is on this issue's CD and is available from various on-line services).

Note that we refer often to the Newton Toolkit platform file functions. The Toolkit documentation and platform file release notes describe these functions, which are provided in lieu of future APIs. You should use these platform file functions where applicable. Call the code directly and don't modify it. That is, use the **call/with** syntax; don't place the code in a slot in your application and use message sending.

UNDOCUMENTED GLOBAL FUNCTIONS

There are four common offenders here: CreateAppSoup, SetupCardSoups, MakeSymbol, and GetAllFolders.

The function kRegisterCardSoupFunc in the platform file replaces CreateAppSoup and SetupCardSoups. It's much simpler to use than the undocumented functions:

```
// RIGHT way
constant kSoupName := "MySoup:MYSIG";
constant kSoupIndices := '[];
constant kAppObject := '["Item", "Items"];
call kRegisterCardSoupFunc with
   (kSoupName, kSoupIndices, kAppSymbol, kAppObject);

// *** WRONG way ***
CreateAppSoup(kSoupName, kSoupIndices, EnsureInternal([appSymbol]),
   EnsureInternal(kAppObject));
AddArraySlot(cardSoups, kSoupName);
AddArraySlot(cardSoups, kSoupIndices);
SetupCardSoups();
```

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-related questions to NewtonMail or eWorld DRLLAMA or to AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.•

The fix for MakeSymbol is to call the Intern function: it does the same thing as MakeSymbol and it's documented.

There's no replacement function for GetAllFolders; just don't call it.

UNDOCUMENTED GLOBAL VARIABLES

The three most common misused global variables are **cardSoups**, **extras**, and **userConfiguration**.

There are two uses of **cardSoups**: one is to register a card soup; the other to unregister it. Registering is taken care of with kRegisterCardSoupFunc (see above). Unregistering is done with another platform file function, kUnRegisterCardSoupFunc:

```
// RIGHT way
call kUnRegisterCardSoupFunc with (kSoupName);

// *** WRONG way ***
SetRemove(cardSoups, kSoupName);
SetRemove(cardSoups, kSoupIndices);
```

You should never access the **extras** global variable. Not only is this variable undocumented, but so is its format. Both are subject to major revisions. The platform file function kSetExtrasInfoFunc is provided for setting information about items in the extras drawer. The most common use of this function is to give your application a different icon (see the ExtraChange DTS sample code on the CD).

There are also platform file functions to manipulate **userConfiguration**:

- kGetUserConfigFunc gets a slot from the userConfiguration soup entry.
- kSetUserConfigFunc lets you set user configuration information.
- kFlushUserConfigFunc should be called when you've changed user configuration information.

```
// RIGHT way
local userName := call kGetUserConfigFunc with ('name);
if userName then
begin
   if StrEqual(userName, "Doctor") then
      call kSetUserConfigFunc with ('name, "The Doctor");
   call kFlushUserConfigFunc with ();
end;

// *** WRONG way ***
if userConfiguration.name AND
      StrEqual(userConfiguration.name, "Doctor") then
   userConfiguration.name := "The Doctor";
```

UNDOCUMENTED SLOTS AND METHODS

This is a broad category of problems. The most common is **keyboardChicken** in the root view. But there are others, like **cursor.current**, **paperRoll.dataSoup**, **dockerChooser** in the root view, **UnionSoup:Add**, and anything in a built-in application. Unfortunately, there is no right way to access most of these. The exceptions are **cursor.current** and **Add**.

```
// RIGHT way
local currentEntry := cursor:Entry();
myUnionSoup:AddToDefaultStore(anEntry);

// *** WRONG way ***
local currentEntry := cursor.current;
myUnionSoup:Add(anEntry);
```

Also, don't rely on the routing slips, such as **mailSlip** and **printSlip,** being in the root view. You can, however, still use those symbols in your routing frame.

UNDOCUMENTED MAGIC POINTERS

If you use one of these, you know it. Just think what would happen if the magic pointer changed from a view to a string: you would get some pretty bad behavior. Note that most of this could be dealt with by catching exceptions.

STORE AND SOUP ASSUMPTIONS

All you can assume is that store 0 is the internal store. You can't rely on there being only one other store, nor can you rely on the position of a store in the array returned by GetStores. Also, don't assume that another store is a card or even that there is just one store per card.

If you support moving or copying items between stores, you shouldn't find the title of the store. Use the constant ROM_cardAction as provided in the platform file:

```
// RIGHT way
routingFrame := {
   print: ...
   ...
   card: ROM_cardAction
}
```

In addition, don't assume that your soup will exist on every store. Currently, if you register your union soup, it's automatically created on every store that enters the Newton; however, this may change in the future:

```
// RIGHT way
GetUnionSoup(kSoupName):AddToDefaultStore(anEntry);

// *** WRONG way ***
aStore:GetSoup(kSoupName):Add(anEntry);
```

Remember that AddToDefaultStore or Add could throw exceptions. Wrap your calls to these functions in exception handlers.

Finally, if you support the soup change mechanism, don't assume that the change is adding or deleting an entry. It could be something else, such as a soup being created or removed from a store.

SCREEN SIZE

Don't assume the screen is any particular size. It could be larger or smaller than current devices. It could also be wider than it is tall. Your application size setup routine (usually in the viewSetupFormScript) should take this into account. Have maximum and minimum sizes. Close your application if it can't handle the current screen size.

```
// Code to close your application
constant kUnsupportedScreenSize :=
   "WiggyWorld does not support this screen size";

DefConst('closeMeFunc, func(x) x:Close());

:Notify(kNotifyQAlert, EnsureInternal(kAppName),
   EnsureInternal(kUnsupportedScreenSize));
AddDeferredAction(closeMeFunc, [self]);
```

UNDOCUMENTED FEATURES OF DATA TYPES

Rely only on the features and details of built-in data types that are documented. There are three common problem areas: order of slots in a frame, precision of integers, and implementation of strings.

The order of slots in a frame is undefined. It just so happens that in the current implementation the first 20 slots are returned in the order added. This is not a documented feature, so don't rely on it.

Integers are documented as having at least 30 bits of precision. This doesn't mean they'll always be 30 bits; they could be wider (as anyone who has used compiled NewtonScript can tell you). Note that compiled NewtonScript integers may not be 32 bits; they also follow the "at least 30 bits" rule.

The biggest offender is assumptions about how strings are implemented. Don't rely on strings being null terminated or being composed of two-byte Unicode characters. The practical upshot is that you should use StrLen to find the length, and StrMunger (or &) for length changes. Don't use Length, SetLength, or BinaryMunger with strings. Don't use the array accessor to set a string; you can check a character, but don't set a character.

MISCELLANEOUS BITS

Don't send messages directly to the IOBox; use the kSendFunc platform file function. Nor should you read the items in the IOBox soups.

Also note that there are platform file functions to register and unregister for Find that you should use.

Always use SetValue when you're changing the view or other system values.

Use only the **body** slot in items that you route. Don't assume that slots other than **body** will survive the routing process. On a related note, don't rely on the **category** slot of **fields** in your SetupRoutingSlip method either.

Don't rely on the closing order of views in the viewQuitScript. If you need to do some ordered cleanup, you can initiate your own message (for example, myViewQuitScript) from the view that first receives the viewQuitScript.

Replace system functions and messages at your peril. It's possible they will support other data types in the future (for example, to take NIL now where before they only took a string).

Don't assume anything about the built-in applications. Don't assume that they exist, or that their soups are there, or that the view structure will stay the same. If you do need to use a system feature (for example, a particular prototype, global function, or root method), test your assumptions.

```
local cardFileExists := GetRoot().cardfile;

if cardFileExists then
begin
   local cardFileSoup := GetUnionSoup(ROM_cardfilesoupname);
   if cardFileSoup then
      ...
end;
// :-0
if GetRoot().keyboardChicken then
begin
   ...
end;
```

Current Newtons have two levels of Undo; this may change. There could be more or fewer levels and it could change to Undo/Redo. It's safest to call AddUndoAction from inside your undo action; this will support Undo/Redo if we implement it, but will do nothing if we do not.

Thanks to our Newton Partners for the questions used in this column, and to jXopher Bell, Henry Cate, Bob Ebert, David Fedor, Stephen Harris, Jim Schram, Maurice Sharp, James Speir, and Bruce Thompson for the answers.•

Have more questions? Take a look at Newton Developer Info on AppleLink.•

Want to Show off your cool code?

YOUR NAME HERE

Do you have code that solves a problem other Macintosh developers might be having? Why not show it off by writing about it in *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., 1 Infinite Loop, Cupertino, CA 95014.

KON & BAL'S PUZZLE PAGE

Zoning Out

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL). 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. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.

KONSTANTIN OTHMER AND BRUCE LEAK

BAL I've got a small problem I'd like you to help me with.

KON Who's paying the airfare this time?

BAL Nothing like that. It's really quite straightforward, and surprisingly reproducible. The problem is that sometimes when I'm using Microsoft Word 5.1a and I pull down a menu, when I let go of the menu there's garbage on the screen where the menu was.

KON That was a problem they were having in the beta release, but I think it's fixed in the final version of Windows 95.

BAL Actually, this is on a Power Macintosh 6100, and I haven't yet installed Windows 95 on top of my SoftPC, which runs on my 68000, which is being emulated by Gary's emulator.

KON Microsoft is still in the loop.

BAL Well, it's not just a Microsoft problem. I can't seem to make it happen with Word by itself. It only seems to happen if I run and quit cc:Mail before running Word.

KON That darn Justice Department! Without them you could just be running Microsoft mail, and you probably wouldn't have this problem.

Try running Word; then launch and quit cc:Mail. Does it still happen?

100 BAL Now Word is working fine. In fact, Word works in every case — at least as far as this problem is concerned — unless I launch and quit cc:Mail before launching and quitting Word. And the interesting thing is that it only happens with the Modern Memory Manager on.

KONSTANTIN OTHMER AND BRUCE LEAK

KON has been holding a steady job at Catapult Entertainment for many months now, but he spends more time playing soccer than working. BAL is at the front of the self-employment line and has finally moved out of his hotel and into a house. Rumor has it that behind the house there's a big archery field.•

KON Just run your machine with the classic Memory Manager. I have problems running THINK C's debugger when I use the Thread Manager and the Modern Memory Manager. There's just too many of these kinds of bugs to deal with!

BAL Not so fast, QuickDraw. The Modern Memory Manager gives you lots of great new features. First of all, your machine will run faster. In addition to being ported native, it also uses much more efficient algorithms. It keeps track of free blocks in a separate list, keeps track of heap zones to make RecoverHandle work better, and has a back pointer so that blocks can be walked either way, drastically decreasing heap-walking time and making things much more efficient — especially when virtual memory is on. Also, the Modern Memory Manager was designed to be bus error proof, in that it returns from any internally generated exception by returning an error to its caller (though this was changed in the latest version of the Modern Memory Manager, as you may have read in the Balance of Power column in *develop* Issue 23). Finally, in the old Memory Manager moving the partition between the system and Process Manager heaps was a total nightmare; this problem was solved in the Modern Memory Manager.

KON Anytime you port something native you have two choices: rewrite the code directly, preserving internal algorithms and data structures, or rethink and reimplement, preserving only the top-level application interface. The first choice virtually guarantees compatibility but makes it difficult to maintain in the future, while the second gives you slightly less compatibility but a much better upgrade path, better maintainability, and a much more efficient system. It sounds like they went with the second choice, but at the obvious expense of some short-term compatibility problems. And it seems like that's what we're dealing with here.

BAL Thanks for the philosophy lesson. Are you going to solve the problem?

KON OK. Launch and quit cc:Mail and check all the heaps. Look for orphaned memory, locked blocks being left around, or any other signs of an application not properly cleaning up after itself.

BAL I need to install MacsBug to do that. I'll install version 6.5d11 because it has some new PowerPC features in case we need them.

KON I'm afraid we will.

90 BAL So after we quit cc:Mail, the system heap grew some, but all the heaps seem fine. We have an extra 128-byte pointer, and we have five extra handles for a total of almost 32K, but three of those (25K) are purgeable.

KON All this extra stuff lying around certainly explains why I have to reboot every couple of hours.

BAL Yeah, and those OS engineers really worked on that problem. On System 7.5 you get a pretty picture and a nice thermometer bar!

KON So try the patch dcmd. It will tell you what traps have been patched. Before you run cc:Mail, type

```
patch s
```

to grab a snapshot of all the traps. When you're in cc:Mail, just type

```
patch
```

and you'll get a list of all the traps that have been patched. It's a great way to find random skankiness.

BAL The only OS trap that they patch is _Rename, and they patch the Toolbox traps _Pack8, _UserDelay, _SysErr, _LoadSeg, _UnloadSeg, and _ExitToShell.

KON OK, and what's still patched after the application quits?

BAL Nothing. It seems to totally clean up.

KON Wonderful. What does Word patch?

80 BAL The OS traps _Rename and _CompactMem, and the Toolbox traps _Pack8, _UserDelay, _HiliteWindow, _FrontWindow, _SysError, _LoadSeg, and _ExitToShell.

KON There seems to be a lot of overlap. We should check a do-nothing generic application. I bet the system is magically patching some stuff when it runs an application.

70 BAL It turns out that all those traps except _HiliteWindow, _FrontWindow, _CompactMem, and _UnloadSeg are always getting patched.

KON It figures. Word is augmenting parts of the Memory Manager and getting in on some Window Manager action, and cc:Mail is playing games with the Segment Loader. Where's that book on Macintosh programming guidelines?

65 BAL I don't think they read that in Redmond. By the way, even though menu code is fairly boilerplate, this one's a mixed bag. Netscape, SimpleText, and FindFile work fine, but Word and THINK Reference fail consistently.

KON Boy, times have changed. I remember when you used to just dive right into MacsBug, disassemble a bunch of code, and get to the bottom of these problems. Now you're looking at what SimpleText does compared to Word!

BAL I'm not the one who's doing it. I don't even touch the computer anymore. It's one of my henchmen, Paul Young.

KON Anyway, there are two ways the bits behind the menus get redrawn. If plenty of memory is available, they get back-buffered and restored with CopyBits. If there's not much memory, an update event is generated.

BAL Since Word is the only application running at the time, we have plenty of memory.

KON Set a breakpoint on CopyBits and pull a menu down. The first break will be when the bits are being saved. Let's look at the address, step over the call, and make sure the right data was put there. When you let the menu up, you'll break on CopyBits again. Is the source data correct — that is, is the source our previous destination?

BAL The base address when the bits are restored isn't the same as the base address when they get saved.

KON Where is the base address? Is it part of a handle that moved?

60 BAL The base address for the restore is $40810000.

KON Someone is dereferencing zero! It sounds like the bits are getting saved in a handle, and somehow the handle is getting trashed. Let's follow the handle from the save and see what happens to it.

55 BAL When the bits are being saved, the base address is part of a handle in MultiFinder temporary memory. The handle is $438 bytes long.

KON What happens to that memory on the restore?

50 BAL The memory still exists, and the data is fine. It's just that the PixMap doesn't point there anymore.

KON So we need to figure out where the Menu Manager is storing the PixMap and why that location is getting trashed.

BAL The Menu Manager uses SaveBits and RestoreBits, which allocate memory for the pixels using the offscreen buffer calls that return PixMaps. The PixMap base address does double duty: when it's unlocked it's a handle; when it's locked it's a pointer. There's a flag in rowBytes to indicate what state it's in. To go from the locked state to the unlocked state, the GWorld routines call RecoverHandle.

KON Let's break on RecoverHandle and see what we get back.

45 BAL It returns 0. But why?

KON It's kind of weird that this happens only with the Modern Memory Manager. In the old Memory Manager, you had to set the heap zone before calling RecoverHandle. The Modern Memory Manager relaxed this restriction and keeps a tree of valid heaps. When you call RecoverHandle, it walks the heap tree. If cc:Mail is somehow corrupting the tree, RecoverHandle will fail.

BAL Nice theory. How are you going to test that?

KON E.T.O. 17 has a debugging version of the native Memory Manager that will print out diagnostics anytime weird stuff happens. Let's install it and reboot.

40 BAL When you boot, you drop into MacsBug with the message "Bad pointer being passed to RecoverHandle 00030020." It looks like "PC Exchange" was loading.

KON Let's try booting with the extensions off. Use the Extensions Manager so that you can keep MacsBug, the Memory control panel (so that we're sure we're in the Modern Memory Manager), and the Debugging Memory Manager.

35 BAL When I run the Extensions Manager, I break into MacsBug with the message "Bad handle; are you unlocking a fake handle?"

KON A complete treatise on all the memory crimes committed in the Macintosh is beyond the scope of this column.

BAL Without superfluous extensions, the problem at boot time goes away, but we still have the problem in Word.

KON Well, let's look at the zones and see if everything looks OK. Let's do an **hz** to list all the heap zones.

BAL OK. But **hz** doesn't use the heap tree, so if you want to check the heap tree you'll have to do it manually.

KON Great. I'll use the SmartFriends debugging trick and call Jeff to figure out how to do that.

Jeff The heap tree is part of the zone header. The system zone starts at $2800, and a pointer to the next zone starts at offset $20. $2820 contains $1672DF0.

KON That should be the Process Manager zone. But that number is really big. How could that be? How many fonts do you have installed?!

Jeff Since the system heap can grow, we put the Process Manager zone header at the end of the block, so we don't have to move the header every time the heap size changes.

30 BAL The next zone in the Process Manager is nil, since at the top level there are only two zones: the system zone and the Process Manager zone.

KON Let's look at the child zones inside the Process Manager.

Jeff The child zones are pointed to by offset $24 in the zone header.

25 BAL The first child zone is the Word zone, which corresponds to what we got from **hz**. And the Word zone header has no child zones.

KON So the world makes sense so far. Does the next zone pointer make sense?

BAL It's kind of wacky. It points inside the Word heap!

KON That's a problem. Does that zone header look reasonable, at least?

20 BAL No. It's trash. It looks like Word code.

KON What happens if you don't run cc:Mail before running Word? And how does the Memory Manager know how to update the zone headers? There's no call to explicitly destroy zones, only create them.

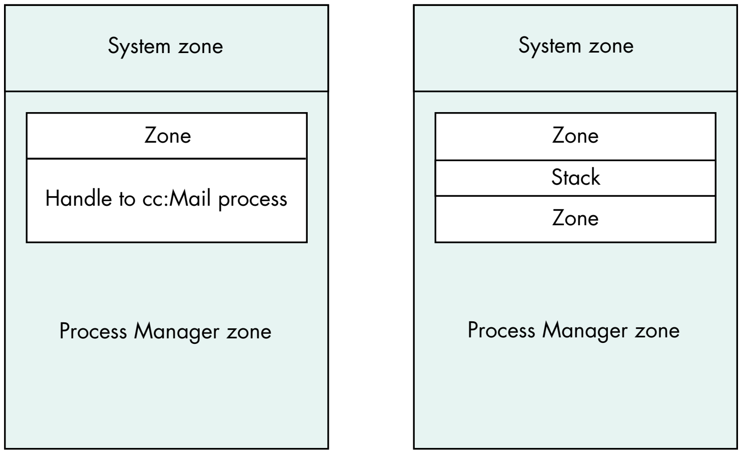
BAL I'll take the second question first. Zones are created by InitZone, and they're never explicitly destroyed. In the Modern Memory Manager, there's new logic in DisposeHandle that checks to see if the handle is a zone; if so, it assumes the zone is destroyed and updates the heap tree.

KON Will the skankiness ever end?

15 BAL If I run Word without first running cc:Mail, the heap tree is OK.

KON Now we just need to figure out why the heap tree is getting trashed. Even though the tree update algorithm is implicit, it seems pretty good at first blush. Let's go through the failing scenario and compare the heap zones to the tree and figure out when they diverge.

10 BAL When we run cc:Mail, **hz** doesn't agree with the zone structure we get by walking the heap tree. Here's what the two structures look like:



KON So the cc:Mail zone is smaller than the handle of the memory it's in. Someone limited the size of the application zone. In the heap tree view, it's clear why: another zone is being allocated; 32K is left between the zones, and that space is being used for the stack.

5 BAL The reason **hz** can't find the second zone is that before the Modern Memory Manager, no one kept explicit track of the zones. Basically, the **hz** command has to search for the zones. It does this by starting from the system zone, which is always pointed to by low memory (and is usually located after the trap tables at $2800). From the system heap zone header, it can find the zone trailer. Right after that block is the Process Manager zone header. It walks all the blocks in a zone and finds all the handles that look like other zones. It starts by assuming that the handle contains a zone, and then checks to see if the zone header points to a block that looks like a trailer and if the trailer points back to the zone header. When it looks for zones inside other zones, it assumes that they begin either at the start of the handle or right after another zone. Since cc:Mail has its stack space between the two zones, the **hz** command can't find it.

KON OK. Unfortunately we're not debugging the **hz** command. But that probably gives us a clue as to why the Modern Memory Manager is getting confused. It seems to keep pretty good track of the zones that are getting created, since that's easy by just watching InitZone. But it gets confused when the zones are being disposed of, since it does that by watching DisposeHandle.

BAL Exactly. The heap tree gets trashed when cc:Mail quits, since the Modern Memory Manager assumes that there's only one zone (and perhaps its children) in any handle. So when it sees the dispose, it throws away the first zone and all its children, but it doesn't throw away the second zone. It works fine with the old Memory Manager since no one ever explicitly keeps track of all the zones. But the Modern Memory Manager uses the heap tree for RecoverHandle, and the tree is trashed, so either the machine crashes or you get garbage.

KON That's pretty interesting. In this case, neither cc:Mail nor Word did anything wrong. The way cc:Mail used the Memory Manager was nonstandard, and when the algorithms in the Modern Memory Manager changed, there were some interesting cases that fell through the cracks. I think the newer version of cc:Mail no longer allocates zones this way. And the Memory Manager will undoubtedly soon be smarter.

BAL Nasty.

KON Yeah.

SCORING

70–100	In the end zone
50–65	Middle ground, the Twilight Zone
25–45	Out there in the ozone
5–20	Low memory, zoned out•

Thanks to Jeff Crawford and Bill Knott for reviewing this column. Special thanks to Rocket Scientist Paul Young, who originally found this puzzler and had the tenacity to narrow it down to a reproducible case.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Managing Editor *Toni Moccia*

Technical Buckstopper *Dave Johnson*

Bookmark CD Leader *Alex Dosher*

Able Assistant *Meredith Best*

Our Boss *Mark Bloomquist*

His Boss *Dennis Matthews*

Review Board *Jim Luther, Dave Radcliffe, Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*

Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Tim Monroe, Cheryl Potter, Joan Stigliani*

Indexer *Marc Savage*

ART & PRODUCTION

Production Manager *Diane Wilcox*

Technical Illustration *Mary Prusmack Ching, Deb Dennis, John Ryan, Laurie Wigham*

Formatting *Forbes Mill Press*

Photography *Sharon Beals, Maggie Fishell, Gretchen Linton*

Cover Illustration *Graham Metcalfe of Metcalfe/Shubert Design*

ISSN #1047-0735. © 1995 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, APDA, AppleLink, ColorSync, HyperCard, LaserWriter, Mac, MacApp, Macintosh, MacTCP, MPW, Newton, Power Macintosh, QuickTime, SANE, and TrueType are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, AppleScript, A/ROSE, Balloon Help, develop, DocViewer, Dylan, Finder, MessagePad, NewtonMail, NewtonScript, OpenDoc, Power Mac, PowerTalk, and QuickDraw are trademarks of Apple Computer, Inc. Adobe, Acrobat, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. UNIX is a registered trademark of Novell, Inc. in the United States and other countries, licensed exclusively through the X/Open Company, Ltd. NuBus is a trademark of Texas Instruments. All other trademarks are the property of their respective owners.

Printed on recycled paper

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. This CD contains a subset of the materials on the monthly *Developer CD Series*, 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 code is updated periodically, so always use the most recent CD.) The CD also contains Technical Notes, sample code, and other documentation and tools (these contents are subject to change). Items 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 in the Developer Services areas on AppleLink and eWorld and at ftp.info.apple.com. (Selected articles are on the World Wide Web at http://www.apple.com, in the Developer Services area.)

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. Append "@eworld.com" to eWorld addresses, and append "@online.apple.com" to NewtonMail addresses.

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., 1 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 and back issues. You can subscribe to *develop* through APDA (see ordering information below) or use the subscription card in this issue. You can also order printed back issues from APDA. For all subscription changes or queries, contact APDA and *be sure to include your name, address, and account number as it appears on your mailing label.*

The one-year U.S. subscription price is $30 (for 4 issues and 4 *develop Bookmark* CDs), or $50 U.S. in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

APDA. To order products from APDA or receive the *Apple Developer Tools Catalog* of all the products available from APDA, 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 23 September 1995

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

On this issue's CD, all the files that used to be in Apple DocViewer format have been converted to Adobe™ Acrobat™. Based on feedback that we've gotten from many of you since we started using DocViewer, we trust you'll be happy with this change. You should find that Acrobat has better search features and resolves some other problems. Because conversion is faster, the information can be more timely. Also, the files take up less space. So we hope you're satisfied — but you probably won't be for very long. It's just not the nature of the computer-using beast.

Think about it: How long after you get an upgrade to some software or hardware product do you start looking ahead to the next version? With the old problems solved and your old needs satisfied, you go on to realize a set of new ones. When it comes to computers, we always want more, and better.

I remember when the Macintosh was first designed, Steve Jobs kept saying it was to be an appliance, like a toaster: you simply plug it in and it does what you want, reliably and without fuss. (We're not talking multi-attachment Cuisinart here.) As you no doubt recall only too well, the options to add to the functionality of the first Macintosh were intentionally limited. It was to be a simple "black box" ("beige box"?).

Now, I know you're glad that that era didn't last very long, but think for a moment about all those non-nerds out there who haven't yet seen fit to buy a computer — all those potential customers Jobs was hoping to attract. They write things and add up numbers sometimes just like we do, don't they? So what are they waiting for?

I think the problem is that they want toasters — machines that work year after year without always needing to be updated, upgraded, or extended. They see the computer as a moving target, constantly advancing to satisfy some relatively insignificant new needs they never even knew they had — doomed to instant obsolescence. Better they should use a pen or pencil.

I suffer from this attitude myself to some degree, at least when I'm wearing my Home User hat. There I use a little old Macintosh with old but reliable software that works every time I do the paperwork on it that I've been doing for ten years now. But at Apple, I become a Computer Professional monster with ravenous needs for the latest and greatest software and hardware — downright insatiable.

So enjoy it while you can: have your fill of our new Acrobat files or whatever innovation pleases you these days. But rest assured that you'll be hungry again in no time.

CRose

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) resisted learning anything about computers in college, where she majored in math, but she couldn't escape them in the real world. First she used gigantic IBM machines at her statistical research job in Manhattan. But California beckoned, and with it came terminals spewing yellow paper and, eventually, computers with display screens. This was a big "Wow!" at first, but now Caroline gets her excitement from non-computer endeavors, such as travel. On her trip this year to the Big Island of Hawaii, she kayaked among (breaching) humpback whales and watched lava flow down cliffs and into the Pacific. Hard to top that!•

LETTERS

FLOATING WINDOWS AGAIN

I'd like to use the library of functions for floating windows described in *develop* Issue 15 by Dean Yu (updated on Issue 21's CD). I'm using CodeWarrior 5.5, and when I try to compile the sample project (or any other project that includes the WindowExtensions.h file) I get a "WindowRef redeclared" error. There seems to be a conflict with the universal headers.

Before I try to get rid of this error myself (and probably make everything wrong), I thought I'd ask if you could suggest a simple and clean solution.

— Fred Klein

On this issue's CD is a new version of the floating windows library that fixes this problem, and others. The problem was that Apple finally "caught up" with Dean and defined things in the universal headers that he had defined, in his forward-looking way, back when he first wrote the article.

Also on the CD you'll find an even newer version of the library that compiles with STRICT_WINDOWS defined. This necessitated a complete rewrite of some portions of the code, so consider it risky. Please try it and send me any bugs you find!

— Dave Johnson

POWERPC ASSEMBLY NITS

Great article on PowerPC™ assembly language in *develop* Issue 21! It was clear, and I learned a lot reading it. But I have two nitpicks. On page 27 you show glue code for a cross-TOC call. The second instruction should be

```
stw    RTOC,20(SP)
```

And the third instruction has a typo in it. It should be

```
lwz    r0,0(r12)
```

— David Shayer

Thanks for catching these. The interesting thing is that the second instruction appears that (wrong) way in the PPCAsm manual. Whoops!

— Dave Evans

UP ON THE DOWNSIDE

I just wanted to tell you that I really liked the Veteran Neophyte column in Issue 21, about the downside of programming. It struck a nerve with me. The thing that goes through my mind whenever I sit down to write some code is "There has to be a better way!" Alas, by the time there is a better way, I will probably have moved on to some other profession.

— Jamie Osborne

Your Veteran Neophyte column on the pains of programming really struck a nerve (and not just because I have carpal tunnel syndrome). I often spend a while putting things on paper, only to abandon the project once I become convinced that I've figured out the solution and its implementation would just be hours and hours of typing. Sort of meta-programming.

— Tom Busey

I just finished reading the Veteran Neophyte columns in Issue 17 and Issue 21, "Why We Do It" and "The Downside." They were given to me by

KEEP US ON OUR TOES!

We welcome your nitpicking letters to the editors, especially regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK. Or you can write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014. 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).•

a friend who is an avid programmer. The type of things you described sounded *just* like my friend; I think he showed the columns to me to explain why every time I see him he's sitting in front of the computer, and why he stays up till all hours of the morning working on programs that end up frustrating him.

I thought I should let you know that your columns were appreciated not only by those who program, but by those who are close to programmers and wonder sometimes what unseen force has gotten hold of them and sucked them into their work.

— Greta Meussling

The "Downside" column seems to have hit home with many people; I got a lot *of comments about it. It's nice to be assured that I'm not the only reluctant programmer in the world, and that I'm not the only one who thinks there ought to be a better way.*

— Dave Johnson

ACROBAT: PRETTY DARN FINE

This probably isn't the first time you've heard this, but how about offering *develop* in Acrobat (PDF format) as well? For me, Acrobat is more convenient than Apple DocViewer as an application and, most important, its files are a lot smaller. I routinely convert *develop* to PDF and then add PDF hyperlinks and bookmarks. For one issue I converted, for instance, the DocViewer version (without the index) is 2.9 meg, while the PDF version is only 770K. It's even smaller than the StuffIt version of the DocViewer document (1.2 meg). And the onscreen appearance is identical.

I still like the HTML versions for their immediacy, but for true WYSIWYG, low conversion effort, and small file size, you can't beat PDF.

— Shannon Spires

We agree with you. You'll notice that on this issue's CD, every issue of develop *has been converted to Acrobat — along with all the other files on the CD that used to be in Apple DocViewer format. Enjoy!*

— Caroline Rose

UP ON THE WEB

Thanks for making both *develop* and *Apple Directions* available on the World Wide Web. We're on a very tight budget and can't afford a subscription at this time. The online versions allow us to access the information and still come out with a product on budget.

— Mattias Fornander

I'm a student who reads *develop* online via the Internet through UCLA's (UNIX®) workstations. Your putting *develop* on the World Wide Web is great! Even though the comfort of reading (and printing) *develop* online will never equal the ease of the regular version, I don't have to fight with ftp and MS-DOS floppy disks to read your magazine. So please continue to publish *develop* in HTML.

IMHO, your magazine is a service to the Mac developer community, and you would help Apple by letting every possible programmer access it without hassle. Thanks for this effort.

— Eric Gouriou

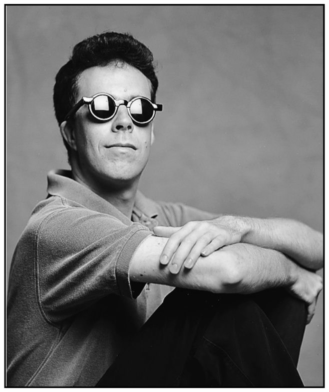
We've got articles from some issues of develop *on the World Wide Web now (at http://www.apple.com, in the Developer Services area) and are working on putting more up there. This kind of feedback helps make it happen — so thanks for writing.*

Readers of the online version: Don't confuse printed develop *with the monthly Apple Developer Mailing; a subscription to the monthly mailing (which includes a CD that has* develop *on it) is rather costly, but it costs only $30 for four quarterly printed issues of* develop *(with Bookmark CD). See the inside front cover of this issue for ordering information. (Sorry, I couldn't resist this opportunity for a plug!)*

— Caroline Rose

Music the Easy Way: The QuickTime Music Architecture

Music has become cheap and plentiful on the Macintosh, and many applications are now making "casual" use of music. With the QuickTime Music Architecture, or QTMA, including music in your application has never been simpler. Its API is straightforward and easy to use, and you don't need intimate knowledge of MIDI protocols or channel and voice numberings. Nor do you need an external MIDI device; QTMA can play music directly out of the Macintosh's built-in speakers. And QTMA is widely available — it's on every Macintosh that has QuickTime 2.0 (or later) installed.



DAVID VAN BRINK

The QuickTime Music Architecture is perfect for adding a little bit of music to your application. It has a set of well-supported high-level calls for playing musical notes and sequences, it deals with MIDI protocols so that your application doesn't have to, and it handles timing for entire tunes. With QTMA, you can specify musical instruments independent of device, and play music either directly out of built-in speakers or through a MIDI synthesizer.

QTMA first became available with QuickTime 2.0 and offers some new features in QuickTime 2.1, which should be available through APDA by the time you read this. The code in this article is written for version 2.1; minor changes will be required for 2.0. (Before making use of the QuickTime 2.1 features, your code should call Gestalt with the gestaltQuickTimeVersion selector and check the version number returned.)

This article shows how your application can use QTMA to play individual notes, sequences of notes composed on the fly, or prescored sequences, and how to read input from external MIDI devices. This issue's CD contains all the sample code and a THINK C project to build and run it. We'll start with a look at QTMA in relation to other ways of supporting music on the Macintosh; then we'll get down to business and play some music with QTMA.

QTMA IN CONTEXT — A LOOK AT MUSIC AND MIDI SUPPORT ON THE MACINTOSH

Support for MIDI and musical applications on the Macintosh platform has a somewhat checkered history. Developers have been faced with such options as writing their own serial drivers, using the MIDI Manager, or using third-party operating system extensions such as the Open Music System (OMS, formerly Opcode MIDI

DAVID VAN BRINK lives in a tiny experimental habitat overlooking the Denny's parking lot in Santa Cruz, California. He experiences life at 14,400 bits per second. See http://www.srm.com for more information.•

System) and the Free MIDI System (FMS) from Mark of the Unicorn. None of these are practical for adding just a little music to your application.

Writing a serial driver to send MIDI output to an Apple MIDI adapter or to any third-party MIDI adapter isn't that complicated if you enjoy writing low-level code to access hardware registers on the SCC serial chip. I say this in all seriousness: that kind of code really is fun to write! But it's not the best way to do things, because changes in the OS and hardware can render your work useless. And writing the low-level serial code for MIDI input has additional complexities, primarily because of the interrupt timings in many parts of the Mac OS.

The MIDI Manager is a slightly better tool to use for MIDI input and output. Unfortunately, Apple's support for this product has been less than consistent, and the MIDI Manager itself has some inherent performance limitations, though these are less critical on faster hardware (68030 processor or better).

Both OMS and FMS are quite appropriate for professional music scoring and editing products. Among the facilities that these extensions provide is a "studio configuration"; this lets the user describe to the system the various MIDI devices attached to the computer so that different applications can access them.

All of these options have drawbacks for making casual use of music: you have to access an external MIDI device, which most users don't have, and you have to use MIDI protocols to talk to that device. QTMA frees you from both of these constraints. It also frees you from needing to know a lot about MIDI itself; if you want to know anyway, check out the information in "A MIDI Primer."

QTMA'S BASIC COMPONENTS

QTMA is implemented in three easy pieces, as QuickTime components for playing individual notes, playing tunes (sequences of notes), and driving MIDI devices.

- The *note allocator component* is used to play individual notes. The calling application can specify which musical instrument sound to use and exactly which musical synthesizer to play the notes on. The note allocator component also includes a utility that allows the user to pick the instrument.
- The *tune player component* can accept entire sequences of musical notes and play them from start to finish, asynchronously, with no further need for application intervention. This is handy if you'd like to play some infernally irritating little melody, or perhaps threnody, during each game level of Boom Three Dee or whenever.
- Individual *music components* act as device drivers for each type of synthesizer attached to a particular computer. Two music components are provided with QuickTime 2.0: the software synthesizer component, to play music out of the built-in speaker, and the General MIDI component, to play music on a General MIDI device attached to a serial port. QuickTime 2.1 supports a small number of other popular synthesizers as well.

PLAYING NOTES WITH THE NOTE ALLOCATOR

Playing a few notes with the note allocator component is simple. To play notes that have a piano-like sound, you need to open up the note allocator component, allocate a *note channel* with a request for piano, and play. That's it! If you're feeling like a particularly well-behaved software engineer, you might dispose of the note channel and close the note allocator component when you're done. We'll get to the code in a moment; first we'll look at some important related data structures.

A MIDI PRIMER

MIDI, or Musical Instrument Digital Interface, uses a serial protocol and a standard 5-pin connector that you'll find on professional electronic music gear made after 1985 or so. The connector's relatively large size, about half an inch in diameter, was chosen so that it could withstand the rigors of the road — in other words, so that even drummers could plug it in.

Because MIDI cables can carry signals in only one direction, synthesizers have separate connectors for MIDI input and MIDI output. (This differs from modem cables, which carry signals in both directions.)

MIDI is a serial protocol running at 31250 baud, 8 data bits, 1 stop bit, no parity. The command structure for a MIDI stream is simple: each byte is either a *status byte* or a *data byte*.

A status byte establishes a mode for interpreting the data bytes that follow it. The high bit is set, and the next three bits indicate the type of status byte. The low four bits are typically used to specify a *MIDI channel*. Thus MIDI can address up to 16 unique channels, each of which may play a different musical instrument sound. Later extensions to MIDI let you address more channels through the use of escape codes and bank switching.

The most common status message is the Play Note message, which has a value of 0x90 plus the MIDI channel number. Each note is defined by a pitch and velocity. The pitch is an integer from 0 to 127, where 60 is musical middle C (61 is C sharp, 59 is B, 72 is the C above middle C, and so on). The velocity is an integer from 0 to 127 that describes how loud to play the note; 64 is average loudness, 127 is very loud, 1 is nearly inaudible, and 0 means to stop playing the note.

So, to play a C-major chord on MIDI channel 0, you send the seven bytes 0x90 0x3C 0x40 0x40 0x40 0x43 0x40 to begin the sound. After a suitable interval, you send 0x90 0x3C 0x00 0x40 0x00 0x43 0x00 to silence it.

All of this is exactly the sort of stuff you don't need to know if you use the QuickTime Music Architecture for your music-playing needs. But you just can't know too many useless facts, right?

NOTE-RELATED DATA STRUCTURES

A note channel is analogous to a sound channel in that you allocate it, issue commands to it to produce sound, and close it when you're done. To specify details about the note channel, you use a data structure called a NoteRequest (see Listing 1). The NoteRequestInfo structure in the NoteRequest is new in QuickTime 2.1; it simply encapsulates the first few fields of the old NoteRequest structure and splits the first of those fields into two, **flags** and **reserved** (which are decribed in the documentation accompanying the QuickTime 2.1 release).

The next two fields specify the probable *polyphony* that the note channel will be used for. Polyphony means, literally, many sounds. A polyphony of 5 means that five notes can be playing simultaneously. The polyphony field enables QTMA to make sure that the allocated note channel can play all the notes you'll need. The typicalPolyphony field is a fixed-point number that should be set to the average number of voices the note channel will play; it may be whole or fractional. Some music components use this field to adjust the mixing level for a good volume.

The ToneDescription structure is used throughout QTMA to specify a musical instrument sound in a device-independent fashion. This structure's synthesizerType and synthesizerName fields can request a particular synthesizer to play notes on. Usually, they're set to 0, meaning "choose the best General MIDI synthesizer." The gmNumber field indicates the General MIDI (GM) instrument or drum kit sound, which may be any of 135 such sounds that are supported by many synthesizer manufacturers. (All these sounds are available on a General MIDI Sound Module.) The GM instruments are numbered 1 through 128, and the seven drum kits are numbered 16385 and higher. A complete list of instrument and drum kit numbers is provided in Table 1. For synthesizers that accept sounds outside the GM library, you

Listing 1. Note-related data structures

```
struct NoteRequest {
   NoteRequestInfo   info;    // · in post-QuickTime 2.0 only
   ToneDescription   tone;
};

struct NoteRequestInfo {
   UInt8    flags;
   UInt8    reserved;
   short    polyphony;
   Fixed    typicalPolyphony;
};

struct ToneDescription {
   OSType   synthesizerType;
   Str31    synthesizerName;
   Str31    instrumentName;
   long     instrumentNumber;
   long     gmNumber;
};
```

can use the instrumentName and instrumentNumber fields to specify some other sound.

THE NOTE-PLAYING CODE

The routine in Listing 2 plays notes in a piano-like sound with the note allocator component. We start by calling OpenDefaultComponent to open up the component. If this routine returns 0, the component wasn't opened, most likely because QTMA wasn't present.

Next we fill in the NoteRequestInfo and ToneDescription structures, calling the note allocator's NAStuffToneDescription routine and passing it the GM instrument number for piano. This routine fills in the gmNumber field and also fills in the other ToneDescription fields with sensible values, such as the instrument's name in text form in the instrumentName field. (The routine can be useful for converting a GM instrument number to its text equivalent.)

After allocating the note channel with NANewNoteChannel, we call NAPlayNote to play each note. Notice the last two parameters to NAPlayNote:

```
ComponentResult NAPlayNote(NoteAllocator na, NoteChannel nc,
   long pitch, long velocity);
```

The value of the pitch parameter is an integer from 1 to 127, where 60 is middle C, 61 is C sharp, and 59 is C flat, or B. Similarly, 69 is concert A, and is played at a nominal audio frequency of 440 Hz. The velocity parameter's value is also an integer from 1 to 127, or 0. A velocity of 1 corresponds to just barely touching the musical keyboard, and 127 indicates that the key was struck as hard as possible. Different velocities produce tones of different volumes from the synthesizer. A velocity of 0 means the key was released; the note stops or fades out, as appropriate to the kind of sound being played. Here we stop the notes after delaying an appropriate amount of time with a call to the Delay routine.

Table 1. The General MIDI instruments and drum kits

Piano		**Bass**		**Reed**		**Synth Effect**	
• 1	Acoustic Grand Piano	33	Acoustic Fretless Bass	65	Soprano Sax	97	Ice Rain
2	Bright Acoustic Piano	• 34	Electric Bass Fingered	• 66	Alto Sax	98	Sound Tracks
3	Electric Grand Piano	35	Electric Bass Picked	67	Tenor Sax	99	Crystal
4	Honky-tonk Piano	36	Fretless Bass	68	Baritone Sax	100	Atmosphere
• 5	Rhodes Piano	• 37	Slap Bass 1	• 69	Oboe	101	Brightness
6	Chorused Piano	38	Slap Bass 2	70	English Horn	102	Goblins
• 7	Harpsichord	39	Synth Bass 1	71	Bassoon	103	Echoes
• 8	Clavinet	40	Synth Bass 2	• 72	Clarinet	104	Space
Chromatic Percussion		**Strings and Orchestra**		**Pipe**		**Ethnic**	
9	Celesta	• 41	Violin	73	Piccolo	• 105	Sitar
10	Glockenspiel	42	Viola	• 74	Flute	• 106	Banjo
11	Music Box	43	Cello	75	Recorder	107	Shamisen
• 12	Vibraphone	44	Contrabass	• 76	Pan Flute	108	Koto
• 13	Marimba	45	Tremolo Strings	77	Bottle Blow	109	Kalimba
14	Xylophone	46	Pizzicato Strings	78	Shakuhachi	110	Bagpipe
15	Tubular bells	47	Orchestral Harp	• 79	Whistle	111	Fiddle
16	Dulcimer	• 48	Timpani	80	Ocarina	112	Shanai
Organ		**Ensemble**		**Synth Lead**		**Percussive**	
• 17	Hammond Organ	• 49	Acoustic String Ensemble 1	81	Square Wave	113	Tinkle Bell
18	Percussive Organ	50	Acoustic String Ensemble 2	• 82	Saw Wave	• 114	Agogo
19	Rock Organ	51	SynthStrings 1	83	Calliope	• 115	Steel Drums
20	Church Organ	52	SynthStrings 2	84	Chiffer	116	Woodblock
• 21	Reed Organ	• 53	Aah Choir	85	Charang	117	Taiko Drum
22	Accordion	54	Ooh Choir	86	Solo Vox	• 118	Melodic Tom
23	Harmonica	55	Synth Vox	• 87	5th Saw Wave	119	Synth Drum
24	Tango Accordion	• 56	Orchestra Hit	88	Bass and Lead	• 120	Reverse Cymbal
Guitar		**Brass**		**Synth Pad**		**Sound Effects**	
• 25	Acoustic Nylon Guitar	• 57	Trumpet	• 89	Fantasy	121	Guitar Fret Noise
26	Acoustic Steel Guitar	58	Trombone	• 90	Warm	122	Breath Noise
27	Electric Jazz Guitar	59	Tuba	• 91	Polysynth	• 123	Seashore
28	Electric Clean Guitar	60	Muted Trumpet	92	Choir	• 124	Bird Tweet
29	Electric Muted Guitar	• 61	French Horn	93	Bowed	• 125	Telephone Ring
30	Overdriven Guitar	62	Brass Section	94	Metal	• 126	Helicopter
• 31	Distortion Guitar	63	Synth Brass 1	95	Halo	127	Applause
32	Guitar Harmonics	64	Synth Brass 2	96	Sweep	• 128	Gunshot

GM Drum Kits

• 16385	Standard Kit
• 16393	Room Kit (a memory-reduced version of the Standard Kit)
16401	Power Kit
16409	Electronic Kit
16410	Analog Kit
16425	Brush Kit
16433	Orchestra Kit

• Bullets indicate the instruments and drum kits that are available for playing on the built-in synthesizer.

Listing 2. Playing notes with the note allocator component

```
void PlaySomeNotes(void)
{
   NoteAllocator    na;
   NoteChannel      nc;
   NoteRequest      nr;
   ComponentResult  thisError;
   long             t, i;

   na = 0;
   nc = 0;

   // • Open up the note allocator.
   na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!na)
      goto goHome;

   // • Fill out a NoteRequest using NAStuffToneDescription to help, and
   // • allocate a NoteChannel.
   nr.info.flags = 0;
   nr.info.reserved = 0;
   nr.info.polyphony = 2;    // • simultaneous tones
   nr.info.typicalPolyphony = 0x00010000;   // • usually just one note
   thisError = NAStuffToneDescription(na, 1, &nr.tone);  // • 1 is piano
   thisError = NANewNoteChannel(na, &nr, &nc);
   if (thisError || !nc)
      goto goHome;

   // • If we've gotten this far, OK to play some musical notes. Lovely.
   NAPlayNote(na, nc, 60, 80);  // • middle C at velocity 80
   Delay(40, &t);               // • delay 2/3 of a second
   NAPlayNote(na, nc, 60, 0);   // • middle C at velocity 0: end note
   Delay(40, &t);               // • delay 2/3 of a second

   // • Obligatory do-loop of rising tones
   for (i = 60; i <= 84; i++) {
      NAPlayNote(na, nc, i, 80);   // • pitch i at velocity 80
      NAPlayNote(na, nc, i+7, 80); // • pitch i+7 (musical fifth) at
                                   // • velocity 80
      Delay(10, &t);               // • delay 1/6 of a second
      NAPlayNote(na, nc, i, 0);    // • pitch i at velocity 0: end note
      NAPlayNote(na, nc, i+7, 0);  // • pitch i+7 at velocity 0:
                                   // • end note
   }

goHome:
   if (nc)
      NADisposeNoteChannel(na, nc);
   if (na)
      CloseComponent(na);
}
```

ROGER SHEPARD’S MELODY

In Listing 2, if you replace the code in the section labeled “Obligatory do-loop of rising tones” with the following code, you’ll receive a secret treat.

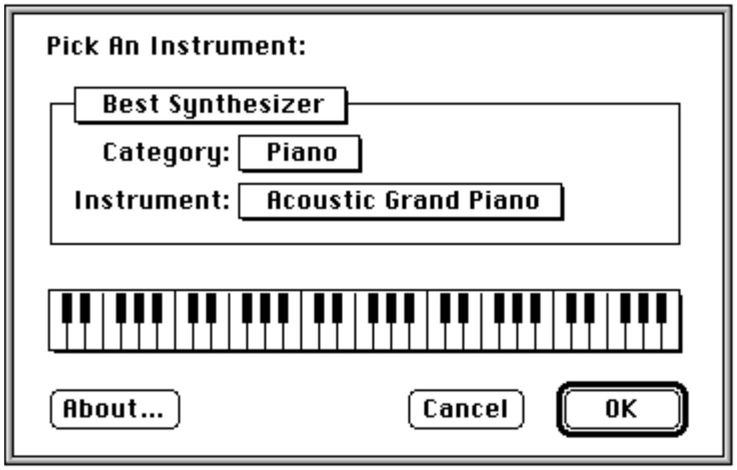
```
i = 0;
while (!Button()) {
   long  j, v;
   for (j = i % 13; j < 128; j+=13) {
      v = j < 64 ? j * 2 : (127 - j) * 2;
      NAPlayNote(na, nc, j, v);
   }
   Delay(13, &t);
   for (j = i % 13; j < 128; j+=13)
      NAPlayNote(na, nc, j, 0);
   i++;
}
```

This snappy little melody was discovered by psychologist Roger Shepard in the 1960s.

Finally, being well behaved, we dispose of the note channel and close the note allocator component.

LETTING THE USER PICK THE INSTRUMENT

Rather than specify the instrument sound itself, your application may want to let the user pick it. For this purpose, a nifty instrument picker utility is provided in the note allocator component. The instrument picker dialog, shown in Figure 1, enables users to choose musical instruments from the available synthesizers and sounds.



Pick An Instrument:

Best Synthesizer

Category: Piano

Instrument: Acoustic Grand Piano

Figure 1. The instrument picker dialog

The routine in Listing 3 shows one way that your application can use the instrument picker. It’s nearly identical to the code in Listing 2, except that the NAPickInstrument routine is called right after the call to NAStuffToneDescription. As in Listing 1, NAStuffToneDescription fills out a ToneDescription record for a particular GM instrument number; NAPickInstrument then invokes the instrument picker dialog and alters the passed ToneDescription to whatever instrument the user selects.

Listing 3. Using the instrument picker

```
void PickThenPlaySomeNotes(void)
{
   ...   // • declarations and initialization

   // • Open up the note allocator.
   ...

   // • Fill out a NoteRequest using NAStuffToneDescription to help,
   // • call NAPickInstrument, and allocate a NoteChannel.
   nr.info.flags = 0;
   nr.info.reserved = 0;
   nr.info.polyphony = 2;    // • simultaneous tones
   nr.info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 1, &nr.tone);  // • 1 is piano
   thisError = NAPickInstrument(na, nil, "\pPick An Instrument:",
                    &nr.tone, 0, 0, nil, nil);
   if (thisError)
      goto goHome;
   thisError = NANewNoteChannel(na, &nr, &nc);
   if (thisError || !nc)
      goto goHome;

   // • Play some musical notes.
   ...

   // • Obligatory do-loop of rising tones
   ...

goHome:
   ...      // • Dispose of the NoteChannel and close the component.
}
```

ADDING EXPRESSIVENESS WITH CONTROLLERS

There’s much more to music than simply playing the right notes at the right times. Although your code can simulate only a scant fraction of the expressiveness of a skillfully played acoustic instrument, there are certain things the note allocator component lets you do that help make your computer-synthesized music sound more interesting.

As we’ve already seen, the NAPlayNote routine has parameters for specifying pitch and velocity, the latter determining the volume of the note; changes in these parameter values can affect the expressiveness of your music. You can also add expressiveness to whatever notes are being played by using QTMA’s *controllers*. A controller is a parameter that’s set independently of the notes being played, with a call to the NASetController routine:

```
ComponentResult NASetController(NoteAllocator na, NoteChannel nc,
   long controllerNumber, long controllerValue);
```

Two simple controllers are the *pitch bend controller* and the *volume controller*. The pitch bend controller alters the frequency of any notes being played. It's like the whammy-bar on an electric guitar, which tightens or loosens all the strings simultaneously. The volume controller affects the sound of all notes similarly to the way key velocity affects the sound of individual notes.

Let's look at some source code that uses the pitch bend controller (Listing 4). This routine plays a major-fifth interval for a half second, "bends" it up by three semitones, holds it a half second, and then bends it back down to its original pitch.

Listing 4. Using the pitch bend controller

```
void PlaySomeBentNotes(void)
{
   ...   // · declarations and initialization

   // · Open up the note allocator.
   ...

   // · Fill out a NoteRequest using NAStuffToneDescription to help, and
   // · allocate a NoteChannel.
   ...

   // · If we've gotten this far, OK to play some musical notes. Lovely.
   NAPlayNote(na, nc, 60, 80);  // · middle C at velocity 80
   NAPlayNote(na, nc, 67, 60);  // · G at velocity 60
   Delay(30, &t);

   // · Loop through differing pitch bendings.
   for (i = 0; i <= 0x0300; i+=10) {     // · bend 3 semitones
      NASetController(na, nc, kControllerPitchBend, i);
      Delay(1, &t);
   }
   Delay(30, &t);
   for (i = 0x0300; i >= 0; i-=10) {     // · bend back to normal
      NASetController(na, nc, kControllerPitchBend, i);
      Delay(1, &t);
   }
   Delay(30, &t);
   NAPlayNote(na, nc, 60, 0);   // · middle C off
   NAPlayNote(na, nc, 67, 0);   // · G off

goHome:
   ...     // · Dispose of the NoteChannel and close the component.
}
```

Most QuickTime controller values are 16-bit signed fixed-point numbers (where the lower eight bits are fractional) and have a range of 0 to 127, with a default value of 0. However, the pitch bend controller has a range of -127 to 127, and the volume controller has a default value of 127, or maximum volume.

The *pan controller* has a slightly different definition from the other controllers. "Pan" refers to the position of the sound in the stereo field. Most synthesizers have audio output for left and right; on such synthesizers, the pan value is interpreted as follows: The default pan position (usually centered) is specified by a value of 0 to the pan controller. To position the sound arbitrarily, values between 1 (0x0100) and 2 (0x0200) are used to range between left and right, respectively. For synthesizers with *n* outputs, values between 1 and *n* are used to pan between each adjacent pair of outputs. Note that the built-in synthesizer doesn't currently support panning.

BUILDING A TUNE

As mentioned earlier, an application can use the tune player component to play entire sequences of notes, or tunes. Applications often find it useful to play a tune that has been precomposed and stored in the application; other times, it may be useful to construct a tune at run time and then play it. In either case, the application must first build the tune. Here we'll take a look at the format of a tune and the routines and macros we use for building one.

THE FORMAT OF A TUNE

The format for tunes is a series of long words, subdivided into bitfields. Your application needs to build a *tune header* and *tune sequence* made up of different types of "events." The tune header contains one or more *note request events*, each a NoteRequest data structure with some encapsulating long words. The tune sequence is made up of *note events* that specify notes and durations, controller changes, and so on, as well as *rest events*; it's the musical score.

In the tune header, each note request event has the structure shown in Figure 2. (It's actually a *general event*, of the note request subtype.) Thus the first word is 0xF*nnn*0017, where *nnn* is the part number, and the last word is 0xC0010017. The part number is referred to later on by note events in the tune sequence. For example, given a header than contains a note request event specifying part 3, subsequent note events that specify part 3 will play in a note channel allocated according to that NoteRequest.

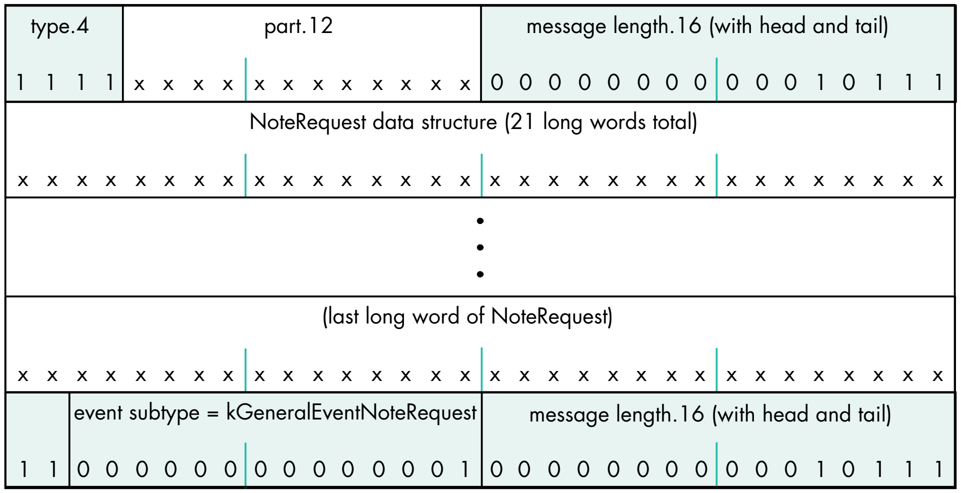
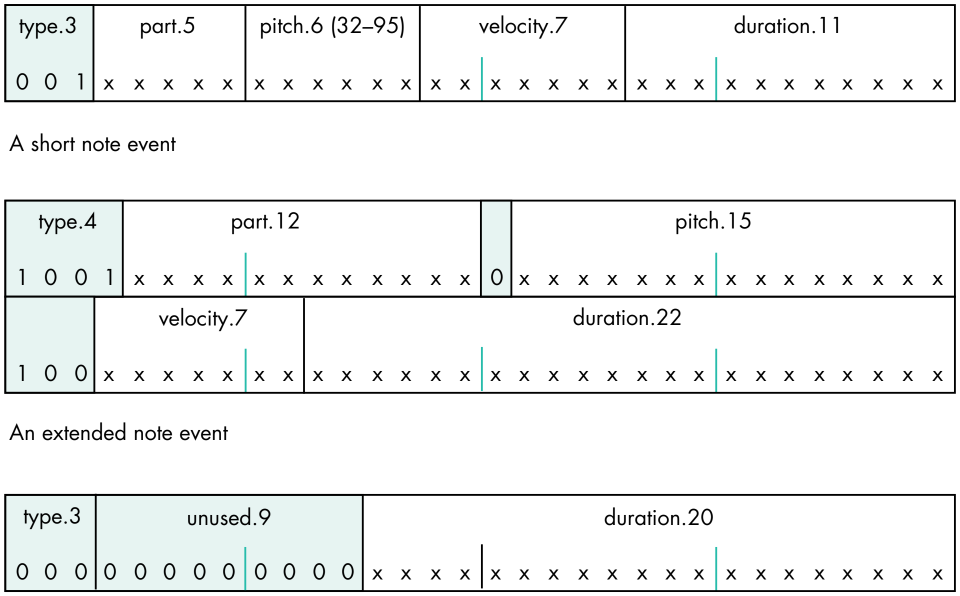

type.4
part.12
message length.16 (with head and tail)
1 1 1 1 x x x x x x x x x x x x 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1
NoteRequest data structure (21 long words total)
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
(last long word of NoteRequest)
x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
event subtype = kGeneralEventNoteRequest
message length.16 (with head and tail)
1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 1

Figure 2. A note request event

In the tune sequence, each note event includes the part, pitch, velocity, and duration of the note; a rest event specifies only a duration (see Figure 3). A note event can have either a short or an extended format. In a short note event, the pitch is limited to the range 32 to 95 (which covers most musical notes) and the part number must be less than 32. If either of these ranges is too small, or if you want to use a fixed-point pitch value or a very long duration, the extended note format may be used. Much of the time you can use the short format, to save space.


type.3
part.5
pitch.6 (32–95)
velocity.7
duration.11
0 0 1 x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
type.4
part.12
pitch.15
1 0 0 1 x x x x x x x x x x x x 0 x x x x x x x x x x x x x x x
velocity.7
duration.22
1 0 0 x x x x x x x x x x x x x x x x x x x x x x x x x x x x x
type.3
unused.9
duration.20
0 0 0 0 0 0 0 0 0 0 0 0 x x x x x x x x x x x x x x x x x x x x

A short note event

An extended note event

A rest event

Figure 3. Note and rest events

Both headers and sequences end with a *marker event* containing all zeroes (equivalent to 0x60000000), shown in Figure 4.

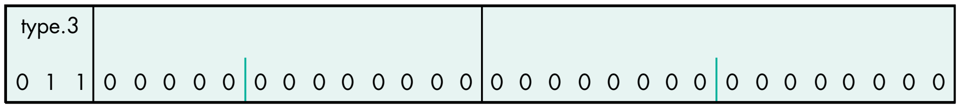


Figure 4. A marker event

THE TUNE-BUILDING CODE

Our sample code includes routines for building the tune header and tune sequence. These routines use some handy event-stuffing macros that are defined in the file QuickTimeComponents.h, and all have the form _StuffSomething(arguments). BuildTuneHeader (Listing 5) uses the following macro:

```
_StuffGeneralEvent(w1, w2, part, subtype, length);
```

The _StuffGeneralEvent macro fills in the head and tail long words of a particular type of general event — in our case, a note request event. Its arguments are, in order: the head and tail long words; the part number; the event subtype (kGeneralEventNoteRequest for a note request event); and the length in long words of the entire event, counting the head and tail. Note that the first two arguments are the head and tail themselves, not pointers — the macro expands to a direct assignment of these arguments.

BuildTuneSequence (Listing 6) uses the _StuffNoteEvent and _StuffRestEvent macros.

Listing 5. BuildTuneHeader

```
#define kNoteRequestHeaderEventLength \
               (sizeof(NoteRequest) /sizeof(long) + 2)    // • long words
#define our_header_length \
   ((2 * kNoteRequestHeaderEventLength + 1)) * sizeof(long)    // • bytes
unsigned long *BuildTuneHeader(void)
{
   unsigned long     *header, *w, *w2;
   NoteRequest       *nr;
   NoteAllocator     na;   // • just for the NAStuffToneDescription call
   ComponentResult   thisError;

   header = 0;
   na = 0;

   // • Open up the note allocator.
   na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!na)
      goto goHome;

   // • Allocate space for the tune header, rather inflexibly.
   header = (unsigned long *) NewPtrClear(our_header_length);
   if (!header)
      goto goHome;
   w = header;

   // • Stuff request for piano polyphony 4.
   w2 = w + kNoteRequestHeaderEventLength - 1; // • last long word of
                                               // • note request event
   _StuffGeneralEvent(*w, *w2, 1, kGeneralEventNoteRequest,
                          kNoteRequestHeaderEventLength);
   nr = (NoteRequest *)(w + 1);
   nr->info.flags = 0;
   nr->info.reserved = 0;
   nr->info.polyphony = 4;    // • simultaneous tones
   nr->info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 1, &nr->tone); // • 1 is piano
   w += kNoteRequestHeaderEventLength;

   // • Stuff request for violin polyphony 3.
   w2 = w + kNoteRequestHeaderEventLength - 1; // • last long word of
                                               // • note request event
   _StuffGeneralEvent(*w, *w2, 2, kGeneralEventNoteRequest,
                          kNoteRequestHeaderEventLength);
   nr = (NoteRequest *)(w + 1);
   nr->info.flags = 0;
   nr->info.reserved = 0;
   nr->info.polyphony = 3;    // • simultaneous tones
   nr->info.typicalPolyphony = 0x00010000;
   thisError = NAStuffToneDescription(na, 41, &nr->tone);  // • violin
   w += kNoteRequestHeaderEventLength;
   *w++ = 0x60000000;     // • end-of-sequence marker
```

(continued on next page)

Listing 5. BuildTuneHeader *(continued)*

```
goHome:
   if (na)
      CloseComponent(na);
   return header;
}
```

```
_StuffNoteEvent(w, part, pitch, volume, duration);
```

The _StuffNoteEvent macro fills in a note event. Its arguments are, in order: the long word to stuff; the part number; the pitch (where, as usual, 60 is middle C); the volume (velocity); and the duration (usually specified in 600ths of a second). The pitch must be between 32 and 95, and the part number must be less than 32. For values outside these ranges, a fixed-point pitch value, or a very long duration, use _StuffXNoteEvent.

```
_StuffXNoteEvent(w1, w2, part, pitch, volume, duration);
```

The _StuffXNoteEvent macro is for extended note events. It's identical to _StuffNoteEvent except that it provides larger ranges for pitch, part, and duration, and the event itself takes two long words.

```
_StuffRestEvent(w, restDuration);
```

The _StuffRestEvent macro fills in a rest event. It takes two arguments: the long word to stuff and the duration of the rest.

Listing 6. BuildTuneSequence

```
#define kNoteDuration 240    // · in 600ths of a second
#define kRestDuration 300    // · in 600ths -- tempo will be 120 bpm

#define our_sequence_length (22 * sizeof(long))      // · bytes
#define our_sequence_duration (9 * kRestDuration)    // · 600ths

unsigned long *BuildTuneSequence(void)
{
   unsigned long  *sequence, *w;

   // · Allocate space for the tune sequence, rather inflexibly.
   sequence = (unsigned long *) NewPtrClear(our_sequence_length);
   if (!sequence)
      goto goHome;
   w = sequence;
   _StuffNoteEvent(*w++, 1, 60, 100, kNoteDuration);    // · piano C
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 2, 60, 100, kNoteDuration);    // · violin C
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 1, 63, 100, kNoteDuration);    // · piano
   _StuffRestEvent(*w++, kRestDuration);
```

(continued on next page)

Listing 6. BuildTuneSequence *(continued)*

```
   _StuffNoteEvent(*w++, 2, 64, 100, kNoteDuration);    // · violin
   _StuffRestEvent(*w++, kRestDuration);

   // · Make the 5th and 6th notes much softer, just for fun.
   _StuffNoteEvent(*w++, 1, 67, 60, kNoteDuration);     // · piano
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 2, 66, 60, kNoteDuration);     // · violin
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 1, 72, 100, kNoteDuration);    // · piano
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 2, 73, 100, kNoteDuration);    // · violin
   _StuffRestEvent(*w++, kRestDuration);
   _StuffNoteEvent(*w++, 1, 60, 100, kNoteDuration);    // · piano
   _StuffNoteEvent(*w++, 1, 67, 100, kNoteDuration);    // · piano
   _StuffNoteEvent(*w++, 2, 63, 100, kNoteDuration);    // · violin
   _StuffNoteEvent(*w++, 2, 72, 100, kNoteDuration);    // · violin
   _StuffRestEvent(*w++, kRestDuration);
   *w++ = 0x60000000;   // · end-of-sequence marker

goHome:
   return sequence;
}
```

It's important to understand that the duration of a sequence equals the total durations of all the *rest* events. The durations within the note events don't contribute to the duration of the sequence! If two note events occur in a row, each with a duration of say 100, they'll both start at the same time, not 100 time units apart. If the next event is an end-of-sequence marker, the notes will immediately be stopped, having played for zero time units. If, however, a rest event is placed between the note events and the end marker, both notes will sound for the duration of the rest event, up to 100 time units.

PLAYING A TUNE WITH THE TUNE PLAYER

Playing a tune with the tune player component is ideal if for some reason your application will be constructing a tune at run time and then playing it. For prescored music, however, the best solution is to create a QuickTime movie containing only a music track and play it as a regular movie with the Movie Toolbox, as described below.

Using the tune player to play a tune without application intervention is straightforward, as illustrated in Listing 7. After building the tune with BuildTuneHeader and BuildTuneSequence, this routine opens up a connection to the tune player component, calls TuneSetHeader with a pointer to the header information, and then calls TuneQueue with a pointer to the sequence data. All the details of playback are taken care of by the tune player. The tune will stop playing when it reaches the end or when the tune player component is closed.

PLAYING PRESCORED MUSIC IN A QUICKTIME MOVIE

The best way to play prescored music is to create a QuickTime movie with just a music track and play it with the Movie Toolbox, which takes care of details like spooling multiple segments of sequence data from disk. This is currently the only way

Listing 7. Playing a tune with the tune player component

```
void BuildSequenceAndPlay(void)
{
   unsigned long     *header, *sequence;
   TunePlayer        tp;
   TuneStatus        ts;
   ComponentResult   thisError;

   tp = 0;
   header = BuildTuneHeader();
   sequence = BuildTuneSequence();
   if (!header || !sequence)
      goto goHome;
   tp = OpenDefaultComponent(kTunePlayerType, 0);
   if (!tp)
      goto goHome;
   thisError = TuneSetHeader(tp, header);
   thisError = TuneQueue(tp, sequence, 0x00010000, 0, 0x7FFFFFFF,
                          0, 0, 0);

   // · Wait until the sequence finishes playing or the user clicks
   // · the mouse.
spin:
   thisError = TuneGetStatus(tp, &ts);
   if (ts.queueTime && !Button())
      goto spin;     // · I use gotos primarily to shock the children.

goHome:
   if (tp)
      CloseComponent(tp);
   if (header)
      DisposePtr((Ptr) header);
   if (sequence)
      DisposePtr((Ptr) sequence);
}
```

to create QuickTime music that will also play under QuickTime for Windows. There are many tools for authoring music into Standard MIDI Files, which are then easily imported as QuickTime movies — but first let's look at the more hard-core method of creating your own sequence and header data and saving it as a QuickTime movie.

CREATING A QUICKTIME MUSIC TRACK

Creating a QuickTime music track is exactly the same as creating any other kind of track. You create or open the movie you're adding the track to, and then add a new track and a new media followed by a sample description and the sample data. For a music track, the sample description is the tune header information, and the data is one or more tune sequences. The routine in Listing 8 constructs a QuickTime movie with a music track and saves it to disk.

IMPORTING A STANDARD MIDI FILE AS A MOVIE

Most music content exists in a format called Standard MIDI File (SMF). All sequencing and composition programs have an option to Save As or Export files to

Listing 8. Creating a QuickTime music track

```
void BuildMusicMovie(void)
{
   ComponentResult      result;
   StandardFileReply    reply;
   short                resRefNum;
   Movie                mo;
   Track                tr;
   Media                me;
   unsigned long        *tune, *header;
   MusicDescription     **mdH, *md;

   StandardPutFile("\pMusic movie file name:", "\pMovie File", &reply);
   if (!reply.sfGood)
      goto goHome;
   EnterMovies();

   // • Create the movie, track, and media.
   result = CreateMovieFile(&reply.sfFile, 'TVOD', smCurrentScript,
      createMovieFileDeleteCurFile, &resRefNum, &mo);
   if (result)
      goto goHome;
   tr = NewMovieTrack(mo, 0, 0, 256);
   me = NewTrackMedia(tr, MusicMediaType, 600, nil, 0);

   // • Create a music sample description.
   header = BuildTuneHeader();
   mdH = (MusicDescription **)
      NewHandleClear(sizeof(MusicDescription) - 4 + our_header_length);
   if (!mdH)
      goto goHome;
   md = *mdH;
   md->descSize = GetHandleSize((Handle) mdH);
   md->dataFormat = kMusicComponentType;
   BlockMove(header, md->headerData, our_header_length);
   DisposePtr((Ptr) header);

   // • Get a tune, add it to the media, and then finish up.
   tune = BuildTuneSequence();
   result = BeginMediaEdits(me);
   result = AddMediaSample(me, (Handle) &tune, 0, our_sequence_length,
      our_sequence_duration, (SampleDescriptionHandle) mdH, 1, 0, nil);
   result = EndMediaEdits(me);
   result = InsertMediaIntoTrack(tr, 0, 0, our_sequence_duration,
      (1L<<16));
   result = OpenMovieFile(&reply.sfFile, &resRefNum, fsRdWrPerm);
   result = AddMovieResource(mo, resRefNum, 0, 0);
   result = CloseMovieFile(resRefNum);
   DisposePtr((Ptr) tune);
   DisposeMovie(mo);

goHome:
   ExitMovies();
}
```

this format. QuickTime has facilities for reading an SMF file and easily converting it into a QuickTime movie. (QuickTime 2.1 corrects some critical bugs in the 2.0 converter.) During any kind of conversion, the SMF file is assumed to be scored for a General MIDI device, and MIDI channel 10 is assumed to be a drum track.

The conversion to a QuickTime movie can happen in several ways. Because the conversion is implemented in a QuickTime 'eat ' component, it very often will happen automatically. Any application that uses the StandardGetFile routine to open a movie can also open 'Midi' files transparently, and can transparently paste Clipboard contents of type 'Midi' into a movie that's shown with the standard movie controller. To explicitly convert a file or handle into a movie, an application can use the Movie Toolbox routines ConvertFileToMovieFile and PasteHandleIntoMovie, respectively.

For those of you who are hard-core MIDI heads, the following two MIDI system-exclusive messages, new in QuickTime 2.1, may be useful for more precise control of the MIDI import process. (Note that QuickTime data is divided into *media samples*. Within video tracks, each video frame is considered one sample; in music tracks, each sample may contain several seconds worth of musical information.)

- F0 11 00 01 *xx yy zz* F7 sets the maximum size of each media sample to the 21-bit number *xxyyzz*. (MIDI data bytes have the high bit clear, so they have only seven bits of number.) This message can occur anywhere in an SMF file.
- F0 11 00 02 F7 marks an immediate sample break; it ends the current sample and starts a new one. All messages after a sample break message will be placed in a new media sample.

Applications can define their own system-exclusive messages of the form F0 11 7F *ww xx yy zz ... application-defined data ...* F7, where *ww xx yy zz* is the application's unique signature with the high bits cleared. This is guaranteed not to interfere with Apple's or any other manufacturer's use of system-exclusive codes.•

READING INPUT FROM A MIDI DEVICE

If the user has a MIDI keyboard attached to the computer, your application can use it as an input device by calling QTMA routines that capture each event as the user triggers it.

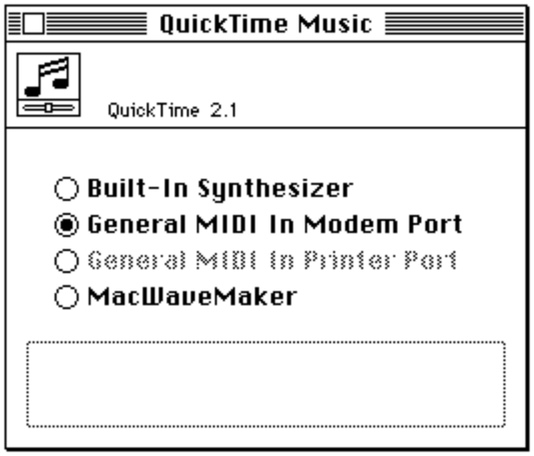
The *default MIDI input* is whichever MIDI port the user has chosen for a General MIDI device from the QuickTime Music control panel, shown in Figure 5. (The default MIDI input can also be specified with the NASetDefaultMIDIInput call in the note allocator, but this call should be made only by music-configuration software, such as the control panel.)

An application can receive MIDI events from the default MIDI input by installing a readHook routine. This routine is called at interrupt level whenever MIDI data arrives. It's installed with the NAUseDefaultMIDIInput call (and later deinstalled with NALoseDefaultMIDIInput).

```
pascal ComponentResult NAUseDefaultMIDIInput(NoteAllocator na,
   MusicMIDIReadHookUPP readHook, long refCon, unsigned long flags);
```

The readHook routine is defined as follows:

```
typedef pascal ComponentResult (*MusicMIDIReadHookProcPtr)
   (MusicMIDIPacket *mp, long myRefCon);
```



QuickTime Music

QuickTime 2.1

○ Built-In Synthesizer
● General MIDI In Modem Port
○ General MIDI In Printer Port
○ MacWaveMaker

Figure 5. The QuickTime Music control panel

When the readHook routine is called, it's passed its refCon (installed with the routine) and a pointer to the MIDI packet. The MIDI packet structure is simply a list of bytes of a MIDI message, preceded by a length:

```
struct MusicMIDIPacket {
   unsigned short length;
   unsigned long  reserved;
   UInt8          data[249];
};
```

The length field is the number of bytes in the MIDI message. (If you're familiar with the MIDI Manager definition of a MIDI packet or with OMS's packet, note that their length field is different from this one: Theirs is the length of both the header and the packet data, so the minimum length would be 6; but in QuickTime's packets, the length field is only the number of bytes of MIDI data actually in the data array.)

In QuickTime 2.0, the reserved field must be set to 0, but in QuickTime 2.1, this field takes on some additional meanings (as reserved fields occasionally do). When an application is using the default MIDI input, it may occasionally lose the use of that input, such as when another application tries to use it, or if the instrument picker dialog box comes to the front. If the use of the input is lost, the reserved field will have the value kMusicPacketPortLost = 1, and the length field will be 0: no MIDI data. When the port is once again available, the readHook routine will receive a packet with the reserved field set to kMusicPacketPortFound = 2, also with no data.

The data array in the MIDI packet contains a raw MIDI message that your readHook routine will have to parse. Our example code parses only the MIDI messages for *note-on events* and *note-off events*; other messages, such as pitch-bend controls, are simply ignored.

The note-on event message has three bytes, 9*c pp vv* (in hexadecimal), where *c* is the MIDI channel that the musical keyboard is transmitting on, *pp* is a MIDI pitch from 0 to 127 (60 is middle C), and *vv* is the velocity with which the key was struck, from 1 to 127. If the velocity is 0, the message signifies a note-off event. Some devices, however, use a separate message type for note-off events; it has the form 8*c pp vv*, where *c* and *pp* are the channel and pitch, and *vv* is the velocity with which the key was released. Nobody in the world pays attention to the release velocity, so in our example we won't either. When an *8c* message is received, we'll just set the velocity to 0 and pretend it was a *9c* message.

Listing 9 shows a readHook routine and the routine that installs it. The main routine, UseMIDIInput, allocates a note channel and then calls NAUseDefaultMIDIInput, specifying a readHook routine that parses note-on or note-off event messages. These messages are expanded into a chord that's played on the note channel. Any packet that isn't of that type — that is, doesn't contain three bytes or start with 0x8n or 0x9n — is ignored.

Listing 9. Parsing MIDI messages in the readHook routine

```
pascal ComponentResult AReadHook(MusicMIDIPacket *mp, long refCon)
{
   MIDIInputExample  *mie;
   Boolean           major;
   short             status, pitch, vel;

   mie = (MIDIInputExample *)refCon;
   if (mp->reserved == kMusicPacketPortLost)    // · port gone? make
                                                // · channel quiet
      NASetNoteChannelVolume(mie->na, mie->nc, 0);
   else if (mp->reserved == kMusicPacketPortFound)    // · port back?
                                                      // · raise volume
      NASetNoteChannelVolume(mie->na, mie->nc, 0x00010000);
   else if (mp->length == 3) {
      status = mp->data[0] & 0xF0;
      pitch = mp->data[1];
      vel = mp->data[2];
      switch (status) {
         case 0x80:
            vel = 0;
         // · Falls into case 0x90.
         case 0x90:
            major = pitch % 5 == 0;
            NAPlayNote(mie->na, mie->nc, pitch, vel);
            NAPlayNote(mie->na, mie->nc, pitch+3+major, vel);
            NAPlayNote(mie->na, mie->nc, pitch+7, vel);
            break;
      }
   }
   return noErr;
}

void UseMIDIInput(void)
{
   ComponentResult   result;
   MIDIInputExample  mie;
   NoteRequest       nr;

   mie.na = OpenDefaultComponent(kNoteAllocatorType, 0);
   if (!mie.na)
      goto goHome;
```

(continued on next page)

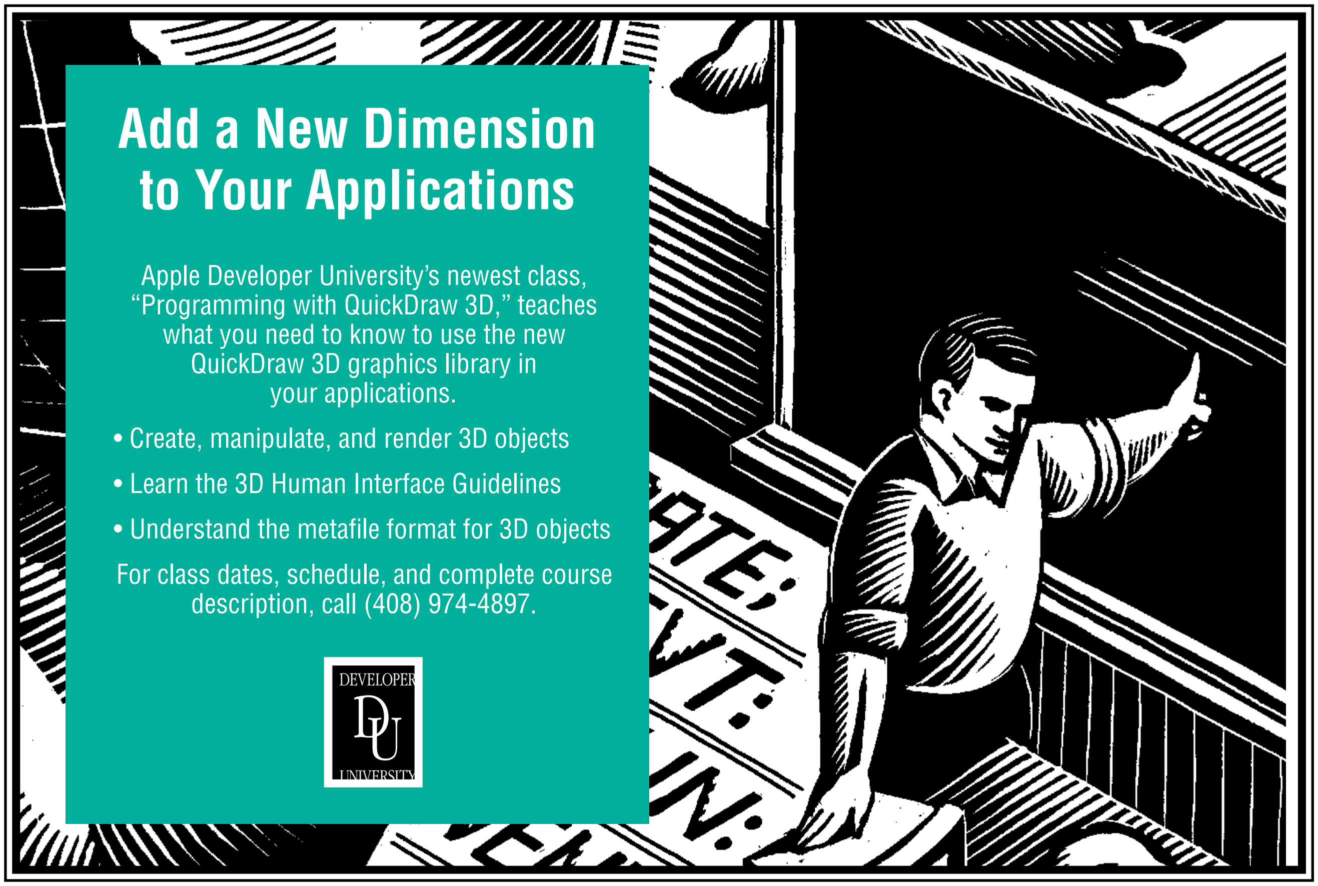
Listing 9. Parsing MIDI messages in the readHook routine *(continued)*

```
   nr.polyphony = 2;
   nr.typicalPolyphony = 0x00010000;
   result = NAStuffToneDescription(mie.na, 1, &nr.tone);   // · piano
   result = NANewNoteChannel(mie.na, &nr, &mie.nc);
   result = NAUseDefaultMIDIInput(mie.na, AReadHookUPP, (long) &mie, 0);
   while (!Button());
   result = NALoseDefaultMIDIInput(mie.na);

goHome:
   if (mie.na)
      CloseComponent(mie.na);    // · disposes of NoteChannel, too
}
```

GIVE QTMA A TRY

Sometimes a little music can make your application easier and more fun to use. Adding music doesn't have to be a complex task; QTMA takes care of all the hard parts, like using MIDI protocols, so you can concentrate more on the music itself. So go ahead, play some tunes and enjoy the music!

Thanks to our technical reviewers Peter Hoddie, Duncan Kennedy, Jim Nitchals, Jim Reekes, and Kent Sandvik.•



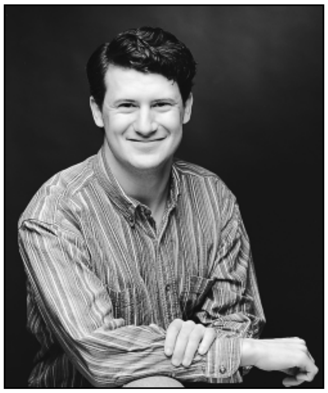
Add a New Dimension to Your Applications

Apple Developer University's newest class, "Programming with QuickDraw 3D," teaches what you need to know to use the new QuickDraw 3D graphics library in your applications.

- Create, manipulate, and render 3D objects
- Learn the 3D Human Interface Guidelines
- Understand the metafile format for 3D objects

For class dates, schedule, and complete course description, call (408) 974-4897.

DEVELOPER DU UNIVERSITY



DAVID HAYWARD

PRINT HINTS

Syncing Up With ColorSync 2.0

In March of this year, Apple announced a major upgrade to the ColorSync extension and API: ColorSync 2.0. Like version 1.0, ColorSync 2.0 is a powerful color management system that allows applications and device drivers to produce consistent color across different devices. However, ColorSync 2.0 dramatically improves the quality, flexibility, and performance of color management. This column focuses on the new features of ColorSync 2.0 and how applications can take advantage of them. (For a good review of ColorSync 1.0 and color management in general, see John Wang's Print Hints column in *develop* Issue 14.)

WHAT IS COLORSYNC 2.0?

ColorSync 2.0 is an extension to the Mac OS that provides a color management system for applications, scanner drivers, printer drivers, and other components of the OS such as QuickDraw and QuickDraw GX. The objective of the color management system is to provide consistent color across devices that have different color ranges, or *gamuts*.

All the versions of QuickDraw GX that have shipped as of this writing (v1.0.1 through v1.1.2) use the ColorSync 1.0 API. ColorSync 2.0 is backward compatible, so QuickDraw GX will work fine if ColorSync 2.0 is installed. QuickDraw GX version 1.2 will add full ColorSync 2.0 support.•

To understand the task of color management, consider the process of scanning, displaying, editing, and printing a color document: In a typical configuration, a color document may interact with three devices — scanner, monitor, and printer — each of which works with color in different ways. A scanner contains a CCD array, which is nonlinearly sensitive to specific frequencies of red, green, and blue light. A monitor hurls electrons at special phosphors to produce varying amounts of red, green, and blue light. And a color printer relies on a mixture of dyes, waxes, or toner to subtract cyan, magenta, yellow, and black from white paper. Because each of these devices uses different physical systems in different color spaces with different gamuts, providing consistent color is difficult. The goal is to provide the best consistency given the physical limitations of each device.

To meet this goal, ColorSync 2.0 requires detailed information about each device and how it represents or characterizes color. This information is encapsulated in a *device profile*. A ColorSync-savvy scanner stores (or "embeds") its profile in the document it creates. A ColorSync-savvy application uses the profile embedded in the document and displays it according to the monitor's profile; a ColorSync-savvy printer renders the document according to the printer's profile.

DEVICE PROFILES

Device profiles are the key ingredient of any color management system because they define the unique color behavior of each device. They're used by *color management module* (CMM) components, which perform the low-level calculations required to transform colors from a source device color space to a destination device color space.

CMM used to stand for *color matching method.* There was disagreement with that name because a CMM component does a lot more than just color matching. So we changed the name to *color management module* to be more accurate.•

ICC profile format. ColorSync 2.0 uses a new profile format defined by the International Color Consortium (ICC), the founding members of which include Apple, Adobe Systems, Agfa-Gevaert, Eastman-Kodak, Microsoft, Silicon Graphics, Sun, and FOGRA (honorary). The *International Color Consortium Profile Format Specification* states the following in its introduction:

> *The intent of this format is to provide a cross-platform device profile format. Such device profiles can be used to translate color data created on one device into another device's native color space. The acceptance of this format by operating system vendors allows end users to transparently move profiles and images with embedded*

DAVID HAYWARD (AppleLink HAYWARD.D) has been working in the Printing, Imaging, and Graphics group in Developer Technical Support for over a year. His proudest achievement to date is the ability to make his hour-long commute every morning without waking up until he hits the speed bumps on Apple's R&D campus. Currently Dave is developing a ColorSync CMM for his closet so that he no longer has to worry about mismatching his clothes.•

profiles between different operating systems. For example, this allows a printer manufacturer to create a single profile for multiple operating systems.

The ICC profile format is designed to be flexible and extensible so that it can be used on a wide variety of platforms and devices. The profile structure is defined as a header followed by a tag table followed by a series of tagged elements that can be accessed randomly and individually. In a valid profile, a minimal set of tags must be present, but optional and private tags may be added depending on implementation needs. Complete definitions of the required tags can be found in the profile format specification. Perhaps just as important, Apple and Adobe have defined how profiles can be embedded in the common graphics file formats PICT, EPS, and TIFF.

There have been changes in the way ColorSync works with profiles as a result of this new format. For example, with ColorSync 1.0, the entire profile format was compact enough to be used as a memory-based data structure, whereas with ColorSync 2.0, profiles can be much larger and typically are disk-based. However, ColorSync 2.0 can still make use of old 1.0 profiles for backward compatibility.

Profile types. There are three main types of device profile: input, display, and output. These types have the following signatures:

- 'scnr' — input devices such as scanners or digital cameras
- 'mntr' — display devices such as monitors or liquid crystal displays
- 'prtr' — output devices such as printers

In addition to these basic types, three other device profile types are defined:

- 'link' — Device link profiles concatenate into one profile a series of profiles that are commonly used together. A profile of this type can simplify and expedite the processing of batch files when the same combination of device profiles and non-device profiles is used repeatedly.
- 'spac' — Color space conversion profiles are used by CMMs to perform intermediate conversions between different device-independent color spaces.
- 'abst' — Abstract profiles provide a generic method for users to make subjective color changes to images or graphic objects by transforming the color data.

Profile quality and rendering intent. Typically you can think of a profile as a self-contained set of data that contains all the information needed for a CMM to perform a color match. Therefore, if an application wants to embed a profile in a document, it shouldn't have to make any changes to the profile — the profile is just a black box of data. This is true for the most part, but there are a few attributes of a profile that an application can change to modify the behavior of the profile. So, it's better to conceptualize a profile as a black box of data with a few switches on the outside. Before embedding a profile in a document, an application can toggle any of these switches by setting the appropriate bit or bits in the profile's header. One of the switches determines the profile's quality and another specifies its rendering intent:

- The quality flag bits provide a convenient place in the profile for an application to indicate the desired quality of a color match (potentially at the expense of speed and memory) as normal, draft, or best quality. In ColorSync 2.0 these qualities do not mandate the use of one algorithm over another; they're just "recommendations" that the CMM may choose to ignore or implement as it sees fit.
- The rendering intent determines how the CMM performs the match. The possible intents are photographic matching, saturation matching, relative colormetric matching, and absolute colormetric matching.

Profile header structure: CMAppleProfileHeader. In the ColorSync 1.0 profile format, the first member of the profile header structure (CMAppleProfileHeader) is a CMHeader structure, which contains all the basic information about the profile. Similarly, the ColorSync 2.0 profile begins with a CM2Header structure. The fields of the CM2Header structure are slightly different from those in the old CMHeader, to reflect some of the improvements provided by the new ICC profile format. However, to be backward-compatible with 1.0, ColorSync 2.0 defines a union of the two header structures. Because the version field is at the same offset in both header structures, it can be used to determine the version of the profile format.

Because ColorSync 2.0 provides support for ColorSync 1.0 profiles, your application should be prepared to handle both formats. Your code should always check the version field of the header before accessing any of the other fields in the header or reading any of the profile's tags.

Profile location structure: CMProfileLocation. ColorSync 2.0 profiles are typically disk-based files, but they can also be memory-based handles or pointers. To allow this flexibility, whenever a profile location needs to be specified (as a parameter for CMOpenProfile, for example) a CMProfileLocation structure is used. This structure contains a type flag followed by a union of an FSSpec, a handle, and a pointer.

Profile reference structure: CMProfileRef. Once a profile has been opened, a private structure is created by ColorSync to maintain the profile until it's closed. A CMProfileRef (defined as a pointer to the private structure) can be used to refer to the profile.

COLOR WORLDS

A *color world* is a reference to a private ColorSync structure that represents a unique color-matching session. Although profiles can be large, a color world is a compact representation of the mapping needed to match between profiles. Conceptually, you can think of a color world as a sort of "matrix multiplication" of two or more profiles that distills all the information contained in the profiles into a fast multidimensional lookup table. A color world can be created explicitly with low-level routines such as NCWNewColorWorld or automatically with high-level routines like NCMBeginMatching.

COLORSYNC 2.0 ROUTINES

Here I'll briefly describe the most commonly used ColorSync 2.0 routines, grouped according to purpose.

> **The API naming convention** is as follows: Calls prefixed with "CM" are high-level color management routines, while those prefixed with "CW" are low-level routines that take a color world as an argument. An "N" before "CM" or "CW" indicates calls that are new to ColorSync 2.0, to distinguish them from the old ColorSync 1.0 calls (which are still supported for backward compatibility).•

Accessing profile files. There is a set of basic routines to work with profiles as a whole. For example, CMNewProfile, CMOpenProfile, CMCopyProfile, and CMGetSystemProfile do what you would expect from their names.

Accessing profile elements. These routines perform more specific operations on profiles and profile elements. CMValidateProfile checks whether a profile contains all the needed tags, CMGetProfileElement gets a specific tag type from a profile, and CMGetProfileHeader gets the important header information of a profile.

Embedding profiles. NCMUseProfile is a simple routine for embedding a profile into a PICT. If you need to extract a profile or embed a profile into a different file format, you can use CMFlattenProfile to embed or CMUnflattenProfile to extract.

QuickDraw-specific matching. These high-level routines provide a basic API to simplify color matching for QuickDraw drawing routines. NCMBeginMatching tells Color QuickDraw to begin matching for the current graphics device using the specified source and destination profiles. NCMUseProfileComment inserts a profile as a picture comment into an open picture. NCMDrawMatchedPicture draws a picture using color matching. CWMatchPixMap matches a PixMap using the specified color world.

Low-level matching. These low-level routines create color worlds and perform color matching. NCWNewColorWorld creates a color world using the specified source and destination profiles, while CWConcatColorWorld creates one using an array of two or more profiles. Using the specified color world, CWMatchColors matches a list of colors and CWMatchBitmap matches a generic bitmap.

Searching profile files. This set of routines allows your application to search the ColorSync™ Profiles folder for the subset of profiles that meets your needs. For example, you could search for only printer profiles and use the search result to provide a pop-up menu for the user. CMNewProfileSearch searches the ColorSync™ Profiles folder for all profile files that match the supplied CMSearchRecord. The matches aren't returned to the caller, but the number of profiles matched and a reference to the search result are returned. The search result is a CMProfileSearch structure that points to private structures maintained by ColorSync and can be accessed with a call like CMSearchGetIndProfile, which opens and returns a CMProfileRef for the *n*th member of the search result.

PostScript code generation. This set of routines allows your application or printer driver to generate PostScript™ code that can be sent to a PostScript Level 2 printer so that the actual matching calculations will be performed in the printer instead of on the user's computer. CMGetPS2ColorRendering gets a color rendering dictionary (CRD) for a specified source and destination profile. CMGetPS2ColorSpace gets a color space array (CSA) for a specified source profile.

BECOMING COLORSYNC-AWARE

At the very least, your application should respect any embedded profiles in the documents it works with. For example, if your application works with PICT files, it shouldn't do anything that would strip out the ColorSync picture comments used for embedding. Even though your application may choose not to make use of the profiles, another application or printer driver may be able to take advantage of them.

PRINTING WITH COLORSYNC

If your application prints QuickDraw data to a ColorSync-savvy printer driver, you need do nothing to get matched output. When the stream of QuickDraw data sent to the driver contains an embedded profile in picture comments, the ColorSync-savvy printer driver will create a new color world to match from the embedded profile to the printer's profile. The driver will then match subsequent QuickDraw operations accordingly before sending them to the printer. If the QuickDraw data stream doesn't contain embedded profiles, the driver will use the current system profile (the profile that the user selected in the ColorSync control panel) as the source profile. That way, the printed output will match the screen display.

One example of a ColorSync-savvy printer driver is the LaserWriter 8.3 driver. Whereas previous versions of LaserWriter 8 allowed the user to choose between "Black and White" and "Color/Grayscale" in the Print dialog, this version adds two new choices. "ColorSync Color Matching" tells the driver to use ColorSync to match an image on the host Macintosh before sending it to the printer. The other option, "PostScript Color Matching," instructs the driver to generate PostScript CSAs and CRDs, which are sent to the printer so that the actual matching is performed in the printer. (The ColorSync API is used to generate the CSAs and CRDs according to the source profiles that may be embedded in the document and the destination profile of the printer.) In either case, the LaserWriter 8.3 driver allows the user to choose a printer profile from a list of printer profiles installed in the ColorSync™ Profiles folder.

Because ColorSync-savvy printer drivers do much of the work for you, it's best if your application prints documents with QuickDraw even if they're not PICT files. For example, if your application reads and prints TIFF files, the best approach is to convert the TIFF data (which may have a profile embedded in tags) to a PicHandle (which would have the profile embedded in picture comments). To print, you draw the PicHandle with DrawPicture into the printer's color graphics port.

If the printer's driver doesn't support ColorSync, your application can still use ColorSync to produce matched output as long as you have an appropriate profile for the device. (There are several commercial tools that build ICC profiles.) Given a source and destination profile, you can use the ColorSync API to match the image or, if your application must send PostScript data directly to a printer, to generate CRDs.

WHAT ELSE A COLORSYNC-SAVVY APPLICATION CAN DO

There is much that an application can do with ColorSync that will help the user work with color. For starters, an application could do the following:

- Provide the user with information on any profiles embedded in a document, and possibly also allow the user to change the quality and rendering intent settings of embedded profiles.
- Include a print preview mode that shows a "soft proof" of the matched output on the display. The application accomplishes this by building a color world with CWConcatColorWorld that matches through three profiles: from the source profile (which is embedded in the document) to the printer's profile (which you allow the user to pick from a list of installed printer profiles) and back to the screen profile (which is the current system profile).
- Along with soft-proofing, it's useful to show the user what colors in the document are out of gamut according to the current destination profile. Gamut checking can be done with routines such as CWCheckColors and CWCheckBitmap.

Note that the LaserWriter engineering team is designing new PrGeneral code for the 8.3.1 version of the driver. This will allow an application to determine what profile is selected in the Print dialog.

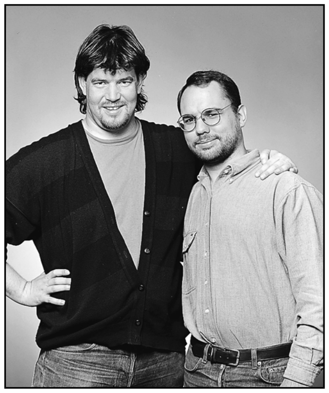
WHERE TO GO FOR MORE

Everything you need to use ColorSync 2.0, including interfaces, libraries, sample code, utilities, and the ICC profile format specification, is on this issue's CD and in the Mac OS Software Developer's Kit. The technical reference for ColorSync 2.0 consists of several chapters in the book *Advanced Color Imaging on the Macintosh*, which is also included on this issue's CD and will soon be available in print from Addison-Wesley; this documentation covers everything from a high-level discussion of color management theory to a detailed description of the ColorSync 2.0 API. Why not take a closer look and see how you can take of advantage of this new improved technology in your application?

Thanks to Paul Danbold, Steve Swen, Nick Thompson, and John Wang for reviewing this column.•

The Basics of QuickDraw 3D Geometries

No matter how realistic or sophisticated you want your 3D images to be, you must always build objects with the primitive geometric shapes provided by the graphics system. Our article in Issue 22 gave the basic information you need to start developing applications with QuickDraw 3D. Here we delve deeper into the primitive geometric shapes provided by QuickDraw 3D and show how to use them effectively. We also give you some tips we've gained from working with developers.



NICK THOMPSON AND PABLO FERNICOLA

Geometric shapes — or geometries — form the foundation of any 3D scene. QuickDraw 3D provides a rich set of primitive geometric types that you use to define the shapes of things. You can apply attributes (such as colors) to geometric objects, collect geometric objects into groups, and copy, illuminate, texture, transform, or otherwise modify them to attain the visual effects you want. In other words, everything that's drawn by QuickDraw 3D is either a geometry or a modification of a geometry. So you need to know how to define geometries (and usually also how to create and dispose of them) to work effectively with QuickDraw 3D. This article describes the geometries available in QuickDraw 3D version 1.0 and shows how they relate to other aspects of the QuickDraw 3D architecture (such as the class hierarchy).

We're assuming that you're already familiar with the basic capabilities of QuickDraw 3D. For a good introduction, see our article "QuickDraw 3D: A New Dimension for Macintosh Graphics" in Issue 22 of *develop* (a copy is on this issue's CD). In that article, we provided an overview of QuickDraw 3D's architecture and capabilities. You can think of QuickDraw 3D as having three main parts: graphics, I/O (the QuickDraw 3D metafile), and human interface guidelines. Here, we provide more detail on the graphics portion of the QuickDraw 3D API and highlight some parts of that API that could use clarification as you try to implement geometries.

NICK THOMPSON (AppleLink NICKT) from Apple's Developer Technical Support group took a trip to Las Vegas this year in a rented Cadillac. He was impressed by some of the ancient architecture on show in this fine city, such as the Pyramid of Luxor, Excalibur's Castle, and Caesar's Palace (he was surprised that the ancient Egyptians, King Arthur, and the Roman emperor had all made it that far west). He was also impressed by the free food and drinks — all he had to do was sit at a table and buy small plastic disks with green scraps of paper that he got from a hole in the wall. Having rented a Cadillac for this trip, Nick now has his heart set on a 1968 Eldorado convertible.•

PABLO FERNICOLA (AppleLink PFF, eWorld EscherDude), the short one in the picture, is the brains behind the operation. His hobbies include traveling to exotic places (such as the local supermarket), eating fine cuisine, and talking to his dog (who is almost as big as Nick, and probably a lot smarter). He's hard at work on the next generation of QuickDraw 3D, which — like Pablo — is bound to be even smarter. Pablo says, "You can use QuickDraw 3D's metafile format everywhere, even for defining virtual environments on the net. So get those applications ready, won't you?"•

To help you get started using geometries, this issue's CD contains version 1.0 of the QuickDraw 3D shared library and programming interfaces, sample code, and an electronic version of the book *3D Graphics Programming With QuickDraw 3D*, which provides complete documentation for the QuickDraw 3D programming interfaces.

A WORD ABOUT RENDERING AND SUBMITTING

Our previous article included an introduction to rendering; we'll review a key concept here — retained vs. immediate rendering. We'll also elaborate on an important point we glossed over in that article: submitting something to be rendered rather than just rendering it. These concepts will help set the stage for what you'll learn here about working with geometries.

RETAINED VS. IMMEDIATE MODE RENDERING

A powerful feature of QuickDraw 3D is that it supports both retained and immediate modes for rendering geometric data; you can even mix these modes within the same rendering loop. In *retained mode*, the definition and storage of the geometric data are kept internal to QuickDraw 3D — as abstract geometric objects. In *immediate mode*, the application keeps the only copy of the geometric data; for efficiency, the application should use QuickDraw 3D data structures to hold the data, but those structures can be embedded in application-defined structures. Retained mode geometric objects and immediate mode geometric data define the shapes of objects. You'll typically use one or more primitive geometric types provided by QuickDraw 3D (such as triangles or meshes) to build up a scene.

Whether you use retained or immediate mode to render geometries usually depends on how much of a model changes from one rendering operation to the next. As we'll illustrate with examples in this section, we prefer to use retained geometries most of the time and to use immediate mode only for temporary objects. Since our preference for retained mode is a departure from the traditional QuickDraw way of drawing, we'll attempt to convince you that retained mode is a much more efficient method of rendering geometries.

Immediate mode. When you use immediate mode rendering, the data that defines a geometry is stored and managed by your application. For example, to draw a triangle you would write code similar to that in Listing 1. If you wanted to draw this triangle many times, or from different camera angles, you would have to maintain the data in your application's data structures.

Typically when using immediate mode, you stick to a single type of geometry (triangles are popular with developers accustomed to lower-level 3D graphics

Listing 1. Rendering a triangle in immediate mode

```
TQ3TriangleData   myTriangle;

// Set up the triangle with appropriate data.
...
// Render the triangle.
Q3View_StartRendering(myView);
do {
   Q3Triangle_Submit(&myTriangle, myView);
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

libraries). If you use multiple geometric types, you need to define a data structure to manage the order of the geometries. An example of rendering several geometries in immediate mode is shown in Listing 2.

Listing 2. Rendering several geometries in immediate mode

```
typedef struct myGeometryStructure {
   TQ3ObjectType                type;
   void                         *geom;
   struct myGeometryStructure   *next;
} myGeometryStructure;

myGeometryStructure    *currentGeometry;
...
Q3View_StartRendering(myView);
do {
   while (currentGeometry != NULL) {
      switch (currentGeometry->type) {
         case kQ3GeometryTypeTriangle:
            Q3Triangle_Submit((TQ3TriangleData *) currentGeometry->geom,
               myView);
         break;
         case kQ3GeometryTypePolygon:
            Q3Polygon_Submit((TQ3PolygonData *) currentGeometry->geom,
               myView);
         break;
      }
      currentGeometry = currentGeometry->next;
   }
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

If you wanted to apply transforms to a geometry as it's being drawn, you would have to add a new case to the switch statement. This gets complicated pretty quickly. As a result, many developers, when given a choice, will use immediate mode only for models that have a fixed geometry and are not being altered.

Retained mode. Creating geometric objects allows renderers to take advantage of characteristics of particular geometries and thus optimize the drawing code. The code in Listing 3 draws a triangle in retained mode.

SUBMITTING

You'll notice that the routine to draw an object is Q3Object_Submit. This probably seems a bit strange: why didn't we call it Q3Object_Draw? The reason is that there are four occasions in which you need to specify a geometry — when writing data to a file, when picking, when determining the bounds of a geometry, and when rendering — and QuickDraw 3D provides a single routine that you use in all of these cases. To indicate which operation you want to perform, you call the Submit routine inside a loop that begins and ends with the appropriate calls. For instance, to render a model, you call Submit functions inside a rendering loop, which begins with a call to Q3View_StartRendering and ends with a call to Q3View_EndRendering (as shown in Listing 3). Similarly, to write a model to a file, you call Submit functions inside a writing loop, which begins with a call to Q3View_StartWriting and ends with a call to Q3View_EndWriting.

Listing 3. Rendering a triangle in retained mode

```
TQ3TriangleData   triangleData;

// Set up the triangle with appropriate data.
...
// Create the triangle.
triangleObject = Q3Triangle_New(&triangleData);
// Render the triangle.
Q3View_StartRendering(myView);
do {
   Q3Object_Submit(triangleObject, myView);
} while (Q3View_EndRendering(myView) == kQ3ViewStatusRetraverse);
```

Listing 4. A submitting function

```
// Submit the scene for rendering, file I/O, bounding, or picking.
TQ3Status SubmitScene(DocumentHdl theDocument)
{
   TQ3Vector3D    globalScale, globalTranslate;

   globalScale.x = globalScale.y = globalScale.z = 
                                      (**theDocument).fGroupScale;
   globalTranslate = *(TQ3Vector3D *)&(**theDocument).fGroupCenter;
   Q3Vector3D_Scale(&globalTranslate, -1, &globalTranslate);
   Q3Style_Submit((**theDocument).fInterpolation, 
      (**theDocument).fView);
   Q3Style_Submit((**theDocument).fBackFacing, (**theDocument).fView);
   Q3Style_Submit((**theDocument).fFillStyle, (**theDocument).fView);

   Q3MatrixTransform_Submit(&(**theDocument).fRotation, 
      (**theDocument).fView);
   Q3ScaleTransform_Submit(&globalScale, (**theDocument).fView);
   Q3TranslateTransform_Submit(&globalTranslate, (**theDocument).fView);
   Q3DisplayGroup_Submit((**theDocument).fModel, (**theDocument).fView);

   return (kQ3Success);
}
```

We recommend that you put all your Submit calls together within a single function (such as the one shown in Listing 4) that you can then call from your rendering loop, picking loop, writing loop, or bounding loop. Organizing your code in this fashion will prevent a common mistake: creating rendering loops that are out of sync with picking or bounding loops. It also simplifies your rendering and picking loops — you just call your submitting function from within the loop. Here's an example of calling the function in Listing 4 from within a rendering loop:

```
Q3View_StartRendering((**theDocument).fView);
do {
   theStatus = SubmitScene(theDocument);
} while (Q3View_EndRendering((**theDocument).fView) == 
                                                  kQ3ViewStatusRetraverse);
```

QUICKDRAW 3D CLASS HIERARCHY

Even if you perform all your rendering in immediate mode — that is, without creating any QuickDraw 3D geometric objects — you still need to create some QuickDraw 3D objects, such as a view, camera, and draw context, in order to render any image at all. So working with geometries in QuickDraw 3D means working with at least some objects. Before going into detail about how to create and use QuickDraw 3D geometric objects, let's review the object system and some of its basic classes.

QuickDraw 3D is an object-based system. We chose to implement the API with the C language, which doesn't support objects directly; nevertheless QuickDraw 3D is organized into a definite class hierarchy. Figure 1 shows part of this hierarchy, emphasizing the classes that are discussed in this article. At the top of the class hierarchy is the basic QuickDraw 3D Object class. Geometries, such as the triangle, polygon, and mesh classes, are at the bottom of the hierarchy.

> **The Object class** is really named TQ3Object. This article uses shortened forms of the QuickDraw 3D class names.•

You can determine the class in which a function is defined simply by looking at the function's name: function names have the form Q3*class-name_method*. For example, the function Q3Shared_GetReference is defined in the Shared class and returns a reference to the shared object that's passed as an argument. The function Q3Object_Dispose is defined in the Object class; it accepts any QuickDraw 3D object as an argument (since Object is the root class) and disposes of it.

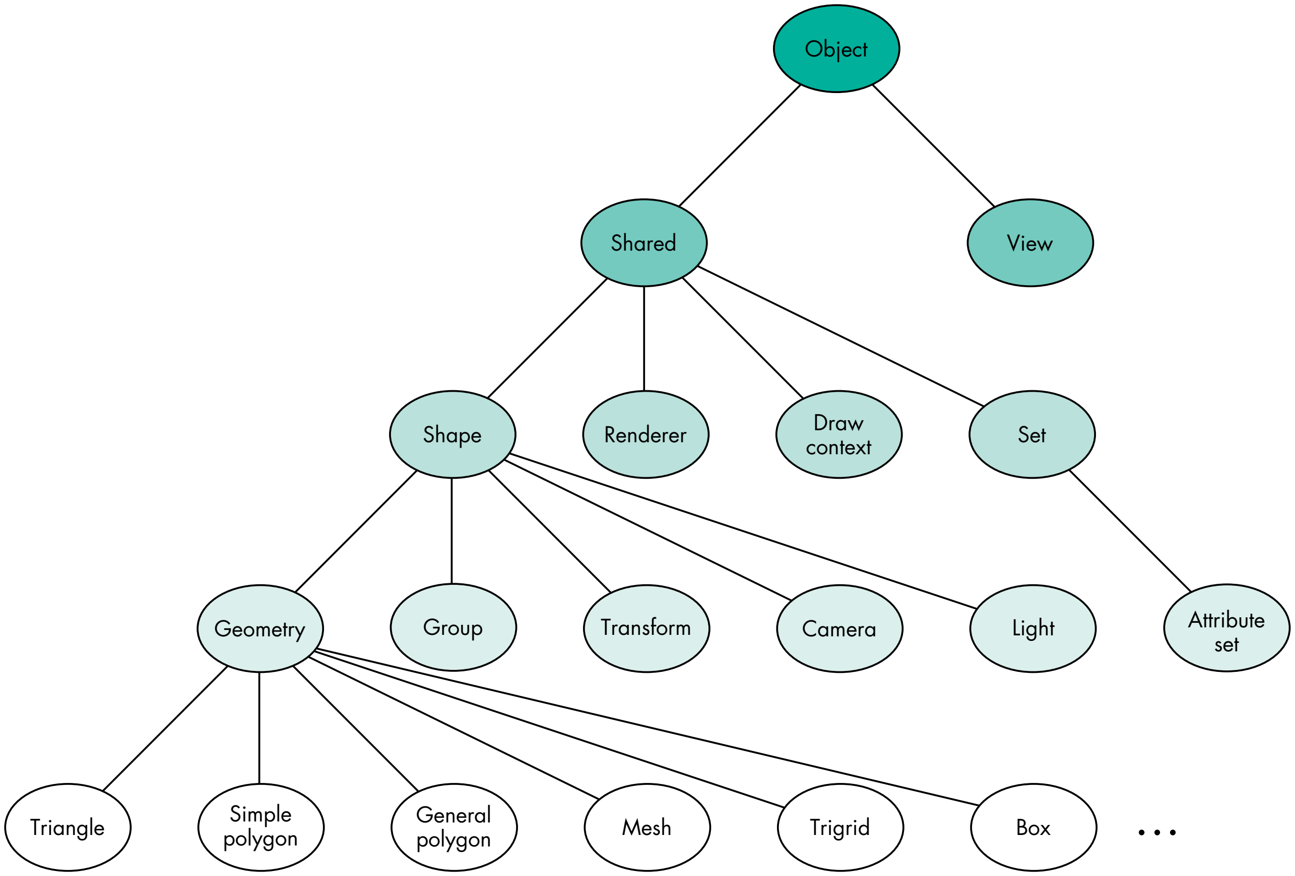


Figure 1. Partial QuickDraw 3D class hierarchy

In the following sections, we'll talk more about the classes shown in Figure 1 and answer some questions developers have had about using them when working with geometries. Then we'll (finally!) talk about the geometric objects themselves and provide sample code for using many of them.

THE SHARED CLASS

Generally speaking, drawing anything with QuickDraw 3D involves working with objects that inherit from the Shared class. There can be multiple references to shared objects (hence the name); the way QuickDraw 3D determines whether a shared object is still referenced is by way of a *reference count*, initially 1. Developers new to QuickDraw 3D are sometimes confused by reference counts, but they're actually very straightforward. When you create a shared object, its reference count is 1. For example:

```
myNewObject = Q3Mesh_New();
// myNewObject now has a reference count of 1.
```

When you get a shared object as a result of a Get call, or pass one as an argument in an Add or Set call, the object's reference count is incremented.

```
// The following calls increment the object's reference count.
Q3Group_GetPositionObject(myGroup, currentPosition, &myExistingObject);
...
Q3Group_AddObject(myGroup, myObject);
...
Q3View_SetDrawContext(myView, myDrawContext);
```

Passing a shared object as the argument to a Dispose call decrements its reference count; only when the count goes to 0 does QuickDraw 3D actually dispose of the memory occupied by the object. As a general rule, you should dispose of the object before the scope of the variable expires. For example:

```
{  // Start of the block. Variables come into scope.
   TQ3Object   myObject = Q3Mesh_New();  // The start of myObject's scope

   // Do something that manipulates myObject.
   ...
   // The scope of myObject is going to end at the next closing brace,
   // so dispose of it before we go out of scope.
   Q3Object_Dispose(myObject);
}  // End of the block.
```

If you were assigning an object reference to a global variable, you would dispose of the object before calling Q3Exit and exiting your program.

Q: Why does my application crash when I call Q3Exit?

A: In the debugging version of QuickDraw 3D, Q3Exit generates a debugging message for each remaining object. The default behavior is to display the message with the DebugStr call; the message is displayed in MacsBug (or whatever debugger you use). So your application isn't crashing; it's trying to tell you to tidy up after yourself! To avoid this unscheduled trip into your debugger, you can install your own error handler and log the message to a file. And, of course, you should fix your application so that it doesn't leak memory!•

Let's take a closer look at what happens to reference counts when you create and dispose of a shared object. Figure 2 shows the typical lifetime of a group of QuickDraw 3D objects (we'll talk more about groups later).

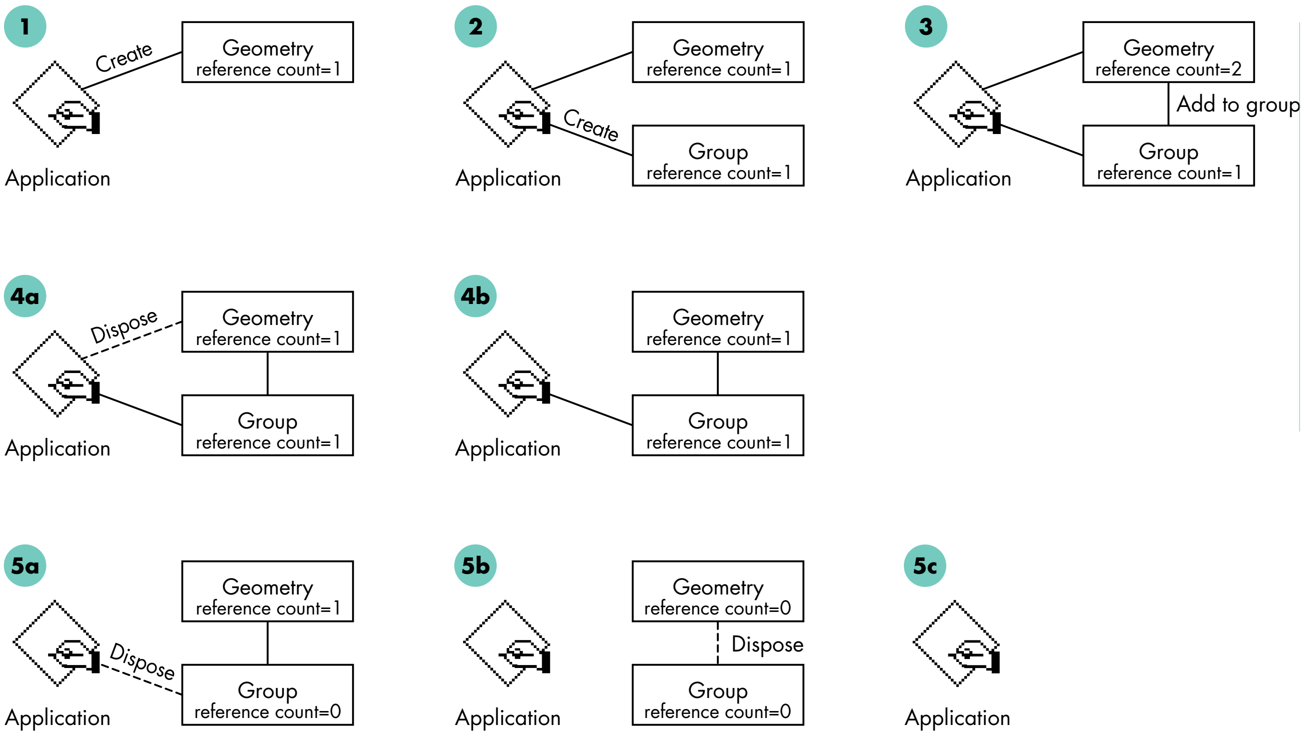


Figure 2. Reference counts in QuickDraw 3D

1. An application creates a geometric object. Its reference count is 1.
2. The application creates a group object. Its reference count is also 1.
3. The application adds the geometry to the group (by calling the function Q3Group_AddObject), which increments the reference count of the geometric object (to 2).
4. The application disposes of the geometric object (by calling the function Q3Object_Dispose), which is safe to do once it's added to the group. This decrements the reference count of the geometry back to 1. The application can then operate on the group (which now contains the geometry).
5. When it's finished with the group, the application can dispose of the group object. This lowers the reference count of the group to 0, which causes QuickDraw 3D to dispose of the group and of all the objects within the group. As you can see, the geometry is disposed of as a side effect of disposing of the group.

THE VIEW CLASS

The view object ties together the elements required to draw a scene; it's the central object that holds the state information for rendering a scene. A *scene* consists of the geometry being drawn (hereafter referred to as the *model*), together with the light, camera, draw context, and other objects. Our previous article discussed how to set up a view; we'll expand on that discussion by describing how to create and manage multiple scenes of a model.

To display a scene, you need at least one view object, and each view object must have a camera associated with it. Each of your application's windows usually has one view object attached to it. When you need to display multiple scenes of the same model, you can create multiple windows, each with its own view object. Then you simply submit the model to the desired view. Alternatively, you can display multiple scenes using a single view object by setting up several different cameras and draw contexts and switching between them — manipulating the view's camera to create each scene (see Figure 3).

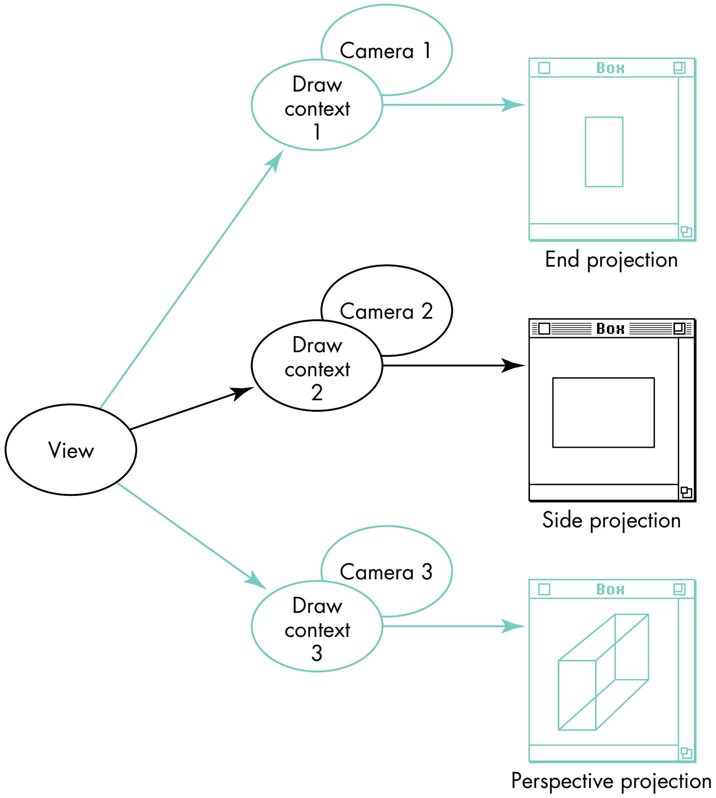


Figure 3. Multiple scenes of the same model

You can have only one active draw context and camera for each view object, so to update one of your windows, you need to manually set the active draw context and camera for the appropriate scene. For this reason, the first option (one view per window) is usually simpler to implement.

THE GROUP CLASS

QuickDraw 3D provides a number of classes for grouping objects together. Groups are useful because they provide a structure to your models, allowing you to express the relationship between different geometric objects. Of course, if you want to use your own data structures for storing your geometries, you can do so, but generally it's more work. Using QuickDraw 3D's group classes, you can create hierarchies of geometric data by nesting groups within other groups. Figure 4 shows the group classes provided with QuickDraw 3D.

You can create a group object by calling Q3Group_New. This creates an object belonging to the generic Group class. QuickDraw 3D provides the following subgroups of the generic Group class, which are distinguished by the types of objects you're allowed to place in them:

- A *light group* places the light objects for a scene in a group, which simplifies lighting management. For example, you could provide an iterator function to loop through the group and turn all the lights on or off.

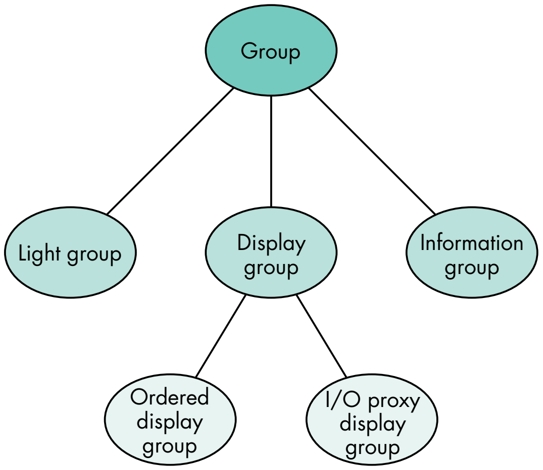


Figure 4. Group classes provided by QuickDraw 3D

- A *display group* manages objects that are drawable, including geometries, styles, and transforms. You can use the function Q3Object_IsDrawable to confirm whether an object is drawable.
- An *information group* stores informational strings, such as the author, copyright, trademark, and other human-readable information within a metafile.

Because we want to talk about geometries, which are drawable objects, we'll concentrate on display group objects. In addition to "plain" display groups, there are two specialized subclasses of the display group class: ordered and I/O proxy. For a plain display group, the order in which items are placed in the group is the order in which they're drawn when the group is submitted, regardless of the class that the objects belong to. For an ordered display group, objects in the group are sorted by object type and are submitted (when you call Q3DisplayGroup_Submit) in the following order: transforms, styles, attribute sets, shaders, geometric objects, groups.

Ordered display groups are most useful when you want to operate on a group of objects as a single entity. For example, you know that transforms are always at the start of the group, so you could manipulate the transform to alter the orientation of the entire group. (If you were using a plain display group, you would have to search for the transform, or otherwise store a reference to it, which makes life more complicated.) Sometimes you'll want to nest a number of ordered display groups within a plain display group. If you were animating a robotic arm, for example, each component of the arm could be an ordered display group that's nested within a plain display group.

You can use I/O proxy display groups to provide multiple representations of the same data. This is useful when dealing with applications that aren't based on QuickDraw 3D or that run on other platforms. For example, some applications might be able to read only mesh objects; your application may want to use NURB patches (another type of geometric object), but you want other applications to be able to read your metafiles. In this case, you could write a NURB patch representation of your data, followed by a mesh representation. To provide both representations of the same data in a metafile, you would create an I/O proxy group that contains the NURB patch object first and the mesh object second, and write the group to the metafile. When you draw with QuickDraw 3D, the objects that appear first in the group are preferred over later objects in the group.

THE TRANSFORM CLASS

The Transform class enables you to change the position, orientation, or size of geometries. When you specify the coordinates for the vertices that define a geometry, the *x*, *y*, *z* values are expressed as floating-point values in *local coordinates*. Rendering, however, and associated operations like backface removal and lighting are performed in *world coordinates*. To transform a geometry from one space to another, QuickDraw 3D multiplies the local coordinates by a local-to-world matrix. The default value for this matrix is the identity matrix, which leaves the original geometry unchanged. By changing the value of the local-to-world matrix, you can transform geometries without having to change the geometries' coordinates.

Using an example from our previous article, let's say that you have a model that contains several boxes (see Figure 5). We could enter the coordinates for the points that make up each of the four boxes, but that's a lot of work (and if you're creating an object for each box, it's a waste of memory). Instead, we define one box at a certain location and call it the reference box. To get the effect of four boxes in different locations, we draw the reference box four times — changing the local-to-world matrix each time before drawing.

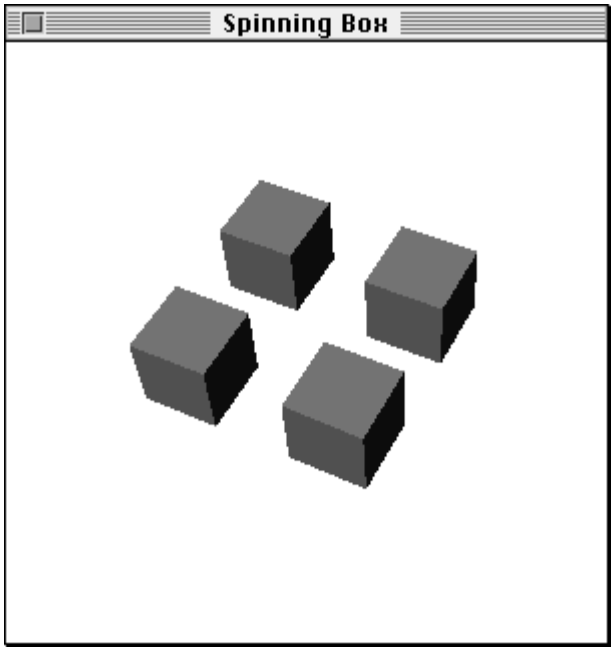


Figure 5. Boxes drawn by changing the local-to-world matrix four times

If you look in the file QD3DTransform.h, you'll notice that there are several different types of transforms. The most general type is the matrix transform, which is a 4 x 4 matrix. To use this transform, you supply the translation, rotation, and scale values in the appropriate entries of the matrix, as shown in Figure 6. You can do any type of transform that can be expressed as a 4 x 4 matrix. In the figure, you can see that the upper 3 x 3 submatrix is a rotation matrix, with the entries in the main diagonal containing the scale factors for x, y, and z. The lower row contains the translation factors.

If you know which type of transform you'll be applying, however, it's better to use one of the more specific types. In this way, QuickDraw 3D renderers and shaders will be able to take advantage of the information contained in the transform; for example, if your local-to-world matrix is just a translate transform, the renderer

$$\begin{bmatrix} S_x * R_{0,0} & R_{0,1} & R_{0,2} & 0.0 \\ R_{1,0} & S_y * R_{1,1} & R_{1,2} & 0.0 \\ R_{2,0} & R_{2,1} & S_z * R_{2,2} & 0.0 \\ T_x & T_y & T_z & 1.0 \end{bmatrix}$$

Note: S is the scale transform, R is the rotate transform, and T is the translate transform.

Figure 6. A matrix transform

doesn't have to transform normals before performing the backface removal operation (because directions are not affected by translations). Also, using the more specific types provides a better abstraction and tends to make the logic of your code easier to understand (and you don't have to deal with all those pesky matrices).

When you change the local-to-world matrix by applying transforms, each transform is concatenated as it's applied through a Submit call. For example, if before drawing a point object, we submit a translate transform, a rotate transform, a scale transform, and then a point, the point will be transformed as follows:

$$p' = p * S * R * T$$

p' is the resulting transformed point and p is the original point. T is the matrix containing the translate operation, R is the matrix containing the rotate operation, and S is the matrix containing the scale operation.

You can apply transforms either by using immediate mode calls or by creating transform objects — just as you do for geometries. Note that transforms accumulate; that is, if you apply a translation, any objects drawn after that will be translated by the same amount. If you want a transform to apply to a certain object only, you can use the Q3Push_Submit and Q3Pop_Submit calls around it or place the object in a group, since groups perform an implicit push and pop (you can change this behavior if you want).

So, let's build on what we've learned so far. We want to draw the model shown in Figure 5. Let's first do it by submitting new transforms in immediate mode, before each box is drawn, as shown in Listing 5.

Alternatively, we could create the model of the four boxes as a group, as shown in Listing 6.

THE ATTRIBUTE SET CLASS

Attributes affect the way an object is rendered in QuickDraw 3D. A view has a default set of attributes, defined in the QD3DView.h file, that can be modified to suit a particular application. If no attributes are supplied for the objects being rendered within a view, the default view attributes are applied. Attributes can be applied in a number of ways: by submitting them to a view object; by adding them to a group; or by attaching them to a geometry, to a geometry's face, or to each vertex of a geometry.

The order in which attribute sets are applied during rendering is based on a fixed hierarchy, as illustrated in Figure 7. Attributes of the same type (such as diffuse color) can override one another; they use the following preference hierarchy, from highest to lowest precedence: vertex, face, geometry, group, view. For example, a specular color attribute at the vertex level does not override a diffuse color attribute at the geometry level, whereas a specular color attribute at the vertex level does override a

Listing 5. Using translate transforms in immediate mode

```
Q3View_StartRendering(viewObject);
do {
   TQ3Vector3D translationX = {2.0,  0.0, 0.0},
               translationY = {0.0, -2.0, 0.0};

   Q3View_Push(viewObject);

   // Note how we are using a retained mode geometry with immediate mode
   // transforms. As we execute each of the calls, the boxes are drawn.

   Q3Object_Submit(referenceBox, viewObject);
   // Move to the right.
   Q3TranslateTransform_Submit(&translationX, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
   // The Pop will move back to the left.
   Q3View_Pop(viewObject);
   // Move down.
   Q3TranslateTransform_Submit(&translationY, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
   // Move to the right.
   Q3TranslateTransform_Submit(&translationX, viewObject);
   Q3Object_Submit(referenceBox, viewObject);
} while (Q3View_EndRendering(viewObject) == kQ3ViewStatusRetraverse);
```

Listing 6. Creating translate transform objects

```
TQ3GroupObject       myModel;
TQ3Vector3D          translationX = {2.0, 0.0, 0.0},
                     translationYAndNegativeX = {-2.0, -2.0, 0.0};
TQ3TransformObject   xform_x, xform_yx;

// Note that as we execute these calls, nothing is drawn.

myModel = Q3Group_New();
xform_x = Q3TranslateTransform_New(&translationX);
xform_yx = Q3TranslateTransform_New(&translationYAndNegativeX);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_x);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_yx);
Q3Group_AddObject(myModel, referenceBox);
Q3Group_AddObject(myModel, xform_x);
Q3Group_AddObject(myModel, referenceBox);

// To draw the boxes, you would call Q3Object_Submit(myModel, myView)
// within a submitting loop.
```

specular color attribute at the geometry level (because they are attributes of the same type). If attributes at any level are not supplied, the parent's attributes apply. If there are no attributes supplied anywhere in the hierarchy, the default attribute set for the view will be used.

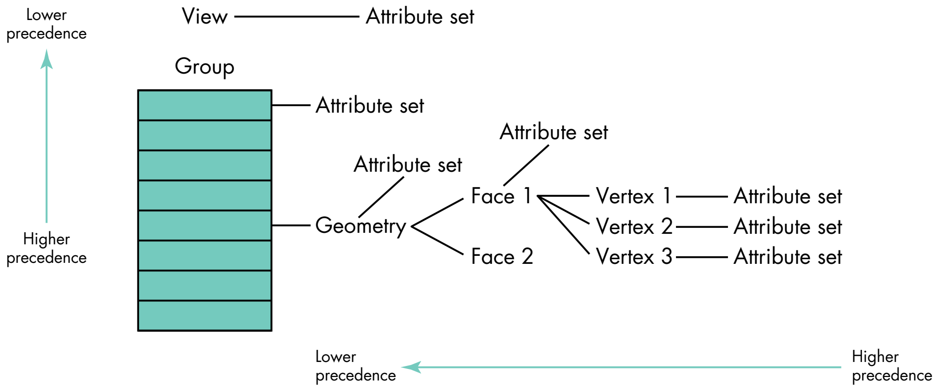


Figure 7. Hierarchy of applying attributes to a geometry

Here are the six most commonly used predefined attribute types that you can specify (there are 12 in all):

- The *diffuse color* is the actual color of the object.
- The *specular color* is the color of the light reflected by the object, which may or may not be the same as the diffuse color.
- The *specular control* determines how much light of the specular color is reflected.
- The *ambient coefficient* determines how much the ambient lighting affects the object.
- The *surface UV* attribute specifies how a texture is mapped to a geometry's vertex.
- A *texture shader* can be applied to a surface that has UV parameterization applied (more on this later).

You can also define your own custom attributes. Later, in the geometry code samples, we'll create attribute sets to affect the way the geometries are drawn.

BUILDING GEOMETRIES

Now we're ready to look at the specific geometries and show how to build them. QuickDraw 3D version 1.0 supports 12 geometries (illustrated in Figure 8). In the code examples later in this article, we'll cover the most commonly used geometries.

- A *marker* object is a bitmap that's displayed face-on at any orientation — similar to a sprite. It's useful for denoting the position of objects and for providing annotations, such as labels on objects in a 3D chart.
- A *point* object is the most basic object in QuickDraw 3D; it specifies discrete points in a scene.
- A *line* object is a line between two points.
- A *polyline* object is a line that consists of multiple segments.
- A *triangle* object is a closed planar geometry defined by three intersecting lines. It's the simplest form of a polygon.
- A *simple polygon* object is a planar geometry described by a list of vertices; it's a figure formed by a closed chain of intersecting straight lines. A simple polygon consists of a single convex contour and may not contain holes.

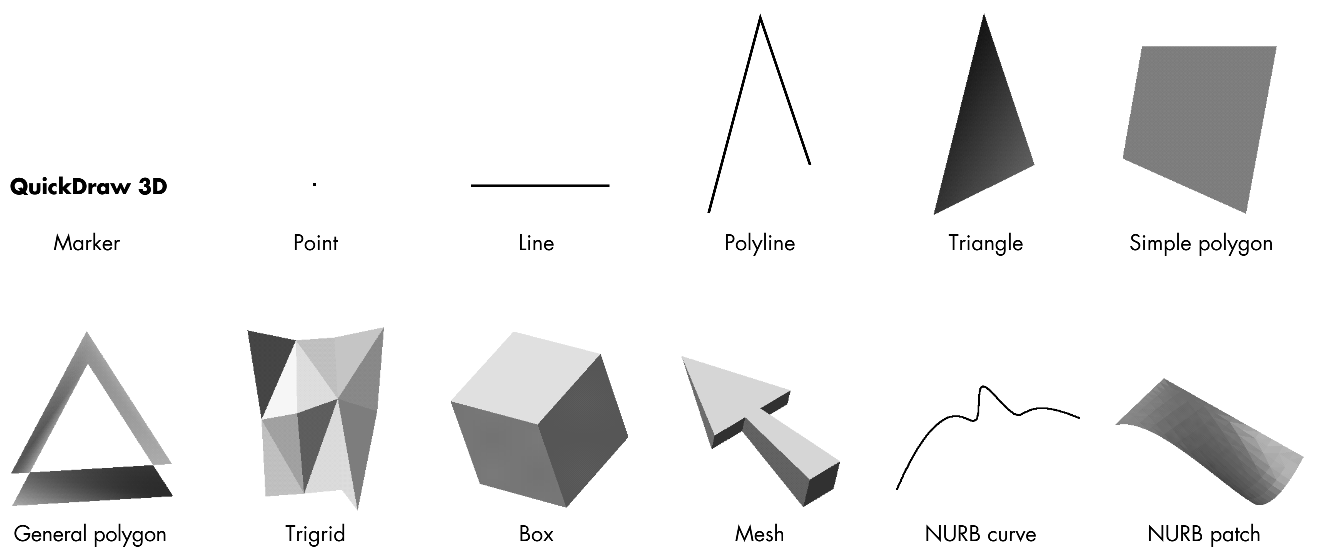


Figure 8. QuickDraw 3D geometries supplied in version 1.0

- A *general polygon* object is a planar geometry that may contain holes, be concave, and consist of one or more contours.
- A *trigrid* object is a grid whose surface consists of multiple triangles that share edges and vertices.
- A *box* object is a three-dimensional rectangular object.
- A *mesh* object is a collection of vertices, faces, and edges that represent a topological polyhedron. It's sometimes referred to as a winged-edge structure.
- A *NURB curve* object is a curve described by a NURB equation.
- A *NURB patch* object is a three-dimensional surface described by a NURB equation.

NURB stands for nonuniform rational B-spline. A B-spline is a parametric curve (a curve defined by coordinates derived from functions sharing a common parameter) whose shape is determined by a series of control points whose influence is described by basis functions.•

SIMPLE GEOMETRIES

Let's start with some simple geometries first: lines, polylines, triangles, simple polygons, and general polygons. In essence, these are the building blocks for QuickDraw 3D. You can use combinations of these to construct your model, or you can use some of the composite geometries, such as meshes and trigrids (described later).

Line and polyline objects. Lines are defined by two noncoincident points. If you want to have multiple line segments, you can use polylines (see Listing 7). In polylines, every vertex after the first one defines a new line. You can attach attributes at the geometry level or at the vertex level (which is useful for having multicolored lines, but remember that you need to use per-vertex interpolation when rendering in order for the multiple colors to apply).

Triangle objects. Triangles are the most basic of the planar geometries in QuickDraw 3D. Triangles are defined by three noncolinear, noncoincident vertices.

Listing 7. Creating a polyline

```
TQ3ColorRGB          polyLineColor;
TQ3PolyLineData      polyLineData;
TQ3GeometryObject    polyLineObject;

static TQ3Vertex3D   points[4] = {
                       { { -1.0, -0.5, -0.25 }, NULL }, // first vertex
                       { { -0.5,  1.5,  0.45 }, NULL }, // second vertex
                       { {  0.0,  0.0,  0.0  }, NULL }, // third vertex
                       { {  1.5,  1.5,  1.0  }, NULL }  // fourth vertex
                     };

// The polyline has four vertices.
polyLineData.numVertices = 4;
polyLineData.vertices = points;

// Add a color to the line as a whole.
polyLineData.polyLineAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&polyLineColor, 0.4, 0.2, 0.9);
AttributeSet_AddDiffuseColor(polyLineData.polyLineAttributeSet,
   &polyLineColor);

// Create the polyline.
polyLineObject = Q3PolyLine_New(&polyLineData);

Q3Object_Dispose(polyLineData.polyLineAttributeSet);
```

In Listing 8, we set a color attribute for the entire geometry and for the individual vertices. When you draw the triangle with flat interpolation, the geometry color is used; when you draw it with per-vertex interpolation, however, the vertex attributes take precedence and you can see a color ramp on the triangle (see Figure 8, where the color ramp is approximated in grayscale).

Simple polygon and general polygon objects. Simple polygons and general polygons are planar objects with multiple vertices. Simple polygons must be convex, but general polygons can be either convex or concave. In addition, general polygons can be self-intersecting and have multiple contours.

As was shown in Figure 8, a general polygon can have a “hole” in it, but a simple polygon never does. This is the primary difference between the two geometries. Processing general polygons takes more time than processing simple polygons, so we advise you to use simple polygons whenever possible.

If the geometry you’re creating is convex, you should use simple polygons to achieve better performance. If your polygons always have three vertices, however, you should opt for triangles. If you don’t know what your geometry looks like (for example, it’s being built by the user on the fly and you don’t want to check the points), use general polygons and set the complexity flag to kQ3GeneralPolygonShapeHintComplex (see Listing 9). Renderers look at this flag as a hint on how to process the general polygon.

GETTING FANCY

There’s nothing wrong with using only simple geometries, as described above. You can build any complex object just with triangles, but from a performance point of

Listing 8. Creating a triangle in a group

```
TQ3ColorRGB         triangleColor;
TQ3GroupObject      model;
TQ3TriangleData     triangleData;
TQ3GeometryObject   triangleObject;

static TQ3Vertex3D  vertices[3] = {{ { -1.0, -0.5, -0.25 }, NULL },
                                   { {  0.0,  0.0,  0.0  }, NULL },
                                   { { -0.5,  1.5,  0.45 }, NULL }};

triangleData.vertices[0] = vertices[0];
triangleData.vertices[1] = vertices[1];
triangleData.vertices[2] = vertices[2];
triangleData.triangleAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triangleColor, 0.8, 0.5, 0.2);
AttributeSet_AddDiffuseColor(triangleData.triangleAttributeSet,
   &triangleColor);

triangleData.vertices[0].attributeSet = Q3AttributeSet_New();
triangleData.vertices[1].attributeSet = Q3AttributeSet_New();
triangleData.vertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triangleColor, 1.0, 0.0, 0.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[0].attributeSet,
   &triangleColor);

Q3ColorRGB_Set(&triangleColor, 0.0, 1.0, 0.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[1].attributeSet,
   &triangleColor);

Q3ColorRGB_Set(&triangleColor, 0.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(triangleData.vertices[2].attributeSet,
   &triangleColor);

// Create the triangle and group.
triangleObject = Q3Triangle_New(&triangleData);
model = Q3OrderedDisplayGroup_New();
if (triangleObject != NULL) {
   Q3Group_AddObject(model, triangleObject);
   Q3Object_Dispose(triangleObject);
}

Q3Object_Dispose(triangleData.vertices[0].attributeSet);
Q3Object_Dispose(triangleData.vertices[1].attributeSet);
Q3Object_Dispose(triangleData.vertices[2].attributeSet);
Q3Object_Dispose(triangleData.triangleAttributeSet);
```

view that's not always the best thing to do. When your object is made up of faces that share vertices, it's a good idea to use a representation that allows the graphics system to reuse the vertex information (such as lighting calculations) for the shared vertices.

With a box, for example, each vertex is shared by three faces, where each face is made up of two triangles. If we draw the box as a bunch of triangles, QuickDraw 3D would have to perform the same lighting calculations on each vertex up to six times. If, on

Listing 9. Creating polygons

```
TQ3PolygonData               polygonData;
TQ3GeneralPolygonData        genPolyData;
TQ3GeometryObject            polygonObject, generalPolygonObject;
TQ3GeneralPolygonContourData contours[2];
TQ3ColorRGB                  color;

static TQ3Vertex3D  polyVertices[4] = {
                       { { -1.0,  1.0, 0.0 }, NULL },
                       { { -1.0, -1.0, 0.0 }, NULL },
                       { {  1.0, -1.0, 0.0 }, NULL },
                       { {  1.0,  1.0, 0.0 }, NULL }
                    },
                    genPolyHoleVertices[4] = {
                       { { -0.5,  0.5, 0.0 }, NULL },
                       { { -0.5, -0.5, 0.0 }, NULL },
                       { {  0.5, -0.5, 0.0 }, NULL },
                       { {  0.5,  0.5, 0.0 }, NULL }
                    };

polygonData.numVertices = 4; polygonData.vertices = polyVertices;
polygonData.polygonAttributeSet = NULL;
polygonObject = Q3Polygon_New(&polygonData);

contours[0].numVertices = 4; contours[0].vertices = polyVertices;
contours[1].numVertices = 4; contours[1].vertices = genPolyHoleVertices;
genPolyData.numContours = 2; genPolyData.contours = contours;
genPolyData.shapeHint = kQ3GeneralPolygonShapeHintComplex;
genPolyData.generalPolygonAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 1.0, 1.0, 1.0);
AttributeSet_AddDiffuseColor(genPolyData.generalPolygonAttributeSet,
   &color);

polyVertices[1].attributeSet = Q3AttributeSet_New();
polyVertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 0.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(polyVertices[1].attributeSet, &color);
Q3ColorRGB_Set(&color, 0.0, 1.0, 1.0);
AttributeSet_AddDiffuseColor(polyVertices[2].attributeSet, &color);

genPolyHoleVertices[0].attributeSet = Q3AttributeSet_New();
genPolyHoleVertices[2].attributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&color, 1.0, 0.0, 1.0);
AttributeSet_AddDiffuseColor(genPolyHoleVertices[0].attributeSet, &color);
Q3ColorRGB_Set(&color, 1.0, 1.0, 0.0);
AttributeSet_AddDiffuseColor(genPolyHoleVertices[2].attributeSet, &color);

generalPolygonObject = Q3GeneralPolygon_New(&genPolyData);
Q3Object_Dispose(genPolyData.generalPolygonAttributeSet);
Q3Object_Dispose(polyVertices[1].attributeSet);
Q3Object_Dispose(polyVertices[2].attributeSet);
Q3Object_Dispose(genPolyHoleVertices[0].attributeSet);
Q3Object_Dispose(genPolyHoleVertices[2].attributeSet);
```

the other hand, we represent the box as a box primitive or mesh object, the lighting calculations are performed only once per vertex. (However, if you attach vertex colors or face attributes, such as normals or colors, the calculations need to be performed more often.)

Here we show how to use two composite geometries — trigrid and mesh objects — as well as UV parameterization, which you may need to supply if you want to apply a texture to a trigrid or mesh.

Trigrid objects. Trigrids are a collection of triangles that share vertices. We create a trigrid in Listing 10.

Listing 10. Creating a trigrid

```
TQ3ColorRGB         triGridColor;
TQ3GroupObject      model;
TQ3TriGridData      triGridData;
TQ3GeometryObject   triGridObject;
unsigned long       numFacets, i;

static TQ3Vertex3D  vertices[12] = {{ { -1.0, -1.0,  0.0 }, NULL },
                                    ... // 10 more lines of vertex data
                                    { {  0.7,  1.0,  0.5 }, NULL }};

triGridData.numRows = 3; triGridData.numColumns = 4;
triGridData.vertices = vertices;
triGridData.triGridAttributeSet = Q3AttributeSet_New();
Q3ColorRGB_Set(&triGridColor, 0.8, 0.7, 0.3);
AttributeSet_AddDiffuseColor(triGridData.triGridAttributeSet,
   &triGridColor);

numFacets = (triGridData.numRows - 1) * (triGridData.numColumns - 1)
   * 2;
triGridData.facetAttributeSet =
   malloc(numFacets * sizeof(TQ3AttributeSet));
for (i = 0; i < numFacets; i++) {
   triGridData.facetAttributeSet[i] = NULL;
}
Q3ColorRGB_Set(&triGridColor, 1.0, 0.0, 0.5);
triGridData.facetAttributeSet[5] = Q3AttributeSet_New();
AttributeSet_AddDiffuseColor(triGridData.facetAttributeSet[5],
   &triGridColor);

triGridObject = Q3TriGrid_New(&triGridData);
```

UV parameterization. Texturing allows you to have more realistic looking models. For texturing to work, the geometry must have *UV parameters* on its vertices, which may have to be supplied by you. The UV parameters are two floating-point values (U and V) that correlate a location on the geometry to a point in the picture of the texture (see Figure 9).

The convention for QuickDraw 3D is to start the UV parameters at 0.0,0.0 at the bottom left, with U increasing toward the right and V increasing upward. You supply the UV parameterization as a collection of vertex attributes.

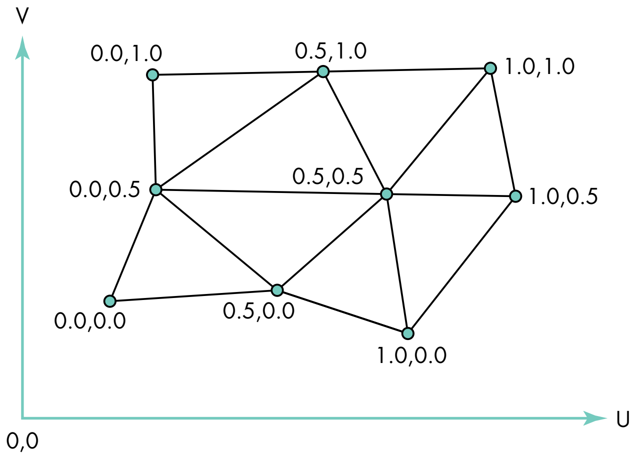


Figure 9. UV parameters on a trigrid's vertices for texture mapping

Once a UV parameterization has been applied to a surface's vertices, the surface can be texture mapped. There are several steps to texturing surfaces with QuickDraw 3D. In general, you'll already have a texture stored in a pixel map somewhere. What you need to do is create a texture shader (of type TQ3TextureObject) and add it into your display group before you add the geometry you want to shade.

Listing 11 is a general-purpose routine for adding a texture shader to a group. It's interesting for a number of reasons: it shows how to search a group for particular objects (in this case, an existing shader that it will replace), how to edit items within a group, and how to add new items.

Listing 11. Routine to texture-map an object

```
TQ3Status AddTextureToGroup(TQ3GroupObject theGroup, TQ3StoragePixmap *textureImage)
{
   TQ3TextureObject  textureObject;
   TQ3GroupPosition  position;
   TQ3Object         firstObject;

   // Create a texture object.
   textureObject = Q3PixmapTexture_New(textureImage);
   if (textureObject) {
      if (Q3Object_IsType(theGroup, kQ3GroupTypeDisplay) == kQ3True) {
         // If the group is a display group...
         Q3Group_GetFirstPosition(theGroup, &position);
         Q3Group_GetPositionObject(theGroup, position, &firstObject);
         if (Q3Object_IsType(firstObject, kQ3SurfaceShaderTypeTexture) == kQ3True) {
            TQ3TextureObject  oldTextureObject;
            TQ3StoragePixmap  oldTextureImage;
            // Replace existing texture by new one.
            Q3TextureShader_GetTexture(firstObject, &oldTextureObject);
            Q3PixmapTexture_GetPixmap(oldTextureObject, &oldTextureImage);
            Q3Object_Dispose(oldTextureObject);
            Q3TextureShader_SetTexture(firstObject, textureObject);
            Q3Object_Dispose(textureObject);
```

(continued on next page)

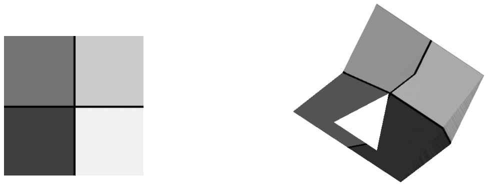
Listing 11. Routine to texture-map an object *(continued)*

```
         } else {
            TQ3ShaderObject   textureShader;
            // Create texture shader and add it to group.
            textureShader = Q3TextureShader_New(textureObject);
            if (textureShader) {
               Q3Object_Dispose(textureObject);
               Q3Group_AddObjectBefore(theGroup, position, textureShader);
               Q3Object_Dispose(textureShader);
            } else
               return (kQ3Failure);
         }
         Q3Object_Dispose(firstObject);
      } else if (Q3Object_IsType(theGroup, kQ3DisplayGroupTypeOrdered) == kQ3True) {
         // If the group is an ordered display group...
         TQ3ShaderObject   textureShader;
         Q3Group_GetFirstPositionOfType(theGroup, kQ3ShapeTypeShader, &position);
         if (position) {
            Q3Group_GetPositionObject(theGroup, position, &firstObject);
            if (Q3Object_IsType(firstObject, kQ3SurfaceShaderTypeTexture) == kQ3True) {
               TQ3TextureObject  oldTextureObject;
               TQ3StoragePixmap  oldTextureImage;
               // Replace existing texture by new one.
               Q3TextureShader_GetTexture(firstObject, &oldTextureObject);
               Q3PixmapTexture_GetPixmap(oldTextureObject, &oldTextureImage);
               Q3Object_Dispose(oldTextureObject);
               Q3TextureShader_SetTexture(firstObject, textureObject);
               Q3Object_Dispose(textureObject);
            } else {
               // Create texture shader and add it to group.
               textureShader = Q3TextureShader_New(textureObject);
               if (textureShader) {
                  Q3Object_Dispose(textureObject);
                  Q3Group_SetPositionObject(theGroup, position, textureShader);
                  Q3Object_Dispose(textureShader);
               } else
                  return (kQ3Failure);
            }
         } else {
            // Create texture shader and add it to group.
            textureShader = Q3TextureShader_New(textureObject);
            if (textureShader) {
               Q3Object_Dispose(textureObject);
               Q3Group_AddObject(theGroup, textureShader);
               Q3Object_Dispose(textureShader);
            } else
               return (kQ3Failure);
         }
      }
   return (kQ3Success);
   } else         // If pixmap shader not created...
      return (kQ3Failure);
}
```

Mesh objects. Listing 12 shows the key components needed to create a simple mesh geometry. We create a mesh consisting of two faces, with one of them having a hole. We also add UV parameters to the vertices so that we can texture-map the mesh. Figure 10 shows the texture map and the resulting textured mesh.

Listing 12. Creating a mesh

```
TQ3GroupObject BuildMesh(void)
{
   TQ3ColorRGB         meshColor;
   TQ3GroupObject      model;
   TQ3Vertex3D         vertices[9] = {
      { { -0.5,  0.5,  0.0 }, NULL },    { { -0.5, -0.5,  0.0 }, NULL },
      { {  0.0, -0.5,  0.3 }, NULL },    { {  0.5, -0.5,  0.0 }, NULL },
      { {  0.5,  0.5,  0.0 }, NULL },    { {  0.0,  0.5,  0.3 }, NULL },
      { { -0.4,  0.2,  0.0 }, NULL },    { {  0.0,  0.0,  0.0 }, NULL }
   };
   TQ3Param2D          verticesUV[9] = {
      { 0.0, 1.0 }, { 0.0, 0.0 }, { 0.5, 0.0 },
      { 1.0, 0.0 }, { 1.0, 1.0 }, { 0.5, 1.0 },
      { 0.1, 0.8 }, { 0.5, 0.5 }, { 0.1, 0.4 }
   };
   TQ3MeshVertex       meshVertices[9];
   TQ3GeometryObject   meshObject;
   TQ3MeshFace         meshFace;
   TQ3AttributeSet     faceAttributes;
   unsigned long       i;

   meshObject = Q3Mesh_New();
   Q3Mesh_DelayUpdates(meshObject);
   for (i = 0; i < 9; i++) {
      TQ3AttributeSet  vertexASet;
      meshVertices[i] = Q3Mesh_VertexNew(meshObject, &vertices[i]);
      vertexASet = Q3AttributeSet_New();
      AttributeSet_AddSurfaceUV(vertexASet, &verticesUV[i]);
      Q3Mesh_SetVertexAttributeSet(meshObject, meshVertices[i],
         vertexASet);
      Q3Object_Dispose(vertexASet);
   }
   faceAttributes = Q3AttributeSet_New();
   Q3ColorRGB_Set(&meshColor, 0.3, 0.9, 0.5);
   AttributeSet_AddDiffuseColor(faceAttributes, &meshColor);
   meshFace = Q3Mesh_FaceNew(meshObject, 6, meshVertices,
      faceAttributes);
   Q3Mesh_FaceToContour(meshObject, meshFace, Q3Mesh_FaceNew(meshObject,
      3, &meshVertices[6], NULL));
   Q3Mesh_ResumeUpdates(meshObject);
   model = Q3OrderedDisplayGroup_New();
   Q3Group_AddObject(model, meshObject);
   Q3Object_Dispose(faceAttributes);
   Q3Object_Dispose(meshObject);
   return (model);
}
```



Texture map

Mesh with texture map applied

Figure 10. Texture map applied to a mesh

Q3Mesh_DelayUpdates and Q3Mesh_ResumeUpdates, used in Listing 12, are two very important routines. Mesh objects can often contain hundreds and even thousands of vertices. When you're building a complex model, we advise that you turn off updates to the internal ordering of the mesh data, so that building the mesh takes as little time as possible. The difference between doing this and not doing this can be, in the case of a complex model containing 3000 polygons, several minutes when Q3Mesh_DelayUpdates is not called, compared with 3 seconds when it is called (on a mid-level computer).

WHAT DO YOU WANT TO BUILD TODAY?

We hope that the hints in this article will save you some time and help you in your development process. We've been pleasantly surprised by some of the applications in which developers have been putting QuickDraw 3D to use; for example, a European developer used QuickDraw 3D to produce 3D representations of his code profiler application's data. Learning the basics of QuickDraw 3D's geometries is the first step toward mining the rich seam of functionality that QuickDraw 3D offers.

Thanks to our technical reviewers Tom Dowdy, Tim Monroe, and Philip Schneider.•

DAVE EVANS

BALANCE OF POWER

Power Macintosh: The Next Generation

The Power Macintosh computer just keeps moving forward. The latest generation brings greatly improved performance and adds the PCI expansion bus and the PowerPC 603 and 604 processors. Software changes that improve performance include the following:

- an improved 680x0 emulator
- a native Resource Manager
- native networking (Open Transport)
- native device drivers
- an improved Memory Manager

I'll describe these new features and discuss how you can maintain compatibility with the new Power Macintosh computers and with future changes to the Mac OS.

THE IMPROVED EMULATOR

First delivered with the Power Macintosh 9500 computer, the new emulator improves on the original in one key way: it actually recompiles 680x0 code into native PowerPC code. Since large portions of the Mac OS are still in 680x0 code, this new emulator speeds up most common operations and offers significant improvements for 680x0 code with tight loops.

Recompiling doesn't mean converting 680x0 instructions one for one into PowerPC instructions. Fully emulating a 680x0 instruction still takes a few PowerPC instructions. But recompiled code is more efficient and optimized. The original emulator had to decipher each instruction every time it was executed, but recompiled code from the new emulator is analyzed once and then executed many times.

Because it takes extra time to recompile code, the emulator doesn't immediately translate all 680x0 code. It operates just like its predecessor until it encounters a loop or similar repetition. Then, instead of emulating the same code repeatedly, it translates the instructions into native code and caches the result. Subsequent calls to that code simply execute the native translation, greatly improving performance.

The cache of translated 680x0 code must stay coherent with memory, much like the caches on the Motorola 68040 processor. Therefore, whenever your software modifies code or changes application jump tables, you should flush the instruction cache. (See the Macintosh Technical Note "Cache as Cache Can" (HW 6) for a more detailed description of cases where flushing the instruction cache is necessary.) In the past you could call Gestalt and check the processor type to flush only on a 68040. Since the new emulator supports only the 68020 instruction set — and Gestalt will indicate that a 68020 is installed — you should now flush any time you modify code or change jump tables.

The best way to flush 680x0 code in the cache is with FlushCodeCacheRange, which flushes only the invalid portion of the emulator's cache. FlushInstructionCache also works but can degrade performance by wastefully purging recompiled code that's still valid. These routines are documented in *Inside Macintosh: Memory*. The C prototype for FlushCodeCacheRange is as follows:

```
OSErr FlushCodeCacheRange(void *address,
   unsigned long count);
```

In 680x0 assembly, you would use

```
MyFlushCodeCacheRange Proc
   ; On entry A0 = address, D0 = # of bytes
   ; Trashes A0, A1, D0. Result in D0, Z bit set.
   ;
   movea.l     D0,A1     ; # bytes in A1
   moveq       #$9,D0    ; selector
   _HWPriv               ; A098
   tst.w       D0        ; result == noErr
   rts
```

OTHER SOFTWARE CHANGES

The first Power Macintosh computer ported critical portions of the Macintosh Toolbox to native PowerPC

DAVE EVANS and fellow Apple engineer Rus Maxham rode 2000 miles on their motorcycles this summer. They journeyed through the lush Central Valley of California, the blistering heat of the southern Arizona deserts, and the neon glitz of Las Vegas. Along the way they enjoyed the camaraderie of fellow bikers and were rescued in their hour of need by a sympathetic motorcycling couple who housed them as Rus rebuilt his BMW's rear drive assembly.•

code. Ultimately we'll take all of the Mac OS native, but for now we've focused on areas that most increase overall performance. So, starting with the Power Macintosh 9500, we've added a native Resource Manager, the native Open Transport networking stack, and native device drivers. I'll discuss each of these in turn and then mention improvements to the Modern Memory Manager.

Even though many calls to the Resource Manager are bound by I/O bottlenecks, porting the Resource Manager to native PowerPC code still substantially improves performance. Often to complete a request the Resource Manager need only look up existing information and return it, and even if file I/O is required the data is often in the system disk cache. For these reasons, many Resource Manager calls will execute much faster on the new machines.

Native Open Transport networking provides a stream-based interface for networking that's independent of the network protocol. You can now implement a variety of network solutions without concerning yourself with protocol details. Documentation on Open Transport is provided on this issue's CD.

Native device drivers provide both a performance improvement and an improved system programming interface (SPI). This SPI is available with all PCI-based Macintosh computers, starting with the Power Macintosh 9500. For more information on these drivers, see the article "Creating PCI Device Drivers" in *develop* Issue 22 and *Designing PCI Cards and Drivers for Power Macintosh Computers*, available from APDA.

Although not new, the native Modern Memory Manager has been improved in two important ways:

- Many of the routines are now implemented as "fat" binaries instead of all native code. When your 680x0 code calls the Memory Manager, it will now execute 680x0-based routines, eliminating the Mixed Mode environment switch once needed to call the native routines. Reducing the number of these switches can measurably improve performance.
- The bus error handlers have been removed, significantly increasing the performance of many of the simple Memory Manager calls and allowing a number of the calls to be made into fat traps. Bugs discovered during the process of removing the handlers have been fixed.

Handles passed to the Memory Manager now go through a rigorous check before they can affect other Memory Manager data structures; however, without the nearly foolproof bus error handling, it's a little more likely that you'll pass an invalid address and crash. If you crash in the MemoryMgr code fragment while testing on the new Power Macintosh computers, you probably passed an invalid pointer or handle. You can use the Debugging Modern Memory Manager to aggressively catch these application errors.

Note also that the bus error handlers would allow system (and even application) heaps to become corrupted, deteriorating the overall user experience without causing the machine to crash. This is much less likely to happen now, but if structures do get corrupted other than by the Memory Manager, a system crash will result.

Also available starting with the latest Power Macintosh machines is support for very large hard disk volumes. In the past, only 2-gigabyte volumes were allowed; then with System 7.5 we relaxed that restriction to 4-gigabyte volumes. But many of you were still hungry for more, so now we allow up to 2 terabytes (that's 2000 gigabytes) of file system address space per volume. Unless you're developing utilities and drivers compatible with the new volume sizes, though, you really don't need to pay attention to the new large-volume support, because the API remains unchanged. The only time an application might want to take advantage of the new support is when it wants to know before attempting to save to disk whether there's enough free space on the volume. Even in this case, the application won't be able to save a file bigger than the existing limit of 2 GB, and the old version of GetVInfo will return values that are "high-water marked" at 2 GB for compatibility reasons, even if more space is available.

If you really do want to know how much space is available, you can do so through an extension to the File Manager API. We extended the API because the existing 32-bit size information was too small to address volumes and files larger than 4 GB. You'll use the following new routine to get 64-bit sizes:

```
pascal OSErr PBXGetVolInfo(XVolumeParam
                paramBlock, Boolean async);
```

This routine takes an extended VolumeParam structure, named XVolumeParam, which you'll find declared in an updated Files.h interface file on the CD. Before using this routine, be sure to call Gestalt with the gestaltFSAttr selector; if the response parameter has the gestaltFSSupports2TBVolumes bit set, the new routine is available. Note that there are also extended Read and Write calls for drivers that want to support volumes larger than 4 GB.

PCI AND NUBUS

Starting with the PCI-based Power Macintosh computers, support for the NuBus™-specific Slot Manager goes away. Some applications used to call the Slot Manager directly to get video and other device information. This will no longer work, so we've provided better methods: the Display Manager API has been extended for all the video device information you'll need, and the new Name Registry API will give you device information independent of the specific expansion bus implementation.

One example of the improved Display Manager API is the way you get display modes for video devices. With the Slot Manager this took a lot of code, but the Display Manager gives you one encompassing routine:

```
pascal OSErr DMNewDisplayModeList(
                GDHandle theGDevice,
                unsigned long reserved,
                unsigned long *modeCount,
                DMListType *theDisplayModeList,
                unsigned long modeListFlags);
```

With this and other new Display Manager routines, you can avoid the Slot Manager altogether when gathering display information. But if you must access other device information, you can use the bus-neutral Name Registry, which manages a tree of device objects that you can access as a linked list. Look for the new header files (Displays.h and NameRegistry.h) on this issue's CD.

MAINTAINING COMPATIBILITY

As Apple improves the Mac OS, compatibility with the documented APIs and SPIs is ensured — but don't assume that if your application runs fine on existing machines, it will continue to do so in the future. We can't ensure complete compatibility if application code makes invalid assumptions or uses unsupported parts of the Mac OS. There are some things you can do to help ensure that your applications will run on future versions of the Mac OS.

First, use only the officially documented APIs. For example, don't assume that the Z status bit will be set correctly on exit from a trap unless it's documented. As we take more traps native, the 680x0 status register becomes irrelevant and such checks break. Here's an example of 680x0 code that now breaks because it assumes the Z status bit will be set by Get1Resource:

```
move.l   #'DAVE',-(sp)
clr.w    -(sp)
_Get1Resource
beq.s    error       ; BAD!
```

You also shouldn't expect results in registers if the trap isn't documented to return them there. It's true that some traps used to accidentally exit with useful data in register D0 or A0, but if that's not documented as part of the API it won't be supported in the future.

Second, test your software using EvenBetterBusError, the Debugging Modern Memory Manager, and any other debugging tools that are appropriate (look in the Testing & Debugging folder on the CD). Stress-testing your software with these tools will catch many errors that otherwise would go unnoticed. EvenBetterBusError catches most stray references to nil, such as writing to location 0 or using nil pointers and handles. The Debugging Modern Memory Manager catches those occasions when you damage a heap or pass invalid addresses.

Finally, as I've said in previous columns, don't use RS/6000 POWER instructions in your native code. Although the PowerPC 601 processor supports many of them, the new 603 and 604 processors do not. We've made an attempt to emulate the POWER instructions in software for these new processors, but this emulation is very expensive. When a 603 or 604 encounters one of these now-illegal instructions, it stops everything and calls our new illegal-instruction handler, which recognizes the instruction that was used and attempts to use a valid one instead. This operation is very time consuming; if your performance-critical code includes POWER instructions, you'll see a severe slowdown. As described in this column in *develop* Issue 21, you should use the DumpXCOFF tool to check your code for any POWER instructions.

NEW DIRECTIONS

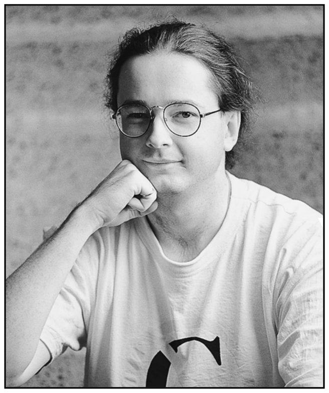
Apple will continue to take advantage of RISC technology and will both improve existing performance and add new functionality. Make sure your code uses documented interfaces so that it will stay compatible and run on future generations of the Power Macintosh. And be sure to check out Open Transport and PCI device drivers — they're exciting new directions that will take you closer to the next generation of the Mac OS today.

Thanks to Bill Knott, Eric Traut, and Jack Valois for reviewing this column.•

Special thanks to Randy and Peggy Marlatt of Flagstaff, Arizona, for road support.•

Implementing Shared Internet Preferences With Internet Config

Having to enter the same Internet preferences, such as e-mail address and news server, into multiple applications is bothersome not just for users, but also for developers who must create the user interface associated with them. The Internet Configuration System (IC) provides a simple user application for setting preferences, and an API for getting the preferences from a database that's shared by all applications. It's easy to add IC support to your application and take advantage of the flexibility gained by IC's use of the Component Manager — a valuable technique in itself.



QUINN "THE ESKIMO!"

Preferences, like nuclear weapons, proliferate. At times it seems that the major developers are engaged in a "preferences race," where each one tries to gain the upper hand by adding a dozen new preferences in each new release. Like the arms race, the preferences race is obviously counterproductive, even dangerous, and yet no one knows how to stop it.

Some of the worst offenders are Internet-related applications. How many times have you had to enter your e-mail address into a configuration window? And what about your preferred type and creator for JPEG files? Doesn't this just seem like a waste of your time? The Internet Configuration System, or Internet Config for short, spares everyone this trouble. And it spares developers the complexities of implementing these preferences in each application.

This article takes you inside Internet Config. Take a good look at the design: IC implements its shared library as a component, and uses switch glue to provide a default implementation if the component is absent. Using the Component Manager to implement shared libraries is a helpful technique not just for IC, but for other APIs as well. Note too that Internet Config is useful for more than its name implies. For example, the extension-to-file-type mapping database is useful for any program that deals with "foreign" file systems. Indeed, IC is a perfectly valid mechanism for storing private preferences that have nothing to do with the Internet.

Although IC is intended as an abstract API, all its source code is placed in the public domain — a condition of its development. This lets me illustrate the text with

QUINN "THE ESKIMO!" (quinn@cs.uwa.edu.au) has a first name but, when asked about it, his usual response is "I could tell you but then I'd have to kill you!" He programs for a living with the Department of Computer Science at the University of Western Australia, but on weekends he gets together with Peter N. Lewis and programs for *fun*. The Internet Configuration System is a product of these misspent recreational hours. Quinn writes in Pascal using a Dvorak keyboard on a Macintosh Duo that he carries around on his bicycle, and he's still trying to figure out how to use this minority status to his economic advantage.•

snippets from the actual implementation and gives you full access to the source code. Both the IC user's kit and the IC developer's kit, which contain code and documentation, are included on this issue's CD. Note that Internet Config was developed independently and is not supported by Apple.

The latest versions of the kits are always available from the ftp sites ftp://ftp.share.com/pub/internet-configuration/ and ftp://redback.cs.uwa.edu.au/Others/Quinn/Config/. In addition, the user kit is available from UMich and Info-Mac mirrors around the world.•

As with any new piece of software intended to be widely adopted, Internet Config needs developer support in order to be successful. I hope this article raises the awareness of IC in the developer community and prompts some of you to support it.

INTERNET CONFIG FROM THE OUTSIDE

Before going inside Internet Config, it's important to know how the system works as a whole. The best way to do this is to get a copy of the Internet Config application and run it (there's a copy on this issue's CD), but if you're too relaxed to do that right now, keep reading for a description of the basics. We'll look at IC first from the user's perspective and then from the programmer's point of view.

THE USER'S PERSPECTIVE

To the user, Internet Config is a proper Macintosh application. It supports the standard menu commands New, Open, Save, Save As, and so on. The only difference is that the files it operates on are preferences files. Figure 1 shows Internet Config and its related files.

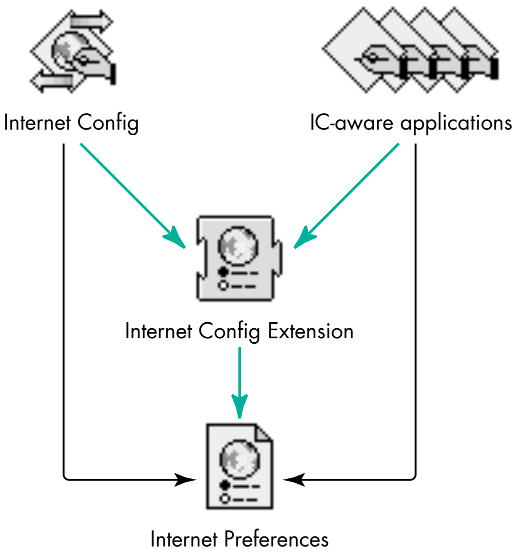


Figure 1. Internet Config and its related files — what the user sees

The first time the Internet Config application is run, it installs the Internet Config Extension into the Extensions folder and creates a new, blank Internet Preferences file in the Preferences folder. It then displays the main window, shown in Figure 2, which allows the user to edit the preferences.

Each of the buttons in the main window displays another window containing a group of related preferences. For example, the Personal button brings up the window shown

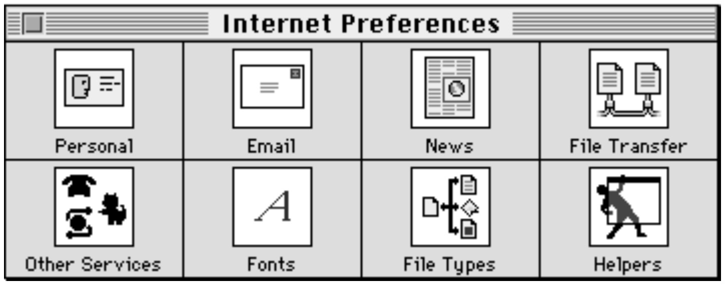
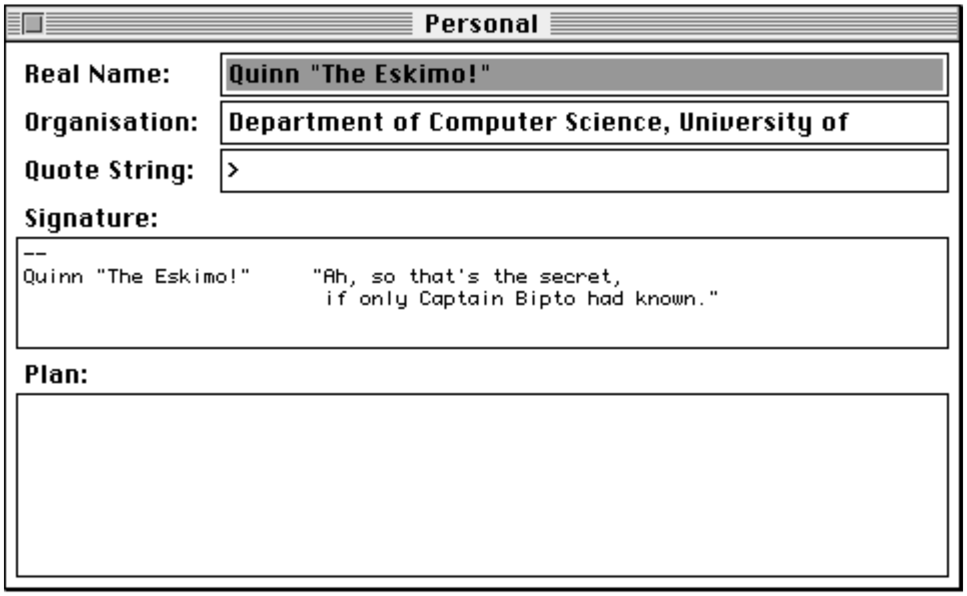


Figure 2. The Internet Config application's main window



Real Name: Quinn "The Eskimo!"

Organisation: Department of Computer Science, University of

Quote String: >

Signature:

```
--
Quinn "The Eskimo!"     "Ah, so that's the secret,
                          if only Captain Bipto had known."
```

Plan:

Figure 3. The Personal preferences window

in Figure 3. The user enters preferences into each of these windows and then quits and saves the preferences.

From this point on, the user never has to enter those preferences again. Any IC-aware program the user runs simply accesses the preferred settings without requiring them to be reentered. This makes the user very happy (we presume).

Users can even run IC-aware applications "out of the box" — they don't have to run Internet Config first. If the Internet Config Extension isn't installed, IC-aware client applications access the Internet Preferences file directly instead of through the extension (as shown by the black arrows in Figure 1). The way this is done is described later in the section "The Inner Workings of an API Routine."

THE PROGRAMMER'S PERSPECTIVE

To programmers, Internet Config consists of a set of interface files that define the API, and a library to be statically linked to their programs. IC can be used from all of the common Macintosh development environments: MPW, THINK, and Metrowerks; Pascal and C; and 680x0 and PowerPC. The examples in this article, like IC itself, were written in THINK Pascal.

What's in an IC preference. Before getting to the details of the API, you need to know more about IC preferences. In IC, a *preference* is an item of information that's useful to the client application program. Each preference has three components: its key, its data, and its attributes.

- The *key* is a Str255 that identifies the preference. You can use the key to fetch the data and attributes.
- The *data* is an untyped sequence of bytes that represents the value of the preference. The data's structure is determined by the client program. The structures of the common preferences are defined in the IC programming documentation.
- The *attributes* represent information about the preference that's supplementary to the preference data, such as whether the preference is read/write or read-only.

In the e-mail address preference, for example, the key is the string "Email". If you pass this string into IC, it returns the preference's data and attributes. By convention, the data for the key "Email" is interpreted as a Pascal string containing the user's preferred e-mail address.

IC's core API routines. Internet Config has the following core API routines. Although the API has a lot more depth, these four routines are all you need to program with IC.

```
FUNCTION ICStart (VAR inst: ICInstance; creator: OSType): ICError;

FUNCTION ICStop (inst: ICInstance): ICError;

FUNCTION ICFindConfigFile (inst: ICInstance; count: Integer;
                         folders: ICDirSpecArrayPtr): ICError;

FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr;
                         buf: Ptr; VAR size: LongInt): ICError;
```

The ICStart routine is always called first. Here you pass in your application's creator code so that future versions of IC can support application-dependent preferences. ICStart returns a value of type ICInstance; this is an opaque type that must be passed to every other API call. ICStop is called at the termination of your application to dispose of the ICInstance you obtained with ICStart.

ICFindConfigFile is called immediately after ICStart. IC uses this routine to support applications with double-clickable user configuration files, a common phenomenon among Internet applications. If you need to support these files, see the IC programming documentation; otherwise, just pass in 0 for the count parameter and nil for the folders parameter.

The ICGetPref routine takes a preference key and returns the preference's attributes in **attr** and its data in the buffer pointed to by **buf**. The maximum size of the buffer is passed in as **size**, which is adjusted to the actual number of bytes of preference data.

The simplest example. The program in Listing 1 demonstrates the simplest possible use of IC technology. All it does is write the user's e-mail address to the standard output. This program calls the four core API routines: it begins by calling ICStart and terminates with an ICStop call; it calls ICFindConfigFile with the default parameters and uses ICGetPref to fetch the value of a specific preference — in this case the user's e-mail address.

Listing 1. The simplest IC-aware program

```
PROGRAM ICEmailAddress;
   { The simplest IC-aware program. It simply outputs the user's }
   { preferred e-mail address. }

   USES
      ICTypes, ICAPI, ICKeys;    { standard IC interfaces }

   VAR
      instance:   ICInstance; { opaque reference to IC session }
      str:        Str255;     { buffer to read e-mail address into }
      str_size:   LongInt;    { size of above buffer }
      junk:       ICError;    { place to throw away error results }
      junk_attr:  ICAttr;     { place to throw away attributes }
BEGIN
   { Start IC. }
   IF ICStart(instance, '????') = noErr THEN BEGIN
      { Specify a database, in this case the default one. }
      IF ICFindConfigFile(instance, 0, NIL) = noErr THEN BEGIN
         { Read the real name preferences. }
         str_size := sizeof(str);   { 256 bytes -- a similar construct }
                                    { wouldn't work in C }
         IF ICGetPref(instance, kICEmail, junk_attr, @str, str_size)
                  = noErr THEN BEGIN
            writeln(str);
         END;  { IF }
      END;  { IF }
      { Shut down IC. }
      junk := ICStop(instance);
   END;  { IF }
END.  { ICEmailAddress }
```

INSIDE INTERNET CONFIG

The IC API just described is really all you need to know to make your program IC-aware; now we'll get into the guts of Internet Config to see how it achieves its magic. We'll look first at its underlying design and then at how its internal structures work together.

THE IC DESIGN: A SIMPLE, EXPANDABLE SYSTEM

The design requirements for Internet Config evolved during early discussions of what an Internet configuration system might look like (see "How Internet Config Came to Be"). These requirements guided the development process and form the basic structure of Internet Config — an efficient, expandable system that's easy to use and easy to support.

Internet Config can accept sweeping changes while maintaining API compatibility, and it allows for patches to support future extensions and bug fixes. We couldn't achieve such expandability with a simple shared preferences implementation, and the consequent loss of simplicity caused a lot of debate during the development process.

The need for simplicity was implicit from the beginning. To add support for Internet Config, application developers have to revise their code. Developers tend to be lazy

HOW INTERNET CONFIG CAME TO BE

Designing Internet Config was a complicated business. The process began in March 1994 with a discussion on the Usenet newsgroup comp.sys.mac.comm. Many people thought simplifying Internet configuration was a good idea, but few agreed how best to achieve the goal, or indeed what the goal was.

We set up a mailing list to swap ideas, and discussion continued apace for weeks. One of the biggest issues was the disparity between the problems we wanted to solve and the ones we *could* solve given our limited resources.

After a week or two of thrashing out the requirements, Peter N. Lewis, Marcus Jager, and I proposed the first API. A few weeks later we shipped the first implementation of the Internet Config Extension.

The problem IC solves is actually quite simple, so it didn't take long to implement the design. As usual, however, it took some time to go from a working implementation to a final product — we shipped Internet Config 1.0 in December 1994. Though we've made minor additions and changes, the initial design survives to this day.

— hey, I mean that as a compliment — and generally prefer simple systems to complicated ones. Developer support is critical for success, so we kept the system simple. Still, it isn't so simple as to compromise the need for expandability.

As we've already seen, IC has several other interesting design features. The API supports applications with double-clickable user configuration files. The Internet Config user application accesses all the Internet preferences through the API, and is thereby isolated from the implementation details. IC-aware applications work even if the Internet Config Extension isn't installed. We even included support for System 6 (much as we resented it).

IC'S INTERNAL STRUCTURES

As you can see in Figure 4, the Internet Config application and IC-aware client programs have very similar internal structures. In fact, except for a few artifacts

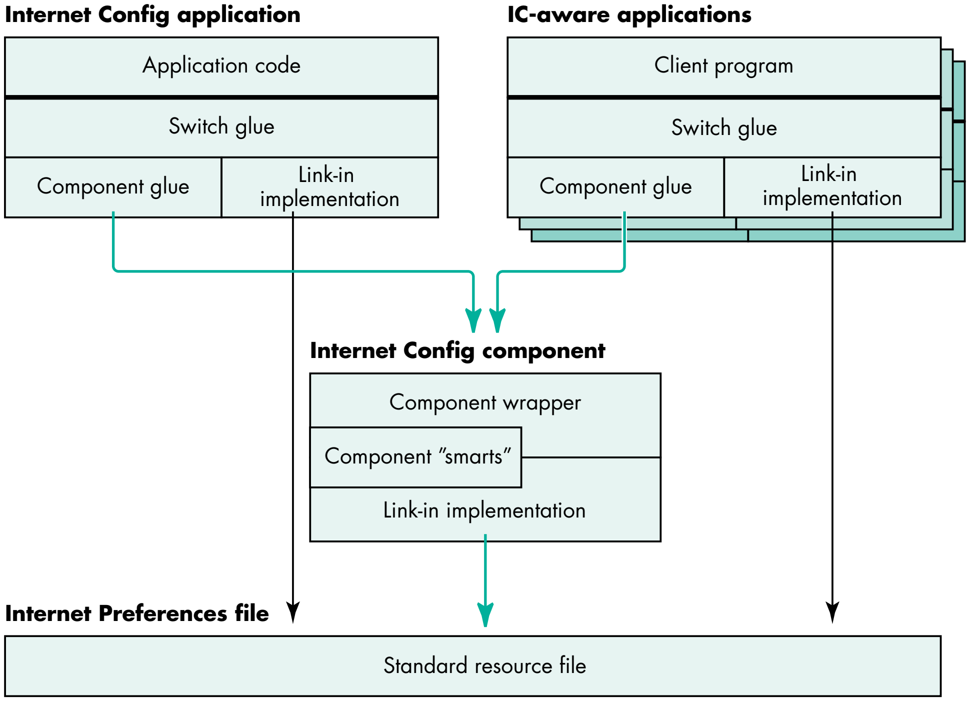


Figure 4. Inside the Internet Config entities — what the programmer sees

caused by implementing “safe saving,” the Internet Config application uses the standard API to modify the Internet Preferences file. The Internet Config component, which the user sees as the Internet Config Extension, is basically a shared library of routines implemented as a component (see “The IC Component and Shared Libraries on the Macintosh”).

The switch glue is a common interface that applications use to call IC. This glue decides whether the Internet Config component is available and, if it is, routes all calls through to it. If the component isn’t present, the calls are routed through to the link-in implementation, which then does the work.

This switching mechanism satisfies two design requirements. It allows the API to be patched by replacing or overriding the Internet Config component. It also allows IC-aware programs to work even if the component isn’t installed; they simply fall back to using the link-in implementation.

THE INNER WORKINGS OF AN API ROUTINE

Now we’ll look more closely at how the Start and GetPref routines are implemented in each part of the Internet Config system. We’ll trace these two calls from the top level, where they’re called by the client program, all the way down to the link-in implementation, where the real action takes place.

This section is quite technical; if you’re not interested in the implementation details, you might want to just skim through it. Many of the details are provided for illustrative purposes only. Take heed! *If you write client programs that rely on these details, they will break in future revisions of IC.* The public interface to IC is defined in the IC programming documentation.

We’ll start with the switch glue and proceed through the standard call path. On the way we’ll examine the component glue, wrapper, and “smarts,” and finally, the link-in implementation. The path is convoluted but rewards you with both data and code abstraction.

Start and GetPref appear in each part of the system, and each appearance has a specific purpose, as we’ll see in a moment. To keep things straight, various instances

THE IC COMPONENT AND SHARED LIBRARIES ON THE MACINTOSH

The Internet Config component is essentially a shared library of routines. So why implement it as a component? The answer lies in the confused state of shared libraries on the Macintosh.

When we started writing IC we knew we’d need a shared library. The problem was not that the system didn’t have a shared library mechanism, but that it had too many. At the time there were four Apple shared library solutions, each with its unique drawbacks: the Component Manager wasn’t a “real” shared library system; the Apple Shared Library Manager (ASLM) had limited availability and lacked PowerPC support and developer tools; the Code Fragment Manager (CFM) lacked 680x0 support; and the System Object Model (SOM) lacked any availability.

These days life is a little better. ASLM now works on the PowerPC platform, CFM is being ported to the 680x0 platform, SOM is imminent, and Apple has issued a clear statement of direction on shared libraries, centered on CFM.

But statements of direction don’t solve problems — they just clear up confusion. The shared library problem persists. When I was writing this article someone asked me for advice about which shared library mechanism to use. My recommendation today is the same as at the start of the IC project: use the Component Manager. It’s still the only solution that has the developer tools, has 680x0 and PowerPC support, and is already installed on most users’ machines.

of the same routine are prefixed to denote which part of the system they're in. The prefixes are listed in Table 1, which shows the various specifications for the GetPref routine as an example. (Note that these specifications vary only in the name's prefix and the type of the first parameter. The "R" in the ICR prefix indicates that these routines actually use the Resource Manager to modify the preferences; all the other routines are glue.)

Table 1. Routine name prefixes

Prefix	Part of System	First Parameter	GetPref Specification
IC	Standard API (switch glue)	ICInstance	FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICC	Component API (component glue)	ComponentInstance	FUNCTION ICCGetPref (inst: ComponentInstance; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICCI	Component internal	globalsHandle	FUNCTION ICCIGetPref (inst: globalsHandle; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;
ICR	Link-in implementation	VAR ICRRecord	FUNCTION ICRGetPref (var inst: ICRRecord; key: Str255; VAR attr: ICAttr; buf: Ptr; VAR size: LongInt): ICError;

THE SWITCH GLUE

The switch glue relies on ICRRecord, the central data structure of IC, shown in Listing 2. The first field of ICRRecord, **instance**, is a ComponentInstance, which normally holds the connection to the Internet Config component. If the component is installed, the instance field holds the connection to it; the rest of the fields are ignored because the component has a separate ICRRecord in its global variables. If the component isn't installed, the instance field is nil, and the link-in implementation uses the rest of the fields to hold the necessary state (as we'll see later).

Listing 2. ICRRecord

```
TYPE
   ICRRecord = RECORD
      { This entire record is completely private to the }
      { implementation!!! Your code will break if you depend }
      { on the details here. You have been warned. }
      instance: ComponentInstance;
               { nil if no component available; if not nil, }
               { then rest of record is junk }
      ...   { other fields to be discussed later }
      END;
   ICRRecordPtr = ^ICRRecord;
```

The switch glue for the application's Start routine, ICStart, is shown in Listing 3. The first thing ICStart does is attempt to allocate an ICRRecord; if it succeeds, it then tries to open a connection to the component with the component glue routine ICCStart. ICCStart either succeeds, setting the internal instance field to the connection to the component, or fails and returns an error. If ICCStart returns an error, ICStart falls back to using the link-in implementation by calling ICRStart. If ICRStart fails, Internet Config fails to start up; ICStart sets **inst** to nil and returns an error.

Listing 3. The switch glue for Start

```
FUNCTION ICStart (VAR inst: ICInstance; creator: OSType): ICError;
   VAR
      err:  ICError;
BEGIN
   inst := NewPtr(sizeof(ICRRecord));
   err := MemError;
   IF err = noErr THEN BEGIN
      err := ICCStart(ICRRecordPtr(inst)^.instance, creator);
      IF err <> noErr THEN BEGIN
         err := ICRStart(ICRRecordPtr(inst)^, creator);
      END;  { IF }
      IF err <> noErr THEN BEGIN
         DisposePtr(inst);
         inst := NIL;
      END;  { IF }
   END;  { IF }
   ICStart := err;
END;  { ICStart }
```

Listing 4. The switch glue for GetPref

```
FUNCTION ICGetPref (inst: ICInstance; key: Str255; VAR attr: ICAttr;
                          buf: Ptr; VAR size: LongInt): ICError;
BEGIN
   IF ICRRecordPtr(inst)^.instance <> NIL THEN BEGIN
      ICGetPref := ICCGetPref(ICRRecordPtr(inst)^.instance,
                                key, attr, buf, size);
   END
   ELSE BEGIN
      ICGetPref := ICRGetPref(ICRRecordPtr(inst)^, key, attr, buf, size);
   END;  { IF }
END;  { ICGetPref }
```

The switch glue for GetPref, and all the other API routines for that matter, is very simple. All it does is consult the internal instance field to determine whether ICStart successfully connected to the component. If so, it calls through to the component glue routine ICCGetPref; otherwise, it calls through to the link-in implementation routine ICRGetPref. This is shown in Listing 4.

The switch glue implementations of both Start and GetPref do a lot of casting between ICInstance and ICRRecordPtr, because the ICRRecordPtr type describes details of the implementation that shouldn't "leak out" to the client's view of IC. The client programs know only of ICInstance, which is an opaque type. The explicit casts could have been avoided with some preprocessor tricks, but we decided to include them longhand for clarity.

THE COMPONENT GLUE

The component glue calls the Internet Config component. In the component glue for the Start routine, shown in Listing 5, Internet Config attempts to connect to the IC component by calling the Component Manager routine OpenDefaultComponent.

Listing 5. The component glue for Start

```
FUNCTION ICCStartComponent (inst: ComponentInstance; creator: OSType):
                           ICError;
INLINE              { standard Component Manager glue }
   $2F3C, $04, $0,  { move.l    #$0004_0000,-(sp) }
   $7000,           { moveq.l   #0,d0 }
   $A82A;           { _ComponentDispatch }

FUNCTION ICCStart (VAR inst: ComponentInstance; creator: OSType):
                           ICError;
   VAR
      err, junk:  ICError;
      response:   LongInt;
BEGIN
   inst := NIL;
   IF Gestalt(gestaltComponentMgr, response) = noErr THEN BEGIN
      inst := OpenDefaultComponent(internetConfigurationComponentType,
                              internetConfigurationComponentSubType);
   END;  { IF }
   IF inst = NIL THEN BEGIN
      err := badComponentInstance;
   END
   ELSE BEGIN
      err := ICCStartComponent(inst, creator);
      IF err <> noErr THEN BEGIN
         junk := CloseComponent(inst);
         inst := NIL;
      END;  { IF }
   END;  { IF }
   ICCStart := err;
END;  { ICCStart }
```

If the Internet Config component isn't installed or can't be opened for any other reason, the routine sets **inst** to nil and fails with a badComponentInstance error. Remember that the calling code, ICStart, will notice this error code and fall back to the link-in implementation, as shown in Listing 4.

If the routine successfully opens a connection to the Internet Config component, it calls the ICCStartComponent routine, which is standard Component Manager glue that calls the component's initialization routine.

The component glue version of GetPref is a lot simpler. It's just a standard piece of Component Manager glue, as shown in Listing 6. The inline instructions of the component glue for GetPref translate into the piece of assembly code shown in Listing 7.

You can read more about the Component Manager and its dispatch mechanism in *Inside Macintosh: More Macintosh Toolbox.*

Calling components from PowerPC code is not described in this article or in *Inside Macintosh: More Macintosh Toolbox*. You can find out how to do this by reading the Macintosh Technical Note "Component Manager Version 3.0" (QT 5).•

Listing 6. The component glue for GetPref

```
FUNCTION ICCGetPref (inst: ComponentInstance; key: Str255;
                           VAR attr: ICAttr; buf: Ptr;
                           VAR size: LongInt): ICError;
INLINE               { standard Component Manager glue }
   $2F3C, $10, $6,   { move.l    #$0010_0006,-(sp) }
   $7000,            { moveq.l   #0,d0 }
   $A82A;            { _ComponentDispatch }
```

Listing 7. Disassembling the component glue

```
move.l   #$0010_0006,-(sp)    ; push the routine selector (6) and the
                              ; number of bytes of parameters (16)
moveq.l  #0,d0                ; _ComponentDispatch routine selector to
                              ; call a component function
_ComponentDispatch            ; call the component through the Component
                              ; Manager
```

THE COMPONENT WRAPPER

Now let's look inside the Internet Config component at the component wrapper (Listing 8). The component wrapper's basic function is to dispatch all of the IC component's routines based on the selector in **params.what**; it uses a big CASE statement to determine the routine's address and then calls the routine with the Component Manager function CallComponentFunctionWithStorage. The Component Manager is smart enough to sort out the parameters at this stage.

Most of the API routines are immediately dispatched by the component wrapper to an internal routine that simply calls the link-in implementation to do the work. For example, the ICCIGetPref routine, shown in Listing 9, calls through to ICRGetPref, changing only the first parameter.

Listing 8. Sections of IC's component wrapper

```
FUNCTION Main (VAR params: ComponentParameters; storage: Handle):
                           ComponentResult;
{ Inside Macintosh has params as a value parameter when it should be }
{ a VAR parameter. Don't make this mistake. }
   VAR
      proc: ProcPtr;
      s:    SignedByte;
BEGIN
   proc := NIL;
   CASE params.what OF
      { Dispatch the routines required by the Component Manager. }
      ...   { routines omitted for brevity }
      { Dispatch the routines that make up the IC API. }
      kICCStart:
         proc := @ICCIStart;
```

(continued on next page)

Listing 8. Sections of IC's component wrapper *(continued)*

```
      kICCGetPref:
         proc := @ICCIGetPref;
      ...   { remaining IC API routines omitted for brevity }
      OTHERWISE
         Main := badComponentSelector;
   END;  { case }
   IF proc <> NIL THEN BEGIN
      IF storage <> NIL THEN BEGIN
         s := HGetState(storage);
         HLock(storage);
      END;  { IF }
      Main := CallComponentFunctionWithStorage(storage, params, proc);
      IF (storage <> NIL) AND
            (params.what <> kComponentCloseSelect) THEN BEGIN
         HSetState(storage, s);
      END;  { IF }
   END;  { IF }
END;  { Main }
```

Listing 9. The component wrapper for GetPref

```
FUNCTION ICCIGetPref (globals: globalsHandle; key: Str255; VAR attr:
                          ICAttr; buf: Ptr; VAR size: LongInt): ICError;
BEGIN
   ICCIGetPref := ICRGetPref(globals^^.inst, key, attr, buf, size);
END;  { ICCIGetPref }
```

So you can see that there are two ways to call ICRGetPref, either from the component's internal routine ICCIGetPref or from the switch glue's ICGetPref. This is consistent with the design outlined in Figure 4. Of course, these routines call two different copies of the code, one linked into the program and one linked into the component.

THE COMPONENT "SMARTS"

The component "smarts" are wedged between the component wrapper and the link-in implementation. Most component wrapper routines don't have smarts; they call straight through to the link-in implementation. Adding smarts to a routine allows it to work better than its link-in cousin without the need to maintain two versions of the routine.

A good example of a smart routine is the component wrapper version of the Start routine, ICCIStart (Listing 10). This fixes a potential localization problem associated with the link-in implementation with a clever sleight of hand. ICCIStart is basically the same as ICCIGetPref in that it immediately calls through to its link-in implementation equivalent. But then it does something tricky: the component calls itself to get the default filename for the Internet Preferences file. For the gory details of why this is "smart," see "Smart Components for Smart People."

One thing to note is that when ICCIStart calls the component to get the default filename, it doesn't do so directly, but instead uses the component glue to call its current_target global variable. Targeting is cool Component Manager technology

Listing 10. A smart component wrapper

```
FUNCTION ICCIStart (globals: globalsHandle; creator: OSType): ICError;
{ Handle the start request, which is basically a replacement for the }
{ open because we need another parameter, the calling application's }
{ creator code. }
   VAR
      err:  OSErr;
BEGIN
   err := ICRStart(globals^^.inst, creator);
   IF err = noErr THEN BEGIN
      err := ICCDefaultFileName(globals^^.current_target,
                           globals^^.inst.default_filename);
   END;  { IF }
   ICCIStart := err;
END;  { ICCIStart }
```

that allows you to write override components (more on this later in "Override Components").

With each new version of Internet Config, the component implementation gets smarter than the link-in implementation. Component smarts are used in IC 1.0 to improve ease of localization; in IC 1.1, they're also used to improve targetability. In a future version of IC, component smarts may be used to implement a preference cache.

THE LINK-IN IMPLEMENTATION

It may be hard to imagine, but everything you've seen so far is glue. The code that does the real work in IC is the link-in implementation. The link-in implementation sees a different view of the ICRRecord, one that contains enough fields to store all the data that the implementation requires. This extended view of the ICRRecord is shown in Listing 11.

The instance field is still there but the link-in implementation ignores it. It's the subsequent fields that are of interest. Most of them are easy to understand with the help of their comments.

SMART COMPONENTS FOR SMART PEOPLE

Because Internet Config needs to know the default filename of the Internet Preferences file when it creates a new preferences file, and because all filenames should be stored in resources so that they can be localized, the default filename should be stored in a resource. This approach is fine for the component, which can get at its resource file with OpenComponentResFile, but doesn't work for the link-in implementation since it can be linked in to a variety of applications.

We considered working around this by requiring all applications to add a resource specifying the name, but this would force all of our developers to add resources to their applications, and the resource ID might clash with their existing resources. The biggest disadvantage, however, is that IC clients are not necessarily applications and may not even have resource files associated with them.

So we solved this problem by making the component version of IC smarter than the link-in version. The link-in version sets default_filename to "Internet Preferences" and leaves it at that, while the component version calls itself to get the correct filename from the resource file.

Listing 11. The full ICRRecord in the link-in implementation

```
TYPE
   ICRRecord = RECORD
      { This entire record is completely private to the }
      { implementation!!! Your code will break if you depend }
      { on the details here. You have been warned. }
      instance: ComponentInstance;
               { nil if no component available; if not nil, then rest }
               {  of record is junk }
      have_config_file: Boolean;
               { determines whether any file specification calls, that }
               { is, ICFindConfigFile or ICSpecifyConfigFile, have been }
               { made yet; determines whether the next field is valid }
      config_file: FSSpec;
               { our chosen database file }
      config_refnum: Integer;
               { a place to store the resource refnum }
      perm: ICPerm;
               { the permissions the user opened the file with }
      inside_begin: Boolean;
               { determines if config_refnum is valid }
      default_filename: Str63;
               { the default IC filename }
      END;
   ICRRecordPtr = ^ICRRecord;
```

The link-in implementation for the Start routine initializes the remaining ICRRecord fields, as shown in Listing 12.

Listing 12. The link-in implementation for Start

```
FUNCTION ICRStart (VAR inst: ICRRecord; creator: OSType): ICError;
   VAR
      junk: ICError;
BEGIN
   inst.have_config_file := false;
   inst.config_file.vRefNum := 0;
   inst.config_file.parID := 0;
   inst.config_file.name := '';
   inst.config_refnum := 0;
   inst.perm := icNoPerm;
   junk := ICRDefaultFileName(inst, inst.default_filename);
   ICRStart := noErr;
END;  { ICRStart }

FUNCTION ICRDefaultFileName (VAR inst: ICRRecord; VAR name: Str63):
                           ICError;
BEGIN
   name := ICdefault_file_name;
   ICRDefaultFileName := noErr;
END;  { ICRDefaultFileName }
```

Finally, there's the link-in implementation for GetPref, portions of which are shown in Listing 13. The actual implementation is a bit long, so the listing leaves out a lot of messing around with resources, bytes, pointers, attributes, and so on. The basic operation of the routine is simple, however: it checks its parameters, opens the preferences file (by calling ICRForceInside), gets the preference, closes the preferences file, and returns.

Listing 13. The link-in implementation for GetPref

```
FUNCTION ICRGetPref (VAR inst: ICRRecord; key: Str255; VAR attr: ICAttr;
                          buf: Ptr; VAR size: LongInt): ICError;
   VAR
      err, err2:             ICError;
      max_size, true_size:   LongInt;
      old_refnum:            Integer;
      prefh:                 Handle;
      force_info:            Boolean;
BEGIN
   max_size := size;
   size := 0;
   attr := ICattr_no_change;
   prefh := NIL;
   err := ICRForceInside(inst, icReadOnlyPerm, force_info);
   IF (err = noErr) AND (inst.config_refnum = 0) THEN BEGIN
      err := icPrefNotFoundErr;
   END;  { IF }
   IF (err = noErr) AND ((key = '') OR
         ((max_size < 0) AND (buf <> nil))) THEN BEGIN
      err := paramErr;
   END;  { IF }
   IF err = noErr THEN BEGIN
      old_refnum := CurResFile;
      UseResFile(inst.config_refnum);
      err := ResError;
      IF err = noErr THEN BEGIN
         ...   { lots of resource hacking here }
         UseResFile(old_refnum);
      END;  { IF }
   END;  { IF }
   IF prefh <> NIL THEN BEGIN
      ReleaseResource(prefh);
   END;  { IF }
   err2 := ICRReleaseInside(inst, force_info);
   IF err = noErr THEN BEGIN
      err := err2;
   END;  { IF }
   ICRGetPref := err;
END;  { ICRGetPref }
```

TOWARD THE FUTURE

The future . . . where Macintosh applications glide along the information superhighway, seamlessly perceiving the user's every preference. You'd better hope your applications are IC aware!

Internet Config is a very flexible system that can expand in several dimensions. Indeed, some are already being explored — in particular, the use of components to maintain and extend the system. And we're looking forward to seeing IC extended in ways we never anticipated.

OVERRIDE COMPONENTS

One of the coolest features of the Component Manager is targeting — one component can capture another and override it. This effectively prevents external programs from using the captured component, while still allowing it to be called by the override component. Very much like inheritance in object-oriented design, this technology lets you write a very simple component that captures the Internet Config component so that you can patch just one routine. For example, the Internet Config RandomSignature extension overrides the ICGetPref routine. If an IC client requests the signature preference, the extension randomly chooses one from a collection of signatures.

The possibilities for override components are endless. Let's say your organization wants to preconfigure all news clients to access a central news server. You can do this by writing a simple override component that watches for programs getting the NNTPHost preference and returns a fixed read-only preference value. This way, all IC-aware news readers use the correct host but can't change it. As we say in the system software business, it's a wonderful third-party developer opportunity.

TOTAL BODY SWAP

Because all client programs call Internet Config through a well-defined API, it's possible to write a replacement for IC and gain complete control of the system. Imagine that you're tired of having the same preferences in all your IC-aware applications. You can change them by writing a replacement that conforms to the existing API. First, replace the Internet Config component with a smarter one that's capable of storing a set of preferences for each application and returning the right preferences to the right application. Then replace the Internet Config application with a much more sophisticated application that can manage multiple sets of preferences, and your job is done. All IC-aware programs will automatically benefit without recompilation.

Or suppose you want to store your user preferences on a central server and access them through some network protocol. Again, IC lets you do it. You could replace the Internet Config component with a network-aware one, and establish the user's identity in some way, perhaps by requiring the user to log on before using any IC-aware programs. You could then choose to use either a Macintosh application to administer the server or tools from the server's native environment.

STAYING CURRENT

No program is ever finished, nor is any program ever 100% bug free. Internet Config is getting better all the time, and you can update to the newest, improved version with a minimum of fuss. When the application detects that its version of the Internet Config Extension is out of date, it simply installs the new one. Because all IC-aware programs are dynamically linked to the component contained within this extension, they automatically receive the update without having to be recompiled.

By the time you read this article, IC 1.1 should be released and busily updating old versions of the Internet Config Extension around the globe. IC 1.1 offers many improvements and bug fixes, including an extended API and a shell for writing override components easily. Share and enjoy!

RECOMMENDED READING

If you want to find out more about Internet Config itself, the following documents may be of interest:

- "Using the Internet Configuration System" by Quinn, *MacTech Magazine*, April 1995.
- *Internet Configuration System: User Documentation* and *Internet Configuration System: Programming Documentation* by Quinn, in the IC User's Kit and IC Developer's Kit, respectively (1994). These kits are provided on this issue's CD.
- "Internet Config FAQ" by Quinn (1994–1995). Available from the ftp site ftp://redback.cs.uwa.edu.au/Others/Quinn/Config/IC_FAQ.txt.

Here's where you can find out more about components, the technology Internet Config is based on:

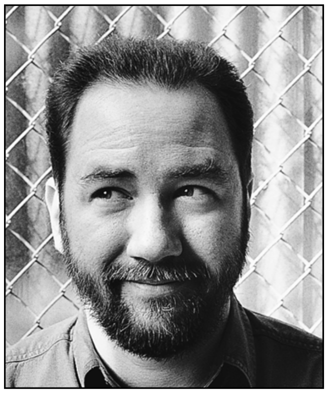
- *Inside Macintosh: More Macintosh Toolbox* (Addison-Wesley, 1993).
- Macintosh Technical Note "Component Manager Version 3.0" (QT 5).
- "Be Our Guest: Components and C++ Classes Compared" by David Van Brink, *develop* Issue 12.
- "Inside QuickTime and Component-Based Managers" by Bill Guschwan, *develop* Issue 13.
- "Somewhere in QuickTime: Derived Media Handlers" by John Wang, *develop* Issue 14.
- "Managing Component Registration" by Gary Woodcock, *develop* Issue 15.

Finally, if you're interested in the mindset of Internet Config's authors, you can do no better than to read the following:

- *He Died With a Felafel in His Hand* by John Birmingham (The Yellow Press, 1994).
- *The UNIX-HATERS Handbook* by Simson Garfinkel, Daniel Weise, and Steven Strassmann (IDG Books, 1994).
- http://www.cm.cf.ac.uk/Movies/

Thanks to our technical reviewers Peter Hoddie, Peter N. Lewis, Jim Reekes, and Greg Robbins. Internet Config is a joint development by Peter N. Lewis and Quinn, with design input from Marcus Jager. We'd like to thank all of those on the Internet Config mailing list and the developers who are supporting the system.•

The Internet Config mailing list is dedicated to discussing the technical details of Internet Config. You can subscribe by sending mail to listserv@list.peter.com.au with the body of the message containing "subscribe config Your Real Name."•



TIM MARONEY

MPW TIPS AND TRICKS

Customizing Source Control With SourceServer

When two engineers on a team edit the same source file at the same time, the resulting chaos can be terrible to behold. Source control was invented to mitigate the problem. Most Macintosh programmers are familiar with the MPW Shell's Check In and Check Out dialogs, and with its Projector commands. The next frontier of custom source control involves SourceServer, a nearly faceless application that implements most of the Projector commands. MPW scripts are easy to write, but they're no match for the power, speed, and friendliness of compiled software. SourceServer exports Projector commands as Apple events, allowing source control from compiled software without launching the MPW Shell in all its pomp and splendor.

Popular third-party development environments often send Apple events to SourceServer for integrated source control. You can also use SourceServer to customize Projector beyond what you might have thought possible. For instance, you can drag source control, kicking and screaming, into the modern world of user experience with drop-on applications. In this column, I'll show you how to check a file in or out with a simple drag and drop, and how to use SourceServer for other things as well. The sample code is provided on this issue's CD; SourceServer is distributed, with documentation, on the MPW Pro and E.T.O. CDs (available from APDA) and with third-party development systems.

APPLE EVENTS FOR SOURCESERVER

Apple events have many faces, but they're primarily a way of communicating between different applications. Each Apple event encapsulates a message as a command with any number of input parameters; the receiver of the message may return any number of result parameters to the sender. The most basic unit of data is the Apple event *descriptor*, which consists of a type code and a data handle. Apple events are built out of descriptors and are themselves special kinds of complex descriptors.

For an excellent introduction to Apple events, see "Scripting the Finder From Your Application" by Greg Anderson in *develop* Issue 20.•

SourceServer's commands are represented as descriptor lists. Its Apple events are exact duplicates of the MPW Shell's Projector commands, but to avoid the overhead of a full command parser, both the command name and each argument are descriptors in the descriptor list. This saves you the trouble of putting quotes and escapes into arguments that might contain spaces or other special characters. The downside is that you have to expand arguments yourself: you can't pass in MPW wildcard characters, backquoted commands for expansion, or other special constructs.

Creating descriptor lists may sound harder than writing MPW scripts, but that's only because it is. I've provided some utility routines to ease the way, though. Listing 1 shows the utilities and illustrates how to make a command to check out a file for modification. As illustrated in the CheckOut routine in this listing, you call the CreateCommand routine first and then use the Add*X*Arg routines to add arguments.

Some of the utilities take Pascal strings, while others take C strings, which could well be considered bad programming practice. I chose this dubious method not because I'm on drugs, but because Pascal strings and C strings are used in different ways. SourceServer's text descriptors are C strings; when passed to these utilities as string constants, they shouldn't be converted from Pascal format in place, since some compilers put constants in read-only areas. If you're internationally savvy, you may have another objection: string constants themselves are bad practice. However, for better or worse, MPW scripts and tools are not internationalized. Just like aliens in *Star Trek*, all MPW programmers are assumed to speak English.

TIM MARONEY wrote TOPS Terminal and BackDrop, and has been a major contributor to TOPS for Macintosh, FaxPro, and Cachet. He has also contributed to Fiery, the Disney Screen Saver, Ofoto, Colortron, and the Usenet Mac Programmer's Guide. Tim learned computer networking while working on the Andrew and MacIP projects at Carnegie Mellon and studied compiler design in graduate school at Chapel Hill. He has written for all three major operating systems and a few minor ones. On the Macintosh, Tim's code has included applications, INITs, control panels, HyperCard stacks, XCMDs, shared libraries, trap patches, plug-ins, scripts, and things more difficult to characterize. Tim is currently doing contract work at Apple, and is available for parties and special events at a nominal cost.•

Listing 1. Creating SourceServer commands

```
OSErr CreateCommand(AEDesc *command, CString commandText)
/* Begin a new SourceServer command; name of command is in commandText. */
{
   OSErr err = AECreateList(NULL, 0, false, command);
   if (err != noErr) return err;
   err = AddCStringArg(command, commandText);
   if (err != noErr) (void) AEDisposeDesc(command);
   return err;
}

OSErr AddCommentArg(AEDesc *command, StringPtr comment)
/* Add a "-cs comment" argument to a SourceServer command. */
{
   OSErr err;
   if (comment[0] == 0) return noErr;
   err = AddCStringArg(command, "-cs");
   if (err != noErr) return err;
   err = AddPStringArg(command, comment);
   return err;
}

/* Other SourceServer argument utilities */
OSErr AddDirArg(AEDesc *command, short vRefNum, long folderID);
OSErr AddProjectArg(AEDesc *command, StringPtr projectName);
OSErr AddUserArg(AEDesc *command, StringPtr userName);
OSErr AddFullNameArg(AEDesc *command, FSSpec *file);
OSErr AddPStringArg(AEDesc *command, StringPtr string);
OSErr AddCStringArg(AEDesc *command, CString string);

OSErr CheckOut(FSSpec *file, StringPtr userName, StringPtr projectName, StringPtr comment)
/* Create a "Check Out Modifiable" command for SourceServer: */
/* CheckOut -m -cs <comment> -d <dir> -project <project> -u <user> <file> */
{
   OSErr           err;
   AEDesc          command;
   CStringHandle   output = NULL, diagnostic = NULL;

   err = CreateCommand(&command, "CheckOut");
   if (err != noErr) return err;
   err = AddCStringArg(&command, "-m");
   if (err == noErr) err = AddCommentArg(&command, comment);
   if (err == noErr) err = AddDirArg(&command, file->vRefNum, file->parID);
   if (err == noErr) err = AddProjectArg(&command, projectName);
   if (err == noErr) err = AddUserArg(&command, userName);
   if (err == noErr) err = AddPStringArg(&command, file->name);
   if (err == noErr) err = SourceServerCommand(&command, &output, &diagnostic);
   (void) AEDisposeDesc(&command);
   /* Display output or diagnostic text as desired. */
   if (output != NULL) DisposeHandle((Handle) output);
   if (diagnostic != NULL) DisposeHandle((Handle) diagnostic);
   return err;
}
```

While on the subject of programming practice, I must gently reprimand SourceServer for its approach to Apple events, in which script commands are simulated through a single 'cmnd' event. SourceServer's idiosyncratic convention dates from the earliest days of Apple events, and modern guidelines discourage this type of design. An application implementing its own Apple events should designate a different command code for each operation, treating arguments as keyword parameters.

Listing 2 shows how to send an Apple event to SourceServer. It's first necessary to find and perhaps launch the SourceServer application. The snippet called SignatureToApp (by Jens Alfke) on this issue's CD accomplishes this with a single function call. Simply pass in the creator code of SourceServer, which is 'MPSP'.

The event must be created before it can be sent. For SourceServer, there's a single parameter, named keyDirectObject, which is the descriptor list containing the command. After sending the event, you must extract the results. The results of an Apple event are returned as keyword parameters in a reply descriptor. First there's the standard keyErrorNumber parameter, which returns an error code if delivery failed. SourceServer returns three other parameters: The 'stat' parameter contains a second error code; if it's nonzero, SourceServer tried to execute the command and failed. When there's an error, there will be diagnostic output in the 'diag' parameter, a handle containing text from the MPW diagnostic (error) channel. Finally, there's standard output — a handle specified by keyDirectObject — which contains explanatory text.

PROJECTDRAG — DRAG AND DROP SOURCE CONTROL

The Macintosh has always had a drag and drop user experience, but the true power and generality of dragging has been widely recognized only recently. The drag paradigm can even be used for source control. To turn Projector into a drag-savvy system, I've written a set of utilities called ProjectDrag (source code and documentation are provided on this issue's CD). You simply drag and drop icons onto the following miniapplications that make up ProjectDrag, and the corresponding function is performed:

- Check In and Check Out, for checking files in and out
- ModifyReadOnly, for editing a file without checking it out
- Update, for bringing a file or folder up to date, as well as canceling checkouts and modify-read-only changes
- ProjectDrag Setup, for configuring the system

These utilities are based on a drop-on application framework called DropShell (written by Leonard Rosenthol and Stephan Somogyi), also on the CD. When a file is dropped onto an application, the application receives an Open Documents ('odoc') event. DropShell takes care of the rigmarole of receiving this and other required Apple events. The ProjectDrag miniapplications pull the file specifications out of 'odoc' events and create SourceServer commands that operate on the files and folders that were dropped on their icons.

> **DropShell is also available on the Internet** at ftp://ftp.hawaii.edu/pub/mac/info-mac/Development/src/ and at other Info-Mac mirror sites.•

Some setup is required. ProjectDrag needs to know the locations of Projector databases. It maps between project names and Projector database files by keeping aliases to database folders in its Preferences folder. To start using a project, simply drag its ProjectorDB file or the enclosing folder onto ProjectDrag Setup. Projector also needs to know your user name, and your initials or a nickname are used in change comments at the start of files. These are stored in a text file in the Preferences folder. ProjectDrag asks you for this information if it can't find it, or you can launch ProjectDrag Setup and give the Set User Name command.

ProjectDrag is scriptable, unlike SourceServer and the MPW Shell. The miniapplications have an Apple event terminology resource ('aete') to advertise their events to scripting systems. This allows you to add source control commands to any application that lets you add AppleScript scripts to its menus.

ProjectDrag is able to run remotely over a network. This circumvents a limitation of SourceServer, which can only be driven locally. ProjectDrag can receive remote Apple events and then drive a copy of SourceServer that's local to it. Among other uses, this could support an accelerator for Apple Remote Access. Checking a file in or out over ARA takes a few minutes, which is fine, especially for those who find tedium particularly enjoyable. Copying files is faster. With local AppleScript front ends for remote ProjectDrag miniapplications, you could copy files to and from a remote "shadow folder" and initiate SourceServer commands at the remote location, where they would execute over a fast network such as Ethernet.

I like to think that I can solve user interface problems in my sleep. When I was writing ProjectDrag, I had a dream of a better user experience. Instead of miniapplications, ProjectDrag would be a magical system extension that would put a single small icon at

Listing 2. Sending commands to SourceServer

```
OSErr SourceServerCommand(AEDesc *command, CStringHandle *output, CStringHandle *diagnostic)
{
   AppleEvent            aeEvent;
   AERecord              aeReply;
   AEDesc                sourceServerAddress, paramDesc;
   ProcessSerialNumber   sourceServerProcess;
   FSSpec                appSpec;  /* SignatureToApp requires this due to a minor bug */
   long                  theLong, theSize;
   DescType              theType;
   OSErr                 err;

   *output = *diagnostic = NULL;   /* default replies */

   /* Find the SourceServer process and make a descriptor for its process ID. */
   err = SignatureToApp('MPSP', NULL, &sourceServerProcess, &appSpec, NULL,
                         Sig2App_LaunchApplication, launchContinue + launchDontSwitch);
   if (err != noErr) return err;
   err = AECreateDesc(typeProcessSerialNumber, (Ptr) &sourceServerProcess,
                      sizeof(ProcessSerialNumber), &sourceServerAddress);
   if (err != noErr) return err;

   /* Create and send the SourceServer Apple event. */
   err = AECreateAppleEvent('MPSP', 'cmnd', &sourceServerAddress, kAutoGenerateReturnID,
                            kAnyTransactionID, &aeEvent);
   (void) AEDisposeDesc(&sourceServerAddress);    /* done with the address descriptor */
   if (err != noErr) return err;
   err = AEPutParamDesc(&aeEvent, keyDirectObject, command);   /* add the command */
   if (err != noErr) { (void) AEDisposeDesc(&aeEvent); return err; }
   err = AESend(&aeEvent, &aeReply, kAEWaitReply + kAENeverInteract, kAENormalPriority,
                kNoTimeOut, NULL, NULL);
   (void) AEDisposeDesc(&aeEvent);    /* done with the Apple event */
   if (err != noErr) return err;

   /* Check for an error return in the keyErrorNumber parameter. */
   err = AEGetParamPtr(&aeReply, keyErrorNumber, typeInteger, &theType, &theLong,
                       sizeof(long), &theSize);
   if (err == noErr && (err = theLong) == noErr) {
      /* Get the standard output from the keyDirectObject parameter. */
      err = AEGetParamDesc(&aeReply, keyDirectObject, typeChar, &paramDesc);
      if (err == noErr) *output = (CStringHandle) paramDesc.dataHandle;
      /* Get the diagnostic output from the 'diag' parameter. */
      err = AEGetParamDesc (&aeReply, 'diag', typeChar, &paramDesc);
      if (err == noErr) *diagnostic = (CStringHandle) paramDesc.dataHandle;
      /* Get the MPW status from the 'stat' parameter -- it becomes our error return. */
      err = AEGetParamPtr(&aeReply, 'stat', typeInteger, &theType, &theLong,
                          sizeof(long), &theSize);
      if (err == noErr) err = theLong;
   }

   (void) AEDisposeDesc(&aeReply);    /* done with the reply descriptor */
   return err;
}
```

some convenient place on the screen. When you dragged a file onto this icon, it would pop open into a temporary window and show you icons for the various options. Dreams are great for creativity, but it's easier to weigh alternatives when you're awake. After I woke up, I realized that miniapplications will be able to do the same thing.

Here's how: In Copland, the next generation of the Mac OS, the Finder will spring-load folders so that they open automatically when you drag onto them. It will also let you stash commonly used folders at the bottom of the screen, where they appear as short title bars. Drag the ProjectDrag folder to the bottom of the screen and you're set! Since the Finder will be providing my dream interface, there's no point in a lot of trap patching and extensibility infrastructure to accomplish the same thing.

Copland will bring another user experience benefit to ProjectDrag: it's planned that document windows will have a draggable file icon in their title bar, so you'll be able to use ProjectDrag on an open document by dragging the icon from its window.

YOU TAKE IT FROM HERE

You can create programs that use SourceServer for many other tasks. On cross-platform projects, Projector is sometimes used to control both platforms' source folders. This can lead to baroque and error-prone processes. With SourceServer, you can create front ends that do the right thing. They could copy to remote folders over a network, or lock read-only files since the other platform doesn't see Projector's 'ckid' resources.

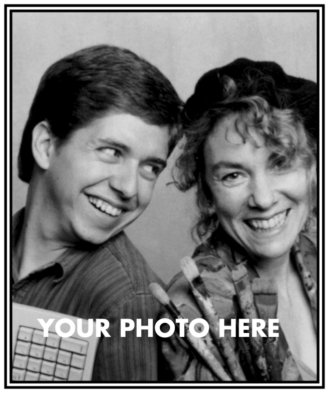
Quality is an interesting area for source control applications. A quality tool could query Projector databases for the frequency and scope of changes at various stages of the project, correlating them with bug tracking to develop project metrics. Along similar lines, a tool could measure the change rate of various files to assist in what the quality gods refer to as root-cause analysis.

SourceServer is much more than a way for development systems to provide integrated source control. It's great for structuring your internal development process as well!

Thanks to Greg Anderson, Arno Gourdol, and B. Winston Hendrickson for reviewing this column.•

Special thanks to Jens Alfke, Jon Pugh, Leonard Rosenthol, and Stephan Somogyi.•

Want to show off your cool code?



YOUR NAME HERE

Do you have code that solves a problem other Macintosh developers might be having? Why not show it off by writing about it in *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.

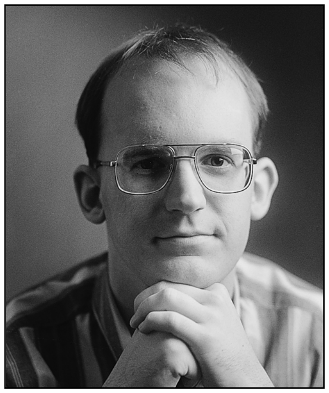
If you're a lot better at writing code than writing articles, don't worry. An editor will work with you. The result will be something you'll be proud to show your colleagues (and your Mom).

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., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

Multipane Dialogs

As applications grow in power and complexity, so does the tendency to present users with numerous cluttered dialog boxes. To simplify the user interface, developers are moving increasingly to dialogs with multiple panes. This article describes how to implement multipane dialogs that users navigate by clicking in a scrolling list of icons.



NORMAN FRANKE

Dialog boxes with multiple panes ("pages" of controls) are an increasingly popular element of the Macintosh user interface. Like simple dialogs, multipane dialogs can be presented when users need to indicate preferences, set attributes of text or graphic objects, or give specifications for complex operations such as searches or formatting, among other things. By grouping related options and providing a single point of interaction for manipulating them, multipane dialogs simplify life for the user and the developer.

Five different kinds of controls for navigating multipane dialogs are in general use: the scrolling list of icons, the pop-up menu, index tabs (simulating the look of tabs on the tops of file folders in a file cabinet), Next/Previous buttons, and icon button sets. Although there aren't any hard-and-fast rules about when you should use one over another, these considerations (suggested by Elizabeth Moller of Apple's Human Interface Design Center) generally apply:

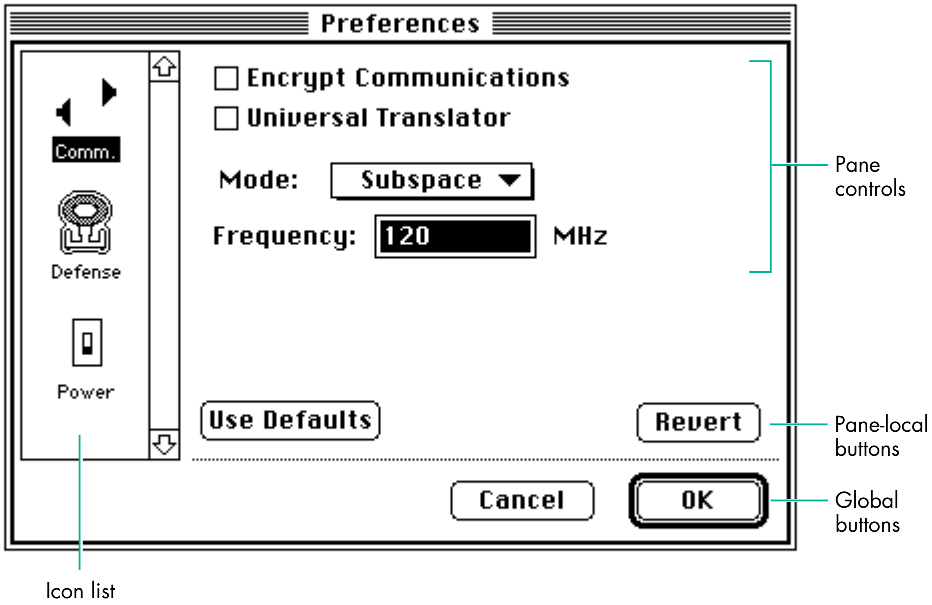
- Novice users have trouble with pop-up menus, so choose a different kind of control if your target audience includes large numbers of these users.
- Index tabs work well for small numbers of panes, but they may not work well when the tabs start overlapping or the number of panes is variable.
- Next/Previous buttons are a good choice when there's more than one mandatory pane. They make it easy for users to step through mandatory and optional panes in sequence.

The sample application MPDialogs on this issue's CD demonstrates the use of a multipane preferences dialog navigated by clicking in a scrolling list of icons, similar to the Control Panel in System 6 and print dialogs in QuickDraw GX. After describing the user interface presented by this sample program, I'll go into the details of how to implement a similar multipane dialog in your own application. Source code for the routines I'll discuss is also included on the CD. This code requires System 7 and is compatible with both black-and-white and color displays.

NORMAN FRANKE misses the large electrical storms and green things of his native Pennsylvania, but not the humidity. He's using the B.S. in computer science he earned from Carnegie Mellon as he writes Macintosh software for a large national laboratory in northern California. Now working on an M.S. in computer science at Stanford, he enjoys writing sound manipulation software for his Macintosh and watching classic and action/adventure movies in his spare time.•

WHAT THE USER INTERFACE LOOKS LIKE

To experience how multipane dialogs work, run the sample program MPDialogs. When you choose Preferences from the File menu, you'll be presented with the interface shown in Figure 1. This is a good illustration of the elements of a multipane dialog.



Preferences

☐ Encrypt Communications

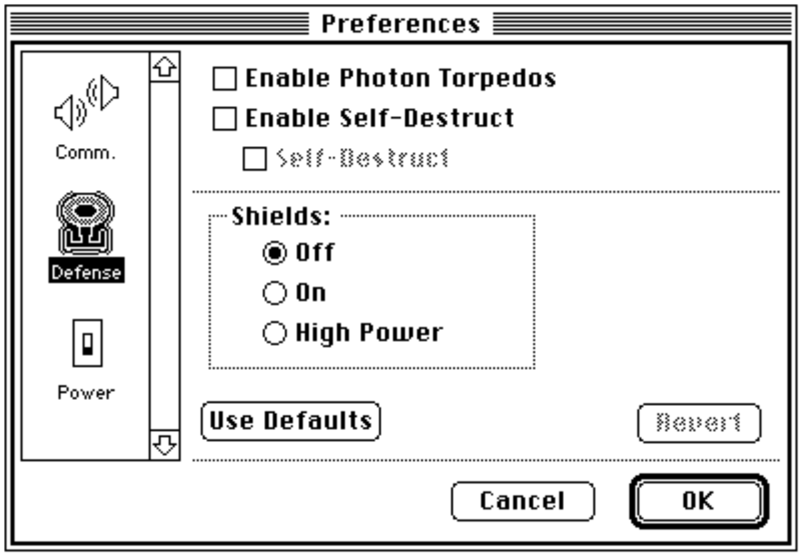
☐ Universal Translator

Mode: Subspace ▼

Frequency: 120 MHz

Figure 1. The Communications pane of the sample multipane dialog

The long vertical rectangle on the left side of the dialog box contains the pane selection icon list. Each icon in this scrolling list has a one-word label under it for identification and represents one pane of the dialog, which is displayed when the user clicks the icon. If you click the Defense icon, for instance, you'll see the pane shown in Figure 2. The arrow and tab keys on the keyboard can also be used to change the pane selection; however, if the current pane contains multiple editable text fields, the tab key will work as in a normal dialog and move the cursor to the next text field.



Preferences

☐ Enable Photon Torpedos

☐ Enable Self-Destruct

☐ Self-Destruct

Shields:

● Off

○ On

○ High Power

Figure 2. The Defense pane of the sample multipane dialog

The bottom portion of the dialog below the line contains two buttons that act on the dialog as a whole: Cancel and OK. The OK button accepts the settings and Cancel aborts all changes and closes the dialog. The two buttons above the line act only on the current pane and are optional: Revert restores the control values in the current pane to what they were when that pane was last opened, and Use Defaults resets the control values in that pane to factory defaults.

The large region above the buttons is where the pane's controls are placed. The sample code supplied on the CD handles actions for checkboxes, radio button groups, and pop-up menus. Command-key equivalents can be used to toggle checkboxes and radio buttons, in addition to the standard keyboard equivalents for OK (Return/Enter) and Cancel (Escape/Command-period). After experimenting with making changes to the control values in the sample program, you can choose Display from the File menu to see the results of your changes.

A couple of custom capabilities can be added to a pane through optional procedures:

- taking special action such as dimming or undimming other controls when items are clicked
- performing data validation such that if validation fails, the user isn't permitted to change panes or exit the dialog with the OK button

These two capabilities are demonstrated in the sample multipane dialog. When you click the Enable Self-Destruct checkbox in the Defense pane, the Self-Destruct checkbox is undimmed. When you enter nondigits in the editable text field in the Communications pane, data validation fails and you're unable to change panes or click OK.

Note that multipane dialogs, like simple dialogs, can take one of three forms:

- standard modal dialog — a dialog that has a border around it and no title bar, that can't be moved around on the screen, and that stays frontmost as long as it's open
- movable modal dialog — a dialog that has a border around it and a title bar, that can be moved around on the screen, and that stays frontmost as long as it's open and the application is frontmost
- modeless dialog — a dialog that looks and behaves like a normal document window with a title bar and a close box, and that isn't always frontmost

The sample program displays a movable modal dialog, but the code provided supports all three forms.

That's all there is to the interface. For some words of wisdom about things to take into account as you design your own multipane dialogs, see "Tips for Designing Multipane Dialogs." Now we'll move along to the details of how to incorporate the multipane dialog routines on the CD into your own application: the resources you need to define, the calls to make to the main routines to open the dialog and handle events, and the customizing you can do with optional procedures.

DEFINING NEEDED RESOURCES

The first step in incorporating the multipane dialog routines is to define the custom resources the code needs. You'll find ResEdit TMPL templates for all the needed resources on the CD. You can put these in the ResEdit Preferences file to make them available at all times or leave them in the application you're editing.

TIPS FOR DESIGNING MULTIPANE DIALOGS

BY ELIZABETH MOLLER OF APPLE'S HUMAN INTERFACE DESIGN CENTER

A multipane dialog is appropriate only when the panes you're presenting are obviously related to one another in some way. With that caveat in mind, here are some suggestions for making your multipane dialogs easy to understand and use:

- Provide a sentence or title to help clarify your intent. For example, you might precede a scrolling list of icons in a preferences dialog with a sentence like "Select items from this list to set your SurfWriter preferences."
- If you use an icon list, label the icons in your list to help users recognize them.
- Visually separate buttons that apply only to the current pane from those that work on all panes (like OK and Cancel in a modal dialog).
- Don't change the size of the dialog or window as the user navigates from pane to pane. Pick a size that accommodates the pane with the most controls.
- Design the dialog so that changing the settings in one pane doesn't change the settings in another. For example, clicking a checkbox in one pane shouldn't disable a checkbox in another pane, because the user won't see the latter action occur and thus won't understand the cause and effect.
- Be consistent in your use of controls. If you use a particular type of control to mean "choose a setting," for instance, don't use the same type of control to mean "navigate between panes" in the same dialog. Users should be able to easily distinguish controls that navigate through multiple panes from controls that make choices in the dialog.
- Order the panes from mandatory to optional, by frequency of use, from general to specific, or, when no other order is apparent, alphabetically. If there are mandatory fields and controls, be sure to put them in the first pane or step the user through mandatory panes before optional ones.
- When the dialog is closed, remember the pane that was last used, unless there are mandatory controls in a pane. If there is a mandatory pane, it should always be displayed when the dialog reopens.

The first resource that needs to be created is the main DLOG and its associated DITL, which will form the basis for the dialog. A sample is provided in the file MPDialogs Resources that you can simply copy into a new project's resource file. The DITL should include six items, numbered as follows:

1. OK button
2. Cancel button
3. Revert button
4. Use Defaults button
5. a user item that defines the icon list rectangle
6. a hidden static text field for default Command-key equivalents

The Revert and Use Defaults buttons can be moved offscreen to make either of them unavailable. (Alternatively, the buttons can be removed and the control **#define**s in the main header file, MPDialogs.h, can be changed to reflect the new numbering.) The icon list is always displayed vertically, and the rectangle doesn't include the scroll bar. The sample application provides the standard Command-key equivalents for OK and Cancel. The standard equivalents for OK are handled in the code; those for Cancel are handled by means of the hidden static text field, which defines default Command-key equivalents for the rest of the controls in the dialog as well.

A DITL needs to be created for each pane. The first item is a hidden static text field that defines Command-key equivalents for the items in the pane; this is in addition to the default list in the main DITL. See "Code for Dialog Command-Key Equivalents" for details of the syntax.

The items are numbered local to each DITL, so that, for example, the first control would be item 2. All user items in the DITL are set to the DrawGray procedure, which outlines the item's rectangle with either the gray color or a stippled gray pattern, depending on the user's monitor.

Next, a DTL# resource should be created with the same resource ID as the main DLOG resource. It contains a list of the resource IDs of the DITLs that comprise a specific multipane dialog and the text displayed under each icon in the list. Then the icon groups are created; they have the same resource ID as the DITL to which they correspond. Small versions of the icons aren't needed, but color versions should be created for display on color-capable Macintosh computers.

Optional DGRP resources can be created for specifying radio button groups. The resource ID is the same as that of the corresponding pane's DITL. Each DGRP can contain multiple groups per pane, if desired; however, a particular radio button should only be used in a single group. Like the per-pane Command-key equivalent strings, items are numbered local to the DITL.

You should also copy the following:

- the pseudo-CDEF with resource ID 251, which provides support for using the icon list as a control (in the file MPDialogs Resources)
- the LDEF with resource ID 130, which implements the icon list definition for the List Manager (in the file Icon LDEF in the LDEF folder)
- optionally, the 'hdlg' resource and corresponding STR# resource for Balloon Help support (in the file MPDialogs Resources)

You can add Balloon Help to a multipane dialog by adding two help items to the individual DITL resources that make up each pane. One is for the controls in the

CODE FOR DIALOG COMMAND-KEY EQUIVALENTS

The Command-key equivalent code I provide in the sample uses a modified version of KeyEquivFilter, a routine in Utilities.c, which is part of DTS Lib on the CD. It takes these two additional parameters:

- The ID of the static text item that contains the mappings. My dialog code calls this routine twice, once for the bottom buttons and a second time for the items in the pane.
- An offset to add to the item numbers when a hit occurs. This allows the code to use relative item numbering for easier specification of Command-key equivalents in panes.

The static text item is an item-match string that follows the general format *=cxxyyzz* or *ccxxyyzz*. The *=c* matches the character *c*, and *cc* matches the character by its ASCII value. The next number, *xx* (a flag byte with the bits set to specify the modifier keys you're checking for), is logically ANDed with the modifier flags from the key-down event and compared to *yy* (a flag byte with the bits set to specify the values of the modifier keys — for example, you can force the Control key to be up). If this comparison is true and if the character *c* matches the character the user typed, the item *zz* is returned as being hit.

Each item-match string is eight characters long and is separated from other such strings that follow by a comma. The numbers in the strings are hexadecimal and case is significant for character matches.

For example, the hidden static text field that's checked for each pane in the sample application is

```
=.190102,1B190102,1B190002
```

The first item-match string checks for a period and for the Control, Option, and Command keys. If only the Command key has been pressed, item 2 is returned as being hit. Similarly, the next item-match string handles Command-Escape (Escape is 1B) and the last item-match string handles Escape by itself.

main DITL and uses an 'hdlg' resource and an STR# resource with the same ID. The second help item is an 'hdlg' resource for each pane's DITL; it should start at item 8 for the first control in the pane. See the file MPDialogs.µ.rsrc on the CD for a sample 'hdlg' resource for the first pane.

CALLING THE MAIN ROUTINES

Now we'll review the calls your application needs to make to the main routines in order to open and close the multipane dialog, handle events, and access the values of the controls in the dialog. But first, let's look at the data your application needs to maintain.

POINTERS AND HANDLES

Your application must maintain a DialogPtr for each dialog used. You also need to declare a handle for storing the returned settings. Passing a pointer to NULL causes the code to allocate a new handle and return it to the caller; otherwise, a handle to an existing record must be provided. For a preferences dialog, this data should be maintained in the application's preferences file in the Preferences folder.

Implementing preferences files is discussed in the article "The Right Way to Implement Preferences Files" in *develop* Issue 18.•

The sample code internally allocates an MPDHdl for each open multipane dialog for storing state information. The handle is stored in the refCon of the dialog.

OPENING, HANDLING EVENTS, AND CLOSING

Your application should call OpenMPDialog for each desired multipane dialog, taking any actions necessary when a dialog is opened, such as disabling menus. This call is passed the resource ID of the DLOG for the dialog, a reference to the handle that stores the returned settings, and four optional parameters, which are described later.

Here's an example:

```
DialogPtr prefDlog = NULL;
Handle thePrefs = NULL;

prefDlog = OpenMPDialog(kPrefDLOG, NULL, NULL, NULL, NULL, &thePrefs);
if (prefDlog) SetMenusBusy();  // If NULL, the dialog couldn't be opened.
```

The main event loop should call DoMPDialogEvent after each event is returned from WaitNextEvent. If DoMPDialogEvent returns true, the multipane dialog routines have handled the event; your application should inspect the DialogPtr to determine whether the dialog has been closed, so that the application can recover from the dialog state. A return value of false indicates that your application should process the event as it would normally. For example:

```
if (DoMPDialogEvent(&prefDlog, &mainEventRec)) {
   // A NULL DialogPtr means the dialog has been closed.
   if (!prefDlog)
      SetMenusIdle();
} else {
   // Process the event as usual.
   ...
}
```

To dispose of the dialog without user interaction, your application can call CloseMPDialog:

```
CloseMPDialog(prefDlog);
```

After the dialog has been closed, it's the application's responsibility to dispose of or save the data handle created with the call to OpenMPDialog. The code I've provided assumes this handle is maintained by the application after creation.

ACCESSING CONTROL VALUES

The following two routines are provided for accessing the control values stored in the data handle:

- GetMPDItem retrieves the value of the control corresponding to the pane and item specified and stores it in a buffer.
- SetMPDItem stores in the handle a value retrieved from a buffer.

Both of these routines assume that the caller knows the length and type of the control's data representation. Items are numbered differently from in the DITL resource — only items that have a value are included, and the values for radio button groups come after those for all other controls in the data. The values of checkboxes, enabled buttons in radio button groups, and pop-up menus are stored as 16-bit integers. Return codes are defined in the header file. Errors are returned for invalid pane and item numbers and buffer lengths.

The routines are declared as follows:

```
short GetMPDItem(Handle theData, short pane, short item, Ptr ptr, short len)

short SetMPDItem(Handle theData, short pane, short item, Ptr ptr, short len)
```

The sample application, in the code for DialogDisplay, provides a basic example of the use of these routines to display the current settings of the controls in the previously closed dialog.

Normally, these routines should be sufficient to access the data in the handle. However, those applications for which it would be more efficient to manipulate the handle directly can use the following format:

```
Last Open Pane
Offset to Pane 1, Offset to Pane 2, ..., Offset to Pane n, NULL
(Pane 1) Length of Item 1, Data for Item 1, ..., Length of Item m, Data
   for Item m, NULL
...
(Pane n) Length of Item 1, Data for Item 1, ..., Length of Item m, Data
   for Item m, NULL
```

The Last Open Pane and the Offset to Pane fields are all long integers and the Length of Item fields are all short integers. The Length of Item value doesn't include the length of itself; to get to the next field you would add

```
Length of Item + sizeof(short)
```

to the pointer. The Last Open Pane field allows the multipane dialog code to display the dialog with the last pane the user had open as the current pane.

That's all you need to know to make basic use of my multipane dialog code. But you can also go a step further: you can customize certain aspects of a multipane dialog by using the four optional parameters to OpenMPDialog mentioned above.

CUSTOMIZING WITH OPTIONAL PROCEDURES

The second through fifth parameters to OpenMPDialog can indicate action procedures that customize dialog behavior by responding to certain events. A value of NULL for any of these parameters tells the application to use the default behavior. To provide custom behavior, you would pass a universal procedure pointer instead of NULL. The procedures can also be changed dynamically, with the InstallAction routine.

The action procedures and the default actions are as follows:

- The Set Defaults action procedure (parameter 2) provides factory defaults for controls. The default action is to set them to 0.
- The Click action procedure (parameter 3) enables you to customize the actions resulting from clicking a control, such as dimming or undimming other controls or performing data validation. The default action is to toggle checkboxes and handle radio buttons via the Radio Group action procedure.
- The Edit action procedure (parameter 4) enables special handling of editable text fields, such as converting the string to an integer. The default action is to store the entire string as a Str255.
- The Radio Group action procedure (parameter 5) enables you to customize the behavior of radio button groups, such as how the values are stored. The default action is to store the value as the index number of the radio button that's enabled in the group; the default value is 1 (the first radio button in the group).

All the action procedure pointers are declared as UniversalProcPtrs for compatibility in case of PowerPC compilation, so they must be allocated before use. The sample program does this by declaring a UniversalProcPtr for each desired action procedure. For example, the one for the Click action procedure is declared as follows:

```
ClickActionUPP myClickAction = NULL;
```

It's initialized in the main routine of the application like this:

```
myClickAction = NewClickActionProc(MyClickAction);
```

Depending on what you want to do in the action procedures, you may need to make use of the MPDHdl stored in the dialog's refCon, mentioned earlier. This is a handle to an MPDRec (shown in Listing 1), which is the main data structure used by the multipane dialog code for state information. None of the elements of this structure should be modified by user code. The four UPP fields can be manipulated via calls to InstallAction and RemoveAction.

The **baseItems** field will be the most useful in the action procedures. It holds the item number of the first item in the pane, which is the hidden static text item used for Command-key equivalents. Thus, if dataH is of type MPDHdl, the index of the first real control (the second DITL entry) in the pane will be (*dataH)->baseItems + 1.

Now let's take a closer look at each of the action procedures.

Listing 1. The MPDRec structure

```
typedef struct MPDRec {
   short          numPanes;      // Number of panes in the dialog
   short          currentPane;   // Current pane being displayed
   short          baseItems;     // Item number of first item in panes
   short          *paneIDs;      // List of IDs for the pane's DITLs
   short          paneDirty;     // Whether Revert should be enabled
   RadioGroupPt   radio;         // Linked list of radio button groups
   Handle         theData;       // Actual storage for dialog values
   Handle         tmpData;       // Temporary storage for dialog values
   Handle         *IconHandles;  // List of icon suites
   ListHandle     theList;       // List Manager list for the icon list
   ClickActionUPP ClickAction;   // Action procedures
   EditActionUPP  EditAction;
   GroupActionUPP GroupAction;
   DefActionUPP   DefAction;
} MPDRec, *MPDPtr, **MPDHdl;
```

THE SET DEFAULTS ACTION PROCEDURE

The Set Defaults action procedure provides factory defaults for checkboxes and other controls, except for radio button groups (handled in the Radio Group action procedure). It's called with a pointer to — and the length of — a buffer holding the internal representation of the value of a single control corresponding to a specific pane and item number. You can call DefaultAction to take the default action for items your code doesn't handle.

The procedure is declared like this:

```
void MySetDefAction(Ptr theData, short len, short iType, short pane,
   short item)
```

The Set Defaults action procedure's defaults for radio buttons apply only to those that aren't part of a radio button group. But using single radio buttons is definitely not advised; all radio buttons should be in groups to be consistent with the *Macintosh Human Interface Guidelines.*

THE CLICK ACTION PROCEDURE

The Click action procedure enables you to customize the actions resulting from clicking a control. For instance, this procedure can handle dimming or undimming other items when certain controls are clicked. It can also provide validation for control settings when the user tries to change the pane or click OK, to ensure that the entered settings make sense.

The procedure receives a DialogPtr and the pane and item numbers. It's declared as follows:

```
short MyClickAction(short mType, DialogPtr dlog, short pane, short item)
```

The mType parameter specifies the message to process when the action procedure is called. The procedure is called with a kInitAction message right after the control is set when the pane is first displayed; this gives you an opportunity to set up the initial state of the dialog. The procedure is called with a kClickAction message after the user has released the mouse button in a control. A kValidateAction message is received for data validation; it's the responsibility of the Click action procedure to put up an alert to notify the user if a setting is unacceptable.

Listing 2 is a Click action procedure from the sample application that undims the third checkbox in the Defense pane (Self-Destruct) if the second checkbox (Enable Self-Destruct) is checked. It also ensures that the editable text field in the Communications pane contains only digits; if this field contains nondigits, the validation fails and the user can't change panes or click OK.

The default Click action procedure, DefaultClickAction, calls the Radio Group action procedure to handle buttons in a radio button group; thus, actions in response to a click in a radio button group should be handled there. Call DefaultClickAction to inherit default functionality for controls not handled in your customization procedure.

Listing 2. A sample Click action procedure

```
short MyClickAction(short mType, DialogPtr dlog, short pane, short item)
{
   MPDHdl   dataH;
   short    iType, val = 0;
   Rect     iRect;
   Handle   iHandle;

   // Obtain multipane dialog state record.
   dataH = (MPDHdl) GetWRefCon(dlog);

   // Handle the second item validation.
   if (mType == kValidateAction) {
      // Validation fails if nondigits are in the field.
      if (pane == kCommPane &&
            item == kFrequency + (*dataH)->baseItems) {
         GetDialogItem(dlog, item, &iType, &iHandle, &iRect);
         GetDialogItemText(iHandle, theStr);
         val = VerifyDigits(theStr);
         if (val)
            StopAlert(ALERT_Invalid, NULL);
      }
      // All other items validate OK.
      return val;
   }

   // If this isn't the second checkbox, handle things the default way.
   if (pane != kMiscellaneousPane ||
         item != kEnableSelfDestruct + (*dataH)->baseItems)
      return (DefaultClickAction(mType, dlog, pane, item));

   // Initialize and Click messages are handled almost the same.
   // Dim the third checkbox based on the value of the second.
   GetDialogItem(dlog, item, &iType, &iHandle, &iRect);
   val = GetControlValue((ControlHandle) iHandle);
```

(continued on next page)

Listing 2. A sample Click action procedure *(continued)*

```
   switch (mType) {
      // Toggle the item in response to the user click.
      case kClickAction:
         val = !val;
         SetControlValue((ControlHandle) iHandle, val);
         // Fall through!
      // In either case, enable/disable next checkbox.
      case kInitAction:
         AbleDItem(dlog, kSelfDestruct + (*dataH)->baseItems, val);
         break;
   }

   // Initialize and Click messages should never fail.
   return 0;
}
```

THE EDIT ACTION PROCEDURE

The Edit action procedure enables special handling of editable text fields. A common implementation is to store the field's string as a long integer, converting the string value to and from this form as needed.

This procedure receives a pointer to a buffer for storage of the control's internal value, a handle to the control, and the pane and item numbers; it returns the length of the space required for the text field. The first parameter is a message that informs the procedure whether to calculate the storage size for this field, initialize the value, or copy the value to or from the field.

The procedure is declared as follows:

```
short MyEditAction(short mType, Ptr hPtr, Handle iHandle, short pane,
   short item)
```

The kCalcAction message requests the amount of storage required for the representation of the field value in memory. The kInitAction message requests that the value of the field be initialized. The kP2TAction message requests that the code retrieve the value of the field and store it in memory (in other words, that the permanent storage value be transferred to the temporary storage area — P2T is shorthand for "permanent to temporary"). Conversely, the kT2PAction message ("temporary to permanent") requests that the code set the field to the value indicated by the representation in memory. Default behavior can be maintained by calling DefaultEditAction, if desired.

Listing 3 is an Edit action procedure from our sample application. Normally, the procedure should check the item and pane numbers to distinguish between different text fields, but the sample application has only one such field.

THE RADIO GROUP ACTION PROCEDURE

To simplify using radio button groups, a single value is stored for the entire group. This value is the relative item number of the enabled button in the group. For example, the value of a group of three radio buttons with the second one enabled would be 2.

Listing 3. A sample Edit action procedure

```
short MyEditAction(short mType, Ptr hPtr, Handle iHandle, short pane,
   short item)
{
   short    ret = 0;
   long     val;
   Str255   textStr;

   Assert(hPtr != NULL);
   switch (mType) {
      case kP2TAction:    // Save value of control.
         GetItemDialogText(iHandle, textStr);
         StringToNum(textStr, &val);
         *(long *) hPtr = val;
         ret = sizeof(long);
         break;
      case kT2PAction:    // Set value of control.
         val = *(long *) hPtr;
         NumToString(val, textStr);
         SetIText(iHandle, textStr);
         ret = sizeof(long);
         break;
      case kInitAction:   // Initialize value.
         *(long *) hPtr = 0;
         ret = sizeof(long);
         break;
      case kCalcAction:   // How much storage do we need for this?
         ret = sizeof(long);
         break;
   }
   return ret;
}
```

In the sample program, radio button groups are stored in a linked list starting from the **radio** field of the MPDRec structure. The RadioGroup structure is defined as shown in Listing 4.

Listing 4. The RadioGroup structure

```
typedef struct RadioGroup {
   struct RadioGroup *next;
   short pane;
   short num;
   short items[1];
} RadioGroup, *RadioGroupPtr;
```

The **next** field points to the next radio button group, to enable traversing the linked list of groups. The **pane** field is the pane number this group belongs to. The **num** field holds the number of items that make up this radio button group. The relative item numbers of these radio buttons are stored in the **items** array.

The Radio Group action procedure enables you to customize the behavior of radio button groups. For instance, an application could choose to store radio button group values differently from the default or handle dimming or undimming of items in response to the user's actions. The Radio Group action procedure receives the same messages as the Edit action procedure. It returns the length of the space required for the radio button group's internal storage; the default is four bytes per group, two for the number of radio buttons and two for the value as a short integer.

Like the Edit action procedure, the Radio Group action procedure is called with the kInitAction and kCalcAction messages. However, these messages occur before the dialog is opened, so the DialogPtr will be NULL at that time. The procedure is declared like this:

```
short MyGroupAction(short mType, RadioGroupPtr group, Handle dataH,
   DialogPtr dlog, Ptr hPtr, short pane, short item)
```

Note that in response to the kInitAction message, the action procedure is expected to store the number of radio buttons in the group in the first two bytes of the internal storage. Here's an example from the default Radio Group action procedure (dataH is of type MPDHdl):

```
for (i = 0; i < group->num; i++) {
   if (GetCheckOrRadio(dlog, group->items[i] + (*dataH)->baseItems - 1))
      *(short *) hPtr = i + 1;
}
```

To obtain the actual item number for the control in the dialog, you just add

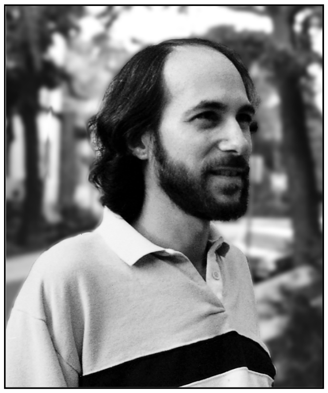
```
(*dataH)->baseItems - 1
```

to the relative number stored in the **items** array, as shown in the above code. As mentioned earlier, the **baseItems** field of dataH is the number of the first pane-specific item in the dialog.

NOW WHAT?

The code that accompanies this article on this issue's CD provides an easy-to-implement method for adding icon-selected multipane dialogs to any application. (The routines for managing radio button groups could be extracted without much difficulty and used elsewhere.) The sample program also provides an example of using the AppendDITL and ShortenDITL routines. So experiment with the sample application and then try out multipane dialogs as a way of simplifying the user interface in your own application.

Thanks to our technical reviewers Tim Craycroft, Nitin Ganatra, C. K. Haun, and Elizabeth Moller. Thanks also to Eric Soldan for ListControl and KeyEquivFilter from DTS Lib.•



CAL SIMONE

ACCORDING TO SCRIPT

Thinking About Dictionaries

I've been thinking lately about the purpose of this column, which debuted in the previous issue of *develop*. Permit me to take a moment to say something about that before I get down to some tips about dictionaries.

During the first couple of years after the birth of the Macintosh, there was a period of chaos, when application developers were figuring out how to extend the basic user interface. For example, some of the most commonly used menu commands appeared in different locations in various applications, and, more important, keyboard shortcuts varied or sometimes weren't present at all. After a while, though, things settled down and almost everyone adopted the standards that were eventually documented in the *Macintosh Human Interface Guidelines*.

AppleScript is the alternate user interface to your application. Now that AppleScript has been available for two years, it's time to move out of the "free-for-all" and develop the same consistency we've all come to enjoy and expect from the Macintosh experience. That's what this column (and the work I do in the AppleScript development community) is all about — encouraging consistency. The tips I offer here reflect undocumented conventions followed by many developers I've worked with, as well as my own thinking about scriptability. Until the time when standards are documented in a "Macintosh Human Scriptability Guidelines," I encourage you to adopt the techniques suggested here.

Though I've said it before, I'll say it one more time: adopting the object model is the single most important factor contributing to consistency in the AppleScript language across applications of different types. One developer I know resists using the object model year after year, arguing that it "isn't appropriate for everything." But the fact is that the object model has been successfully applied to a whole range of applications. Every major C++ framework now supports it or has add-ons to support it, and up-and-coming languages will support it. Even if your application has only one object (such as the dictionary of a small paging program I've seen), just do it!

ORGANIZING YOUR DICTIONARY

So far in the scripting world, various developers have used different schemes in their dictionaries for organizing the events in a suite, the parameters in an event, the properties in an object, and so forth. Some organize them according to their function, others order them alphabetically, and still others don't seem to have any scheme whatsoever (probably because scripting support was added a bit at a time or as an afterthought). For the sake of consistency across different scriptable applications, using some standard scheme is preferable.

If you're including an entire standard suite (such as the Core suite) from AppleScript's system dictionary (listed in the Rez files named EnglishTerminology.r, FrenchTerminology.r, and so on) and then overriding or extending the suite to add your own terms, make sure that your overrides appear in the same order as they do in the system dictionary and that extensions come after all the overrides. If you're implementing your own terminology, either as extensions to existing suites or in your own suites, organize it as described in the following paragraphs.

When you're adding new terms to a previously created dictionary (for example, when upgrading your application to provide deeper scripting support), remember to insert the new terms according to the same scheme or schemes you originally implemented. It's a good idea to keep some notes in your internal design documents describing the ordering schemes you used, so that you can be consistent with your earlier work (unless you're redoing your scripting implementation from scratch — for instance, when you're converting from an old non–object model implementation to the object model).

CAL SIMONE (AppleLink MAIN.EVENT) works way too hard at Main Event Software in Washington DC. He took his last summer vacation five years ago; it's been so long, he's forgotten what a vacation is like, and he can't imagine where he'd go. He's been to beautiful mountainous places like Colorado, Alaska, British Columbia, and Switzerland, and to a few islands like Saint Thomas and the Bahamas. Cal would really like to hear your suggestions on possible future topics for this column, as well as your ideas for good vacation spots.•

Suites. So that your dictionary is consistent with dictionaries in other applications, include the standard Registry suites first (Required suite first, then the Core suite, then any other Registry suites). Then include any custom suites you create.

Events. Order commands that correspond to events in one of four ways: by likelihood of use, according to function, chronologically, or alphabetically. The method you choose will depend on how your application is used and the nature of your users. As an example of each of these schemes, I'll show how some of the Core suite verbs might be organized.

If certain commands are to be used more frequently than others, order them according to likelihood of use. Present those commands that will be used most frequently at the beginning and those seldom used at the end:

get	(more of these than anything else)
set	(quite a few of these, too)
count	(a fair amount of counting)
make	(sometimes new objects are created)
open	(sometimes they're opened)
close	(and closed)
print	(printing isn't done as frequently)
delete	(neither is deleting)
quit	(quitting is done only occasionally)

If your users will logically group the operations, use an ordering according to function. Group together commands that are related in some way:

make	(make and delete)
delete	
open	(open and close)
close	
set	(set and get)
get	
count	(the rest are unrelated)
print	
quit	

If the commands are normally used in a certain order, choose a chronological ordering. First present the commands that will be used first, followed by the commands that will be used later:

make	(this often comes first)
open	(or else opening comes first)
set	(then setting properties)
get	(and later getting properties)
count	(counting comes in the middle)
print	(printing happens later)
close	(then comes closing)
delete	(deleting is near the end)
quit	(last, we bail out)

If the commands aren't going to be used in any particular order, or you don't know what that order is likely to be, and there's no logical grouping, list the commands alphabetically, as the Core suite does. Although alphabetical order isn't as helpful as the other schemes, script writers will at least be able to find commands more easily in your application's dictionary.

Parameters. Make an effort to list parameters in an order that encourages the writing of natural, grammatically correct sentences for commands. For example:

```
make  new  <type class>
      [at  <location reference>]
      [with data  <anything>]
      [with properties  <record>]
```

If the order of an event's parameters doesn't matter as far as sentence style is concerned, order them according to the frequency of likely use.

```
close  <reference>
      saving  <yes|no|ask>
      saving in  <file specification>
```

Object classes and properties. I'd suggest placing the outermost objects in your containment hierarchy first, objects contained in the outermost objects next, and objects that don't contain any other objects last. Remember that every object class representing an actual object must be listed as an element of some other object, eventually leading back to the application class (the null container). Primitive class definitions and record definitions (which aren't part of the containment hierarchy) and abstract classes (which aren't instantiable objects but are used to hold lists of inherited properties) should be placed in the Type Definitions or Type Names suite, and clearly labeled as a record definition or abstract class. (See my article, "Designing a Scripting Implementation," in *develop* Issue 21.)

Properties of objects can be ordered according to one of the schemes described above for events.

WHEN YOU ALLOW MULTIPLE VALUE TYPES

Occasionally in your dictionary you might need to specify a parameter or property for which any of several types is acceptable. Using the wild card ('****') as the type of a parameter or property tells your user that you'll accept *anything* (or at least a wide variety of mixed types). Don't do this to be lazy or to finish your dictionary quickly; do it only if you mean it. If you accept only one type, explicitly indicate so. If you allow two different types, you can either create a compound "type" or use identical keyword entries.

Defining a compound "type." One way of handling cases where you can accept two different value types for a parameter or property is to make up a new "type" to represent a combination of acceptable types in your dictionary. This isn't a real type that you'd have to check for or deal with in your application's code, but instead just serves to indicate in your dictionary that your application will handle *either* type. This works particularly well when the value types are simple. For example:

```
class reference or string: Either a reference or
   a name can be used.
```

You can use your new "type" in a parameter or property definition as follows:

```
class connection
properties:
   window  <reference or string>  -- the
      connection's window can be referred to
      either by a reference or by its name
```

To define a new type, make a new object class and place it in the Type Names suite (see my article in Issue 21).

Using identical keyword entries. You can also use multiple entries with identical keywords to specify alternative ways of filling in a parameter or property value. This works well when the value types are complex or are highly dissimilar. For example, the **display dialog** command has two **with icon** listings, one for specifying the icon by its resource name or ID and the other for displaying the stop, note, or caution icon:

```
display dialog  <anything>  -- title of dialog
   ...  other parameters
   [with icon  <anything>]  -- name or id of the
      icon to display
   [with icon  <stop|note|caution>]  -- or display
      one of these system icons
```

Note the use of "or" in the second entry's comment: make sure you use the same 4-byte ID for *both* parameter entries.

Although you could have many entries to show every possible individual type that a parameter or property takes, this might become confusing to the user. So I'd recommend that you use this sparingly, and when you do use it, try to limit the number of similar entries to 2.

MAKING USE OF THE COMMENT AREA

You can use the comment area (available for each suite, event, parameter, class, and property entry) to help clarify how your vocabulary is to be used. Since your dictionary is often the initial "window" through which a user looks to figure out what to do, descriptive comments can make the user's task a lot easier. And remember that your users aren't necessarily programmers, so you should avoid terms like FSSpec in your comments. I'll give some examples to show you what I mean.

- For Boolean parameters and properties, if there are two possible states, include a description of the true and false conditions, such as "true if the script will modify the original images, false if the images will be left alone."
- If the possible states are on and off, you need only include the true condition ("If true, then screen refresh is turned on") or ask a question ("Is the window zoomed?").
- For enumerations, include a general description of what the parameter or property represents; the individual enumerators should be self-explanatory. For example, "yes|no|ask -- Specifies whether or not changes should be saved before closing."
- Don't use the comment field to explain a set of possible numeric values when an enumeration (with descriptive enumerators) is better. Instead of "0=read, 1=unread, ..." use "read|unread|..."
- For compound "types," describe the parameter or property, as well as the choices for value types listed: "the connection's window (either a reference or name can be used)."
- For "anything" (unless you actually allow *any type* the user can think of), describe which specific types you allow: "[... descriptive info] (a string, file reference, alias, or list is allowed)."
- If you allow either a single item or a list, indicate so: "the file or list of files to open."
- If the parameter or property has a default value (used when the user doesn't include an optional parameter or set the property), mention it (this applies to values of any type): "replacing yes|no|ask -- Replace the file if it exists? (defaults to ask)."

Keep in mind that if you include an entire standard suite (such as the Core or Text suite), your own comments should reflect the style of the comments in that suite. See the Scriptable Text Editor's dictionary as an example of fairly good comment style; it shows the standard versions of the Required, Core, and Text suites and adds some of its own terminology.

A COUPLE MORE DICTIONARY TIPS

While I’m on the subject of dictionaries, here are a couple of extra tidbits.

Use only letters and numbers for terms in dictionaries. Don’t use a hyphen (-), a slash (/), or any other nonalphanumeric characters in your dictionary entries. For example, if you use **Swiss-German**, AppleScript will treat it as **Swiss - German** (subtraction), which is not what you want; if you use **Read/write**, it will be treated as **Read / write** (division). Note that **Read/write** is in the standard Table suite, but it won’t compile properly.

All terms must start with letters. Using **9600** as an enumerator won’t work; you would have to use something like **baud9600**.

Finally, pick names for your terms that are descriptive for a user, especially a nonprogrammer. If you pick a term like **x**, users won’t be allowed to use **x** as a variable name in their scripts. For instance, instead of “x <small integer> -- the x coordinate” use “horizontal coordinate <small integer> -- the x coordinate.”

IT’S YOUR THING

Unlike writing code, designing a scripting vocabulary isn’t an exact science. It’s up to you to decide in what manner (and how effectively) humans will interact with this new interface. Applying “programming language” concepts and standards won’t always work. You need to keep an eye toward the human aspects of the AppleScript language and to work out a scheme that reflects careful attention to your users.

You may occasionally see guidelines here that aren’t completely clear-cut or that even conflict with each other, and every so often I’ll adjust what I’ve said in an earlier column. This is the nature of an evolving language. If you’re not completely at home with this, seek out an expert in scriptability design for advice. But remember, vocabulary design is by nature as much art as science.

Thanks to Sue Dumont and C. K. Haun for reviewing this column.•

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.
1 Infinite Loop, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)974-6395

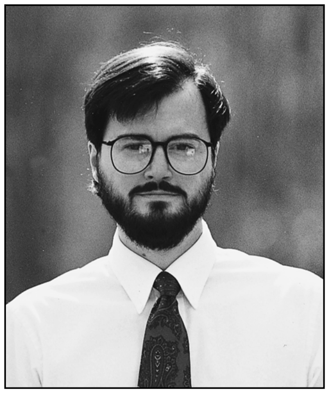
Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
1 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 319, Buffalo, NY 14207-0319 or AppleLink APDA (on the Internet, apda@applelink.apple.com). You can also call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, or (716) 871-6555 from all other locations.

Document Synchronization and Other Human Interface Issues

One of the things the Finder does best is maintain the illusion that an icon and its window represent a single object. Using the routines described in this article, your application can help maintain that illusion. You can ensure that when the user renames an open document, the change is reflected in the document window's title. You can also gracefully handle problems that may arise if the document file is moved. Other improvements that make your application's interface more consistent with the Finder's include preventing a second window from opening when an open document's icon is double-clicked and adding a pop-up navigation menu to the document window's title bar.



MARK H. LINTON

To rename a folder or file in the Finder, you click the icon name, type a new name, and press Return. For folders, if the window is open, the change is reflected right away in the window's title bar. But for files, if the document is open in your application, its window may not reflect the name change. Try this little experiment: Create a document in your application and save it. Switch to the Finder, find your document, and change its name. What did your application do? If it's like most applications, nothing happened: the document window has the same name as before. Go ahead and try to use Save As to give the file the same name you gave it in the Finder. You probably get an error message. Now try to save the document under the original name. Do you still get an error message? Quit your application and read on for a way out of this frustration.

The only convenient way for a user to rename a document is with the Finder. (The Save As command doesn't rename a document; it creates a copy of the document with a new name.) As you've just seen, name changes made in the Finder aren't automatically reflected in an open document window. Another change that's often not picked up by the application is when the user moves the document to a different folder. The code in this article helps synchronize your application's documents with their corresponding files, so that a document will respond to changes made outside the application to its file's name or location.

This article also describes how to prevent a duplicate window from being opened if the user opens an already open document in the Finder and how to add a pop-up menu to the document title bar to help the user determine where the file is stored. All

MARK H. LINTON (mhl@hrb.com) lives in Centre Hall, Pennsylvania, with his wife Gretchen. When he isn't jetting around the globe or meeting with some high government officials as part of his job as senior engineer at HRB Systems, he can be found in his log cabin at the base of Mount Nittany playing with his Macintosh.•

the code for implementing these features is provided on this issue's CD, along with a sample application that illustrates its use.

DOCUMENT SYNCHRONIZATION

The *Electronic Guide to Macintosh Human Interface Design* says that applications should "match the window title to the filename." Specifically, when a user changes the document name in the Finder, you should update all references to the title. The guide also refers to the *Macintosh Human Interface Guidelines*, page 143, where it says, "The document and its corresponding window name must match at all times."

When I first started looking at the problem of document synchronization, I assumed that the animated example in the *Electronic Guide to Macintosh Human Interface Design* was the way to go. In this animation, the application checks for a name change when it receives a resume event. However, I became uncomfortable with this approach, because it would cause a delay between the user's changing the name of the document in the Finder and the application's updating the window title. Using a resume event relies on a separate action by the user, namely, bringing the application to the foreground. This seemed nonintuitive and didn't support the illusion that a window and its icon represent a single object. Also, it's possible that with Apple events and AppleScript an application could be launched, do some work, and quit without ever being frontmost — that is, without ever receiving a resume event.

The truth is that these days, with multiple applications running at the same time, with networked, shared disks everywhere, and with applications and scripts pulling the puppet strings as often as users, a file's name or location may change at any time, whether the application is in the foreground or the background. A script might move or rename a file or, if the file is on a shared volume, another user on the network could move or rename it or even put the file in the Trash — all behind the application's back. The only solution I found under the current system software was to regularly look at the file to see if its name or location has changed. In other words, the application has to poll for changes.

Polling is generally a bad idea, but there are cases when it's the only reasonable way to accomplish a task, and this is one of them. However, I tried to keep the polling very "lightweight" and low impact by using the following guidelines:

- An application shouldn't poll any more often than it absolutely needs to. Waking up an application causes a context switch, and context switches take a significant amount of time. Forcing the system to wake up an application every few ticks just so that it can look for file changes would be a bad idea, especially when the application is in the background. Instead, the application should poll only when an event has already been received — that is, when the application is awake. Set your WaitNextEvent sleep time appropriately, and wait at least a second or two between "peeks." (The Finder, for instance, polls for disk changes every five seconds or so.)
- Avoid any polling that causes disk or network access; if at all possible, examine only information that's in RAM on the local machine. Network access in particular can be a real drain on performance.

The sample code follows this advice, doing everything it can to be unobtrusive. It polls for file changes only once every second while in the foreground. In the background, the application's WaitNextEvent sleep time is set to ten seconds, so it only wakes up — and thus polls — every ten seconds if nothing else is going on. To detect changes to files, I chose to examine the volume modification date of the volume containing the file, since this information is always available in local RAM, even for a shared volume. If that date changes, I look deeper to see if the change is one I'm interested in. As you dip into the code, you'll see the details.

I use the file reference number to track files because it survives changes in the name and parent directory. However, this requires that the files be kept open. If you can't keep your files open, you might want to look at John Norstad's excellent NewsWatcher application, which uses alias records to synchronize files. NewsWatcher is on this issue's CD; its official source can be found at ftp://ftp.acns.nwu.edu/pub/newswatcher/.•

Friendly as it is, this polling solution is appropriate only for the current system software; future system software versions (such as Copland, the next generation of the Mac OS) will provide a much better way to detect changes. Your application will be able to subscribe to notification of changes that it's interested in. In fact, polling the current file system structures will be unfriendly behavior under Copland, which will have demand-paged virtual memory and a completely new file system. For this reason, the sample code is designed to work only under System 7. You'll be able to easily retrofit the code to run under Copland once the details of the correct way to detect file changes have been worked out.

THE HEART OF THE MATTER

Every Macintosh programmer eventually comes to grips with how to keep track of all the information associated with a document. I use a structure called a *document list* and I have a set of routines that support it. The document list reverses some common assumptions used by developers. Developers often use the window list to track their windows and attach their document data to it, but this limits Apple's ability to redefine the window list. My recommendation is to create a document list (almost identical to the window list) containing the document data and attach the windows to it. In this way, the actual *structure* of the window list is not a concern. You'll find my implementation of the document list and its supporting routines on this issue's CD.

While the code presented here is specific to my implementation, you can easily generalize it as needed. The code below shows how your application might call DSSyncWindowsWithFiles, a routine that keeps your documents synchronized with the Finder by checking for and handling changes made outside the application to file names or locations. Call the routine from within your main event loop when you receive an event (including null events). Note that error checking has been removed from the code shown in the article, but it does appear on the CD.

```
while (!done) {
   gotEvent = WaitNextEvent(everyEvent, &theEvent, gSleepTime,
      theCursorRegion);
   if (gotEvent)
      DoEvent(&theEvent);
   DSSyncWindowsWithFiles(kDontForceSynchronization);
}
```

This minor change does most of the work for your application. The machinery that makes it happen lies within DSSyncWindowsWithFiles (see Listing 1). This routine first checks to make sure that enough time has passed since the last check for changes. If so, or if the caller requested immediate synchronization, it iterates through each of the windows registered in the document list, calling DSSyncWindowWithFile to process each of these windows.

DSSyncWindowWithFile, shown in Listing 2, begins by getting the file reference number for the window from the document list. If it's appropriate to continue

Listing 1. DSSyncWindowsWithFiles

```
#define kCheckTicks 60

pascal void DSSyncWindowsWithFiles(Boolean forceSync)
{
   WindowPtr      theWindow;
   static long    theTicksOfLastCheck = 0;
   long           theTicks;

   theTicks = TickCount();
   if (theTicks > (theTicksOfLastCheck + kCheckTicks) || forceSync) {
      theTicksOfLastCheck = theTicks;
      for (theWindow = DSFirstWindow(); theWindow != nil;
           theWindow = DSNextWindow(theWindow)) {
         DSSyncWindowWithFile(theWindow);
      }
   }
}
```

Listing 2. DSSyncWindowWithFile

```
pascal void DSSyncWindowWithFile(WindowPtr aWindow)
{
   short theFRefNum;

   DSGetWindowDFRefNum(aWindow, &theFRefNum);
   if (DoSyncChecks(theFRefNum, aWindow)) {
      HandleNameChange(theFRefNum, aWindow);
      HandleDirectoryChange(theFRefNum, aWindow);
      HandleMoveToTrash(theFRefNum, aWindow);
   }
}
```

(DoSyncChecks returns true), DSSyncWindowWithFile calls three other routines to handle name changes, changes that move the file to a different folder, and changes that move the file to the Trash.

THE CHECKPOINT

The DoSyncChecks routine (Listing 3) checks for changes to the volume that the file is on. If the volume has been modified, DoSyncChecks returns true to DSSyncWindowWithFile, which consequently calls the next three routines — HandleNameChange, HandleDirectoryChange, and HandleMoveToTrash.

A FILE BY ANY OTHER NAME

After determining that the volume containing the file has been modified, DSSyncWindowWithFile calls HandleNameChange (Listing 4). This simple routine compares the names of the window and the file; if they're not exactly the same, it updates the window to reflect the new filename. A minimal implementation of document synchronization might include only this routine.

Listing 3. DoSyncChecks

```
static Boolean DoSyncChecks(short aRefNum, WindowPtr aWindow)
{
   Boolean       doCheck = false;
   unsigned long theLastDate, theDate;
   short         theVRefNum;

   if (aRefNum != 0) {
      DSGetWindowFileVRefNum(aWindow, &theVRefNum);
      GetVolumeModDate(theVRefNum, &theDate);
      DSGetWindowVLsBkUp(aWindow, &theLastDate);
      if (theLastDate != theDate) {
         DSSetWindowVLsBkUp(aWindow, theDate);
         doCheck = true;
      }
   }
   return doCheck;
}
```

Listing 4. HandleNameChange

```
void HandleNameChange(short aFRefNum, WindowPtr aWindow)
{
   Str255   theTitle, theName;

   GetWTitle(aWindow, theTitle);
   GetNameOfReferencedFile(aFRefNum, theName);
   if (!EqualString(theTitle, theName, true, true))
      SetWTitle(aWindow, theName);
}
```

Have you been wondering where the magical file management calls that DoSyncChecks and HandleNameChange use come from — for example, GetVolumeModDate and GetNameOfReferencedFile? See the file EvenMoreFiles.c on the CD for details. This is my tribute to Jim Luther's excellent MoreFiles collection. Whenever I need a routine that's not in the standard header, I write it and add it to the collection. Someday we'll be up to SonOfMoreFiles and NightOfTheLivingMoreFiles.•

MOVING TO A NEW NEIGHBORHOOD

After checking, and possibly synchronizing, the filename, DSSyncWindowWithFile calls HandleDirectoryChange (Listing 5) to see whether the file has been moved. This routine starts out by comparing the old parent directory to the new parent directory. If they're not the same, the file has been moved and the routine stores the file's new parent directory for later use by the application. It's possible that the file was moved to a parent for which the user doesn't have access privileges. In that case, a later Save will fail and revert to a Save As.

GETTING TRASHED

Finally, DSSyncWindowWithFile calls HandleMoveToTrash (Listing 6) to see if the file is in the Trash. If it is, HandleMoveToTrash gets the FSSpec corresponding to the file reference number, which will be needed later. If the application is running in

Listing 5. HandleDirectoryChange

```
void HandleDirectoryChange(short aFRefNum, WindowPtr aWindow)
{
   long     theOldParID, theNewParID;

   DSGetWindowFileParID(aWindow, &theOldParID);
   GetFileParID(aFRefNum, &theNewParID);
   if (theOldParID != theNewParID)
      DSSetWindowFileParID(aWindow, theNewParID);
}
```

Listing 6. HandleMoveToTrash

```
static void HandleMoveToTrash(short aFRefNum, WindowPtr aWindow,
      Boolean *inTrashCan)
{
   FSSpec         theFile;
   Boolean        inBackground;
   short          theResponse;
   EventRecord    theEvent;

   FileInTrashCan(aFRefNum, inTrashCan);
   if (*inTrashCan)
      GetFileSpec(aFRefNum, &theFile);
   if ((aFRefNum != 0) && *inTrashCan) {
      if (DSIsWindowDirty(aWindow)) {
         InBackground(&inBackground);
         if (inBackground) {
            DSNotify();
            do {
               InBackground(&inBackground);
               if (WaitNextEvent(everyEvent, &theEvent, gSleepTime, nil))
                  DoEvent(&theEvent);
               FileInTrashCan(aFRefNum, inTrashCan);
            } while (inBackground && *inTrashCan);
            DSRemoveNotice();
         }
         if (*inTrashCan) {
            ParamText(theFile.name, "\p", "\p", "\p");
            theResponse = Alert(rCloseAlert, nil);
            switch (theResponse) {

               case kSave:
                  DoSave(aWindow);
                  /* Fall through */

               case kDontSave:
                  ZoomWindowToTrash(aWindow);
                  DoCloseCommand(aWindow);
                  break;
```

(continued on next page)

Listing 6. HandleMoveToTrash *(continued)*

```
               case kPutAway:
                  DSAESendFinderFS(kAEFinderSuite, kAEPutAway, &theFile);
                  *inTrashCan = false;
                  break;
            }
         }
      } else   /* Window is clean; just close it */
         DoCloseCommand(aWindow);
   }
}
```

the background, and there are unsaved changes to the document, the routine notifies the user (with the Notification Manager) that the application needs assistance. While waiting for the user to respond to the request for assistance, HandleMoveToTrash handles events normally and also checks to see whether the user has moved the document back out of the Trash. After all, there's no sense in asking the user what to do about a file in the Trash if it's no longer there. If the user responds to the request, or moves the file out of the Trash while the application is still in the background, HandleMoveToTrash removes the notification. If the file is still in the Trash when the application becomes frontmost, an alert appears asking the user what to do.

Now if this were the Finder, there would be no question of what to do in this situation. When the user drags the icon for a folder to the Trash, the folder is essentially gone, so the associated window doesn't remain on the desktop. In the application world, life is a little more problematic. What happens if there are unsaved changes in the document? If the application blindly closes the document when the user drags the icon to the Trash, data could be lost. This would be a Bad Thing.

My mother always told me, "When in doubt, ask." So if there are unsaved changes to the file, an alert gives the user three choices: Don't Save, Remove From Trash, and Save. The Save and Don't Save options are simple: each closes the window as expected. Remove From Trash is a little tricky and takes advantage of the Scriptable Finder and Apple events.

The Remove From Trash case is similar to the Finder situation in which the user decides not to throw the document in the Trash and chooses Put Away from the File menu. HandleMoveToTrash handles this change of mind the same way the Finder handles it with Put Away: it sends the Finder a Put Away Apple event specifying the file in question as the target. (If the Scriptable Finder isn't available, the same action can be simulated manually; see the code on the CD for details.)

HOW CAN YOU BE IN TWO PLACES AT ONCE?

That's all there is to document synchronization. Now let's take a look at some other ways you can make your application's interface more consistent with the Finder's.

Many applications create a new window when an already open document is opened again in the Finder. But if the Finder were to open a second copy of a folder when you double-click the icon of a folder that's already open, wouldn't you be surprised? One of the guiding principles of human interface design is consistency; if your application doesn't perform the same action as the Finder (in this case, bring an already open window to the front), the user must learn and remember what will happen in each particular situation. This detracts from the user's happiness with your application.

Making your application notice that the document is already open is easy if you're using the document list. The following code would appear where you normally call your open-file routine. When the application receives an event to open a file, it checks to see if the file is already registered in the document list. If it's registered, the application simply brings it to the front instead of opening it again.

```
if (DSFileInDocumentList(aFile, &theWindow))
   SelectWindow(theWindow);
else
   DoOpenFile(aFile);
```

POP-UP NAVIGATION

A nifty feature introduced with the System 7 Finder is the pop-up menu in the title bar that allows the user to determine the location of an open folder and to navigate the file system without having to resort to browsing (see Figure 1). The user simply holds down the Command key and presses on the window title to see the menu. The computer knows where your document is; it just needs a good way to present the information. If you have Metrowerks CodeWarrior, you'll find that it does something similar to the System 7 Finder. Your application can provide the same interface.

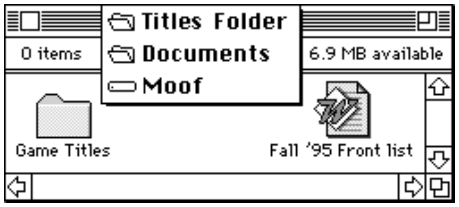


Figure 1. The Finder's pop-up navigation menu

To provide a pop-up navigation menu for your document windows, replace the existing call to FindWindow in your mouse-down event handler with a call to the DSFindWindow routine. DSFindWindow is simply a wrapper for the Window Manager's FindWindow routine. If FindWindow returns inDrag, DSFindWindow does some additional checking to determine whether the window is frontmost, the Command key is down, and the mouse is in the window title area. If the mouse-down event meets these conditions, DSFindWindow calls DSPopUpNavigation, which implements the menu and returns inDesk as the window part, telling the application to ignore the click.

Note that DSPopUpNavigation makes an assumption about the location of the window's title that may not be true for nonstandard window types or in future versions of the system software. In such cases the pop-up menu will still work fine, though it may not be cosmetically correct. This is another area of the code that should be revisited when Copland becomes available.

CONSISTENCY PAYS OFF

Consistency is one of the key principles that make using the Macintosh the wonderful experience that it is. If your program responds to the user's actions in the same way that the Finder does — in particular, maintaining the illusion that an icon and its window represent a single object — your users can explore your application with skills they've already acquired. The techniques presented here show how to provide that extra measure of consistency with the Finder that keeps the Macintosh interface clean, consistent, and seamless. They're not too hard to implement, they're fun, and they just happen to be useful!

RECOMMENDED READING

- *Electronic Guide to Human Interface Design* (Addison-Wesley, 1994). This CD (available from APDA) combines the *Macintosh Human Interface Guidelines* and its companion CD, *Making It Macintosh*, into one easy-to-swallow capsule. Take one every night before going to bed, and wake up with a more consistent user interface.
- *Macintosh Human Interface Guidelines*, (Addison-Wesley, 1993). Available separately from APDA in book form.
- Polya, G., *How to Solve It* (Princeton University Press, 1945). This book explains a logical approach to problem solving. Very simply, the approach is: understand the problem, compare it to a related problem that has been solved before to arrive at a plan, carry out the plan, and examine the solution. That's what I've done with the subject of this article.

Thanks to our technical reviewers Jens Alfke, Greg Anderson, Arno Gourdol, Bill Keenan, Jim Luther, and Elizabeth Moller.•

Macintosh Q & A

Q *How do I create a menu with an icon as its title?*

A Set the menu title to 0x0501*handle*, where *handle* is the result of calling GetIconSuite. An example snippet of code follows; this code assumes that the menu title is already five bytes long.

```
void ChangeToIconMenu()
{
   Handle      theIconSuite = nil;
   MenuHandle  menuHandle;

   GetIconSuite(&theIconSuite, cIcon, svAllSmallData);
   if (theIconSuite) {
      menuHandle = GetMenuHandle(mIcon);
      if (menuHandle) {
         // Second byte must be 1, followed by the icon suite handle.
         (**menuHandle).menuData[1] = 0x01;
         *((long *)&((**menuHandle).menuData[2])) = (long)theIconSuite;
         // Update display (typically you do this on startup).
         DeleteMenu(mIcon);
         InsertMenu(menuHandle, 0);
         InvalMenuBar();
      }
   }
}
```

Q *We're drawing palette icons with a loop that consists basically of the following:*

```
GetIcon...
HLock...
CopyBits...
ReleaseResource...
```

Our native PowerPC version seems to draw these icons a lot more slowly than even our 680x0 version running under emulation, which suggests a Mixed Mode Manager slowdown. Which of these routines are currently native on the PowerPC? GetIcon and ReleaseResource together seem to take over 90% of the time.

A As you suspected, the Resource Manager calls you're using aren't native, and they generally call File Manager routines, which aren't native either. But be careful: what's native and what isn't is changing over time, and you shouldn't design your application based on today's assumptions.

That said, relying on the Resource Manager to be fast is generally not a good idea, native or not. One approach is to "cache" the icons, making sure they're in RAM at all times. (In general, you should do this for any user interface element that will be redrawn repeatedly while your application is open.) You can load your icons as one of the first few operations in your initialization code, just after calling MaxApplZone (possibly moving them high and locking them, since you don't want them to move during a CopyBits operation). This technique yields very good performance on the redraws that the palette needs, in exchange for a few kilobytes of memory. Don't forget to mark the resources as nonpurgeable.

Even better, if it will suit your purposes, would be to use the Icon Utilities to retrieve and draw your icons (as documented in *Inside Macintosh: More Macintosh*

Toolbox) and to build an icon cache. Using the Icon Utilities helps your application do the right thing for different screen depths. Also, the icon-drawing routines have been optimized to perform well under a variety of conditions.

Q *How can we detect that our application is already running and bring it to the front?*

A Simply iterate through the currently running processes with GetNextProcess, calling GetProcessInformation for each one and comparing its process signature with your application's (for an example, see the article "Scripting the Finder From Your Application" in *develop* Issue 20, page 67, Listing 1). If your application is running, call SetFrontProcess to bring it to the front.

Q *WindowShade is causing a problem for our application, which saves the window position and size when it saves a document to disk. If our application's windows are "rolled up" with WindowShade, its windows appear to have zero height. Is there any way to determine whether a window is rolled up? If so, can we determine its true size and the global coordinates of the top left corner of the content region, so that we can restore and reposition the window when the document is reloaded from disk?*

A When WindowShade "rolls up" a window, it hides the content region of the window. You can tell a window is rolled up when its content region is set to an empty region and its structure region is modified to equal the new "shaded" window outline. WindowShade doesn't do anything with the graphics port, though, so if you need to store the window's dimensions before closing it, use the window's portRect.

With regard to the window's position, WindowShade modifies the bottom coordinates of the structure and content regions of the window, but the top, left, and right coordinates are not changed. These are global coordinates, so you can use the top and left coordinates to track and save the global position of the window on the screen regardless of whether the window is rolled up.

Q *Sometimes balloons won't show up when I call HMShowBalloon; I get a paramErr (-50) instead. The hmmHelpType is khmmTEHandle. HMShowBalloon calls TextWidth on the hText of my TEHandle (the result of which is 1511, the width of 338 characters), then multiplies that by the lineHeight (12), yielding 18132. It then compares this to 17000, doesn't like the result, puts -50 into a register, and backs out of everything it has done previously. What's the Help Manager doing?*

A The Help Manager checks against 17000 to ensure that the Balloon Help window will always be smaller than a previously determined maximum size. Currently, you're limited to roughly the same number of characters with a styled TEHandle as you are with a Pascal string: 255 characters.

Keep your help messages small by using clear, concise phrases. If you absolutely need more text in a balloon, you can create a picture of it and use khmmPict or khmmPictHandle to specify it for your help message. This is not recommended, however; "picture text" has the disadvantage of being difficult to edit or translate to other languages.

Q *Is there any way I can stop LClick from highlighting more cells when the user drags the cursor outside the list's rView area? My program allows users to select more than one* *item from a list and then drag and drop these selected items into another list. But I run into a problem with the LClick function: when I drag these items outside the list's rView area, it still highlights other cells. What can I do?*

A If you want to use LClick and not change the highlighting of cells when the cursor leaves the rView of the list, you should install a click-loop procedure that tracks the mouse. When the mouse is outside your list's rectangle, return false to tell the List Manager that the current click should be aborted. It turns out that this is a nice way to start a drag as well, since you know that the mouse has left the rectangle. It might look like this:

```
GetMouse(&localPt);
if (PtInRect(localPt, &(*list)->rView) == false)
   return false;     // We're out of the list.
else
   return true;
```

Q *I'm developing a Color QuickDraw printer driver and want to match colors using ColorSync 1.0.5 with a custom CMM. I was told that for efficiency I should manually match colors inside my Stdxxx bottlenecks, instead of calling DrawMatchedPicture. Is this really more efficient? Why?*

A Surprising as it may be, it is more efficient for printer drivers to manually match colors inside Std*xxx* bottlenecks than to call DrawMatchedPicture. This is because ColorSync 1.0's DrawMatchedPicture doesn't use bottlenecks as you expected. It does install a bottleneck routine that intercepts picture comments (so that it can watch the embedded profiles go by), but it doesn't do the actual matching in bottleneck routines. Instead, it installs a color search procedure in the current GDevice. Inside the search procedure, each color is matched one at a time.

While this implementation has some advantages, it's painfully slow on PixMaps, because even if the PixMap contains only 16 colors, each pixel is matched individually. This has been changed in ColorSync 2.0. To boost performance, PixMaps (which are, after all, quite common) are now matched in the bottlenecks instead of with a color search procedure. (See the Print Hints column in this issue of *develop* for more on ColorSync 2.0.)

Q *I need to add some PostScript comments to the beginning of the PostScript files generated by the LaserWriter GX driver. On page 4-119 of Inside Macintosh: QuickDraw GX Printing Extensions and Drivers, it says that you can override the GXPostScriptDoDocumentHeader message to do this. I wrote a QuickDraw GX printing extension to implement this, assuming that all I had to do was to override the GXPostScriptDoDocumentHeader message and buffer the desired data with Send_GXBufferData. Here's an example of my code:*

```
OSErr NewPostScriptDoDocumentHeader(gxPostScriptImageDataHdl hImageData)
{
   OSErr    theStatus = noErr;
   char     dataBuffer[256];
   long     bufferLen;

   strcpy(dataBuffer, "%%DAVE'S TEST DATA");
   bufferLen = strlen(dataBuffer);
   theStatus = Send_GXBufferData((Ptr) dataBuffer, bufferLen,
                                          gxNoBufferOptions);
   if (theStatus != noErr)
      return theStatus;
   theStatus = Forward_GXPostScriptDoDocumentHeader(hImageData);
   return theStatus;
}
```

Unfortunately, this causes a bus error when Send_GXBufferData is called, even if I put Send_GXBufferData after the call to Forward_GXPostScriptDoDocumentHeader. Why doesn't this work?

A The override in your extension is basically correct, but the order of your code needs to be slightly different:

```
// Note that the string is terminated with a return character:
#define kTestStr "%%DAVE'S TEST DATA\n"

OSErr NewPostScriptDoDocumentHeader(gxPostScriptImageDataHdl hImageData)
{
   OSErr    theStatus = noErr;
   char     dataBuffer[256];
   long     bufferLen;

   theStatus = Forward_GXPostScriptDoDocumentHeader(hImageData);
   if (theStatus != noErr)
      return theStatus;

   // Note that we do (sizeof(...) - 1) below to strip off the C string
   // null terminator for the string defined.
   theStatus = Send_GXBufferData(kTestStr, (sizeof(kTestStr) - 1, 0);
   return theStatus;
}
```

Make sure that the string is terminated with a return character. If you're using a **#define** to allocate static space for the string (which is not recommended), remember that it allocates the string plus a null terminator; **sizeof** then returns the size of the string, so you need to subtract 1 from the total. This string should come from a resource or a file.

If you want to add to the header from an application (to avoid writing the extension), you can add an item of type 'post' to the job collection, using the tag gxPrintingTagID. If the first character of this item is a % character, it will appear in the job header.

Q *Our application has multiple QuickDraw GX shapes layered on top of each other. The bottom object is a graphic, and the objects on top of it are text shapes. The text objects are transparent, permitting the underlying graphic to show through. Are there functions in QuickDraw GX to facilitate refreshing the background shapes when characters are deleted in the text layout shapes above it? We need to refresh the graphic with minimal flicker and want to avoid resorting to the standard CopyBits routing.*

A QuickDraw GX doesn't have any direct functions to facilitate refreshing or redrawing only a portion of a shape covered by another shape. However, there are a few methods that can be used in conjunction with various QuickDraw GX and QuickDraw calls to accomplish your goals. Here are three approaches that might work for you:

- As you know, you can have QuickDraw GX draw directly into a GWorld, and use CopyBits to update the appropriate area. This approach is good if you need to draw QuickDraw and QuickDraw GX objects in the same window.
- If you merge multiple shapes into a QuickDraw GX picture, you can use the picture's clip shape to update the area in question. Make your graphic shape the bottom shape in the picture's hierarchy. This forces QuickDraw GX to draw the graphic as the first shape, with the other shapes drawn on top. QuickDraw GX pictures are smart, in the sense that they respect the clip shapes associated with the picture and all of the shapes contained within the picture.

  To update the smallest possible area, convert the QuickDraw update region to a QuickDraw GX path. Then get the current clip shape of the picture with GXGetShapeClip, and save it for later restoring. Use the path as the "new" clip shape of the picture and draw. Finally, restore the picture's clip shape.
- Create a QuickDraw GX offscreen bitmap to perform flicker-free updating in a manner similar to using CopyBits. This method, though, is based completely on QuickDraw GX. When updating the screen, clip your drawing to the area you want to update. For an example, see the "3 circles - hit testing" sample that ships with QuickDraw GX.

Q *I'm using a layout shape to represent an area for editable text that will have a fixed position, style, font, size, and width. This layout shape has some default text that the user is prompted to change (text content only, no other attributes). Each time text is added (the new text replaces the previous text string), the user interface code checks whether the size of the new string goes beyond the defined width. I do this by comparing the width of the local bounds with the width given within the layout shape geometry. In all cases, the justification setting is 0, but the flush setting varies (left/0, center/0.5, right/1.0).*

Sometimes the width of the local bounds reaches a point where it's wider than the width defined by the shape; in other cases, it approaches the width but never reaches or surpasses it. In this situation, the text is updated and begins to compress itself within the defined width. How can I allow text to be entered till the width is reached, but not compressed?

A The problem you describe was fixed in QuickDraw GX 1.1.1 with a new API call:

```
Fixed GXGetLayoutJustificationGap (gxShape layout);
```

This function returns information that was always generated during layout's justification processing but was never made publicly visible before. It represents the signed difference between the specified width for the layout and the measured (unjustified) width.

By setting a width in the layout options, but leaving the justification factor at 0, you can keep adding text until the results of the GXGetLayoutJustificationGap call changes sign from positive to negative. At that point, the text starts to compress, so you should prohibit new text entry. It's a very fast call (since its result is cached as part of the layout process anyway), so calling it on every typed character shouldn't slow things down at all.

Some examples may help clarify the use of this call: Suppose you create a layout with the width field of the gxLayoutOptions set to 500 points and the justification factor set to **fract1** (full justification). If the unjustified width of the layout is only 450 points, GXGetLayoutJustificationGap returns +50 points; if the unjustified width is 525 points, this function returns -25 points. A positive value means the line will be typographically stretched to fill the specified width, while a negative value means the line will be typographically condensed.

Note that the justification factor in the gxLayoutOptions doesn't have to be **fract1** in order for this function to return useful results. For instance, if you set a width value but leave the justification factor at 0, the line will not be justified unless its unjustified width exceeds the specified width. In this case, layout will typographically shrink the line. A client program that wants to determine when the end of a line is reached (for line-breaking purposes) can call this function after every character is added (as the user types, for example); as soon as the value becomes negative, the client knows that the margin has been reached.

Q *There are three options on the General panel of the QuickDraw GX Print dialog — Collate Copies, Paper Feed, and Quality — that we would like to move to one of our own panels. We have solutions that differ from the default ones, and we want to rename these solutions and associate them with our printer. How can I eliminate those options from the General panel?*

A There's no mechanism in QuickDraw GX to remove panel items from the standard Print panels, except for the Quality item. The Quality collection item (gxQualityTag = 'qual'), whose structure is defined in PrintingManager.h, has a Boolean field called disableQuality. To eliminate the Quality item from the panel, specify true for the disableQuality field in your driver. Although you cannot remove the other items, you can disable them (dim them in the panel) by getting the collection item and setting the locked attribute with SetCollectionItemInfo.

Q *Do I need to call GXCloneColorProfile before calling GXConvertColor? Since the color passed into GXConvertColor by ColorSync is destroyed, should the color profile passed in as part of the color be disposed of? If not, isn't that a memory leak?*

A Calling GXCloneColorProfile isn't necessary, and it would require additional work that doesn't need to be done. The gxColor structure is a public data structure, not an object: the application, not QuickDraw GX, handles adding and maintaining references to objects with respect to gxColors (and gxBitmaps). QuickDraw GX maintains owner counts when the profile is attached to another QuickDraw GX object (using GXNewBitmap, GXSetInkColor, and so on). This is not a memory leak.

For example, consider this scenario: When an application gets a shape's color, the ink's profile has two owners — the shape and the application. Therefore, the application can reference the profile in gxColor structures, even if the shape is disposed of. Once the application calls GXDisposeColorProfile, the reference is no longer valid. Cloning the color profile does nothing except to require that GXDisposeColorProfile be called afterward. As a result, all that happens is that time is wasted as the owner count goes from a positive number to that number plus 1, and then back down.

Q *Does QuickDraw GX send the GXDoesPaperFit message when I'm setting up input tray dialogs, or is the driver supposed to do this? If QuickDraw GX doesn't, it's possible for users to request completely invalid paper sizes, which can violently crash most raster drivers.*

A QuickDraw GX sends the GXDoesPaperFit message in the default implementation of the input trays dialog to constrain the configuration options, and drivers that perform their own input trays dialog should do the same. A driver should override this message if it needs other than the default logic, which responds that everything fits.

The packing buffer size specified in the 'rpck' resource is set to the expected maximum size needed. Unfortunately, this is far smaller than what's needed when handling larger than expected paper sizes. To work around this, you can set the packing buffer size so that it can accommodate the largest paper size the printer can use.

Q *I've been experimenting to see what happens when a print job is canceled part of the way through. If I cancel when GXOpenConnection and GXStartSendPage have both completed successfully, I get unexpected GXCleanupOpenConnection and GXCleanupStartSendPage messages. If I cancel at another point in the job (for example, during rendering via the Remove button in the desktop printer status window), GXCleanupStartSendPage and GXCleanupOpenConnection messages are passed through after ImageDocument exits. This behavior seems very odd, and it doesn't appear to be discussed anywhere in the documentation. Shouldn't GXCleanupOpenConnection and GXCleanupStartSendPage be called only if their respective routines return an error?*

A The unexpected GXCleanupOpenConnection and GXCleanupStartSendPage messages are coming from the default implementations of ImageJob and ImagePage. The ImageJob code calls Send_GXSetupImageData, and if an error occurs, it sends GXCleanupOpenConnection. ImagePage calls Send_GXRenderPage and sends GXCleanupStartSendPage if an error occurs. If GXStartSendPage or GXOpenConnection doesn't complete successfully, the respective cleanup calls are not sent. Although the documentation states otherwise, this behavior is correct.

Q *Is the layout of the PostScript printer preferences ('pdip') resource documented correctly in Inside Macintosh: QuickDraw GX Printing Extensions and Drivers?*

A No. There's a bug in the documentation for the 'pdip' resource on page 6-88 of *Inside Macintosh: QuickDraw GX Printing Extensions and Drivers*. The render options field is in fact a long word. The resource is defined correctly in the interfaces (PrintingRezTypes.r) and in the MPW 411 files for QuickDraw GX.

Q *Is there a simple way to detach a QuickTime movie from its original file? I'm trying to place a copy of a QuickTime movie in my application's resource fork.*

A See John Wang's QuickTime column in *develop* Issue 17. You can use the technique presented in that column to extract a movie and put it into the resource fork of a different file. You'll find the column and the accompanying sample code, MultipleMovies, on this issue's CD.

Q *How can I determine whether QuickTime 2.0's MIDI music function is available and whether the larger set of 41 instruments is available? If the MIDI function is available, we need to add code to enable the music portion of our game.*

A The QuickTime Music Architecture became available in QuickTime 2.0 (as described in David Van Brink's article in this issue of *develop*), so checking the QuickTime version in a Gestalt call (selector gestaltQuickTimeVersion) will tell you if the MIDI function is present.

When the QuickTime Musical Instruments Extension is installed in your System Folder, it gives you the musical instruments supported by Apple. This extension is actually a component. If you need to know whether the instruments are present, call FindNextComponent, searching for a component that has a type of 'inst' and a subtype of 'ss '. Here's a code snippet:

```
pascal Boolean AreQuickTimeMusicInstrumentsPresent(void)
{
   ComponentDescription   aCD;

   aCD.componentType = 'inst';
   aCD.componentSubType = 'ss  ';
   aCD.componentManufacturer = 'appl';

   if (FindNextComponent((Component)0, &aCD) != NULL)
      return true;
   else
      return false;
}
```

Q *Are there any known compatibility problems between QuickTime 2.0 and QuickTime for Windows? I'm creating a dual-platform application and want to use QuickTime 2.0 for the video. Is there anything that I should avoid on either platform, or anything I should watch out for?*

A In most cases, you don't have to be concerned about using the same movie for playback on both platforms, as long as the movie is in a flattened format and in a single-fork file. To be sure your movie files are single-fork files, select "Make cross-platform" in the MoviePlayer application when saving your movies (or do something analogous in other applications that produce cross-platform movies).

QuickTime supports sound, video, text, music (MIDI), and MPEG tracks under both Windows and the Mac OS. One difference between the two versions is that you can have only one of each track open under Windows (except for the number of sound tracks; starting with QuickTime for Windows 2.0.1, you can enable/disable multiple sound tracks).

The biggest difference between the two versions is the API: QuickTime for Windows 2.0 doesn't support all the API calls available under the Mac OS. Nearly all of the movie controller APIs are supported, as well as many of the basic calls, but the calls to create manipulable movie tracks are missing. You can't create specific media handlers with QuickTime for Windows 2.0, but you can write data handlers and codecs for the Windows environment.

While working with QuickTime for Windows, you'll have to keep track of all the possible configuration issues that users might encounter. We distribute

README files with the latest information about compatibility and configurations (video/sound cards, drivers, and so on).

For additional information, the Mac OS Software Developer's Kit includes detailed documentation regarding API and architecture issues concerning QuickTime and QuickTime for Windows. Also see *How to Digitize Video* by Weiskamp and Johnson (Wiley Press) for another good source of information regarding the practical issues of both QuickTime and AVI movie creation. Although this book is a bit out of date in the details (it was written to cover QuickTime 1.6.1 and QuickTime for Windows 1.1.1), much of it is still valid.

Q *Can we use a different A5 world with QuickTime? Our plug-in architecture uses A5 for global access, but we allow the A5 world to move. QuickTime doesn't seem to be able to deal with this, and it doesn't realize that EnterMovies was called once the A5 world moves. We currently work around this by locking down our A5 world, but we would rather not do this. If we need to keep doing this, is locking down the A5 world an adequate fix, and can you recommend another solution?*

A You can use a different A5 world with QuickTime, but whatever A5 world you use, you'll have to lock it down. QuickTime allocates a new set of state variables for each A5 world that's active when EnterMovies is called. However, since QuickTime uses A5 to identify each QuickTime client, if A5 changes (your A5 world moves), QuickTime won't recognize that you've called EnterMovies for that client.

Q *How do I determine the correct time values to pass to GetMoviePict to get all the sequential frames of a QuickTime movie?*

A The best way to determine the correct time to pass to get movie frames is to call the GetMovieNextInterestingTime routine repeatedly. The first time you call GetMovieNextInterestingTime, its flags parameter should have a value of nextTimeMediaSample + nextTimeEdgeOK to get the first frame. For subsequent calls, the value of flags should be nextTimeMediaSample, and the whichMediaTypes parameter should be VisualMediaCharacteristic ('eyes') to include only tracks with visual information.

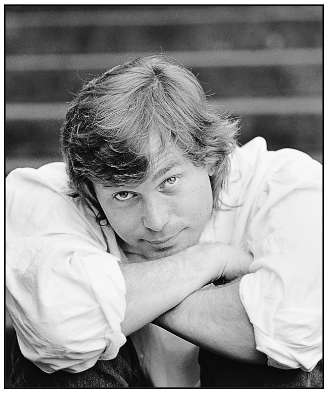
Q *I noticed that certain commercial candies have no taste when they initially hit my tongue. It's only after I start sucking them that the flavor appears. I think there's some sort of coating on them. What is it? Is it harmful?*

Also, what is it that creates that beautiful high gloss I get with my car wax and floor wax? I just love the way it shines after a good hard buffing.

A Carnauba wax is the answer, in both cases. It's a hard wax obtained from the leaves of a Brazilian palm tree (*Copernicia prunifera*), and is used a lot in polishes of all types. It really does buff up beautifully, doesn't it? It also is completely tasteless and nontoxic, and makes a dandy confectioner's glaze, used to keep the candy from sticking to itself.

These answers are supplied by the technical gurus in Apple's Developer Support Center.•

Have more questions? See the new Macintosh Technical Q&As on this issue's CD. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)•



DAVE JOHNSON

THE VETERAN NEOPHYTE

A Feel for the Thing

I used to think there was no room for mystery in the world of computers. I didn't think there was any use for fudge factors or rules of thumb or hunches in the clean, exact, hermetically sealed bubble of logic we all spend so much time diddling and poking. That stuff belongs to "real world" engineering, not software engineering, right? Software is always bounded and orderly, always understood completely from top to bottom, with no dangling ends, no frayed edges, and no baling wire and duct tape holding things together. There's never a need for vague, hand-waving explanations of how it all works, because we know how it works.

That's what I used to think. I'm not so sure anymore.

Ultimately, of course, the operation of computers is deterministic and absolutely predictable. There's guaranteed to be a complete explanation for any event on the computer; the search for an answer will always find one. It's like playing Go Fish with a deck of cards that contains only threes — "Got any threes?" "Yep." "Got any threes?" "Yep." "Got any threes?" "Yep." The answer itself, of course, may be convoluted and difficult, and is often *way* too much trouble to actually track down ("Have you tried rebooting?"), but it's always there. The world inside computers has a definite, impermeable bottom, like a swimming pool.

The real world, on the other hand, is more like being out in the middle of the ocean: the bottom is nowhere in sight, and in fact is so far away that it may as well not exist at all. Trying to completely explain things in the real world is generally an exercise in futility, though one that humans seem to have a capacious appetite for (that's what science is all about, after all). The real world is so vast and complex that our explanations are never really complete. The answers always lead to more questions, and the edges of our knowledge remain frayed and ragged and crumbling, even though the center may have a seemingly solid, well supported integrity.

The thing that got me thinking about all this is boomerangs. I've been learning to throw boomerangs lately, and it's extremely satisfying — and somehow endlessly novel — to throw something away from yourself as hard as you can, and have it return several seconds later, hovering gently down into your waiting hands like a bird coming home to roost. (Such a perfect flight, of course, is a rare thing for a novice like me. More often, if the boomerang comes anywhere near me, it's slicing past at a frightening rate of speed while I cringe, covering my head.) While I've been learning to throw boomerangs, I've also been trying to watch myself learn to throw boomerangs — sort of meta-boomeranging — and I noticed that a complete explanation of what was happening was not only absent, but completely unnecessary: I don't need to know how boomerangs work to learn to throw them well.

Boomerang throwing is one of those real-world activities — there are many of them — that are governed by rules of thumb, by approximation and estimation, and by "feel." There are lots of variables involved in producing a good boomerang flight, and they're all sort of woven together, interconnected and interdependent. The direction of the throw, the angle of the boomerang as it leaves your hand, the forward power of the throw, and the amount of spin all contribute to the flight characteristics, but the way they combine and interact is complex and nonobvious. How's a poor, bewildered boomerang neophyte to make any sense of it all?

Well, the only way to learn to throw boomerangs is to get yourself a decent boomerang (very important!), read a little about it or get a lesson from someone, and then just get out there and start throwing. You need to experience it; you need to feel the smooth, flat weight of the thing, notice the way it slices the wind as it leaves your hand, and watch as it spins and swoops. Every throw you make adds to a growing store of knowledge about boomerang behavior. Slowly, you begin to sense the structure of the rules that govern the flight of the boomerang, to get a feel for it, to gain some control. But no matter how long you work at it, there's always more you can learn about boomerangs. Boomerang throwing, like most things in the real world, has no bottom.

But even though things in the real world are webby, tangled, and complex, with no real bottom and no real

DAVE JOHNSON has an ever-lengthening list of life goals, things that he'd like to accomplish or experience before leaving this mortal coil. Some recent additions include making marshmallows from scratch, milking a cow, and hugging a full-grown bear. (Is bear breath better than dog breath? There's only one way to find out!) If you have a cow or bear Dave could visit, please let him know.•

center, and even though complete understanding is out of our reach, that doesn't stop us from getting things done. Even though we may not understand exactly what's going on when we throw a boomerang, we can learn to throw them anyway, and can actually learn to throw them with incredible skill. Scientists don't have a complete understanding of fluid mechanics, but we can still design hydraulic lifts that lift, toilets that flush, and airplanes that fly.

Though it seemed profound when I first thought of it that way, it really isn't anything remarkable at all. It's the stuff our everyday sensory world is made of. It's our standard, animal mode of operation. We depend heavily on trial and error, on finding and keeping strategies that work. We invent myths and superstitions to explain things we don't understand, we guess, we fake it, we operate by feel. And it works just fine.

But we don't need that sort of thing in the clean, deterministic world of computers, right? If we know the answer is within our reach, then why gloss over it? There's one very good reason: it's gotten to the point where it's often really hard to reach the answer. Computers have become so complex that finding the real answer is often a Herculean feat requiring great effort and stamina. The things that we're "growing" in the machine are getting very deep and webby and complex, just like things in the real world. That nice smooth bottom we all know and love is getting pretty remote and hard to see, and in fact trying to keep it in sight often holds us back.

The truth is, we *need* fakery, or myth, or something similar, to avoid being hopelessly mired in complexity, and to let us feel cozy even in the face of something too deep to comfortably understand. The idea that an icon in the Finder, a document window in an application, and a file on the hard disk are all "the same thing" is a fiction, an illusion created from smoke and mirrors, and one that users don't even think about anymore (unless, of course, an application screws up the illusion; see Mark Linton's article in this issue for some code to help you avoid such a faux pas). But it's precisely that kind of myth and abstraction that lets people ignore all the underlying complexity and just go about their everyday business. Without that kind of trickery most people would be lost.

Humans have a deep need for some sort of explanation, and we'll often ignore aspects of a situation, or even make stuff up out of thin air, if it helps us to find an "answer." Remember the frictionless inclined planes and perfect vacuums of college physics? Without that kind of glossing over of details, we'd have been helpless. (A college housemate of mine and I used to joke about running a college physics stockroom: boxes of frictionless, massless pulleys on the shelves; gallon jugs of zero-viscosity liquid at our feet; coils of infinite and semi-infinite wires hanging neatly on the pegboard wall. Those wires have no thickness or mass, thank goodness, or the storage requirements would be prohibitive.) This need for explanation is what has led us to science, and to religion, and to superstition. These are not the same thing, of course, but they can all serve the same purpose: a soothing, protective balm on the raw edges of our incomplete knowledge. They give us a ground to stand on, a rail to hold on to, as we totter along in the darkness, going who knows where, hoping the batteries will hold out long enough to get an answer.

Now that I think about it, I'm happiest with a generous helping of myth and fiction stirred into my computing. It can help make the computer — which, let's face it, is essentially a gritty, sharp-edged, and hostile machine — feel more rounded and friendly. It can provide a useful disguise, like a plastic nose and glasses on something seething and alien, making it recognizable, familiar, even comforting and amusing. If it's done well, it can even let me learn to use a computer in much the same way I learn to throw a boomerang: by picking it up and trying it, by mucking around and getting a feel for it, by discovery.

Maybe best of all, it lets computers keep a little of their mystery. The mystery and magic of the Macintosh are why many of us are programmers, after all. Mysterious things, things that don't have clean and obvious boundaries, are inevitably more interesting and more fun. There's no denying that computers have a dull, featureless, dreary bottom. But in the other direction there seems to be no boundary; the top, if there is one, is as far away as the sky. So yes, I think there's plenty of room for mystery in the world of computing. Plenty of room indeed.

RECOMMENDED READING

- *Many Happy Returns: The Art and Sport of Boomeranging* by Benjamin Ruhe (Viking, 1977).
- *How to Hide Almost Anything* by David Krotz (William Morrow and Company, 1975).

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.•

Newton Q & A: Ask the Llama

Q *I have a program that communicates with the desktop. Part of the information sent is real numbers. I've found functions to stuff almost every other type of data into a binary object except real numbers. How do I do that?*

A You have two choices. First, you could just print the real number as a string (using SPrintObject), send the string, and convert it back on the other side. Clearly this isn't a good idea if you want to maintain a high degree of precision. The other choice is to construct the correct type of binary object for the target desktop machine. In other words, take the Newton real representation and convert it into, say, IEEE floating point. Then you can use BinaryMunger to stuff the binary object into whatever packet of data you're constructing.

Note that Newton uses SANE representation for real numbers that are in the representable range. However, the representation of exceptions (such as NAN and infinity) are different and undocumented. At this time you should avoid converting these types of real numbers.

Q *Can you give me a short and clear description of the different types of Newton memory?*

A There are three important "pools" of so-called internal memory, each with different tradeoffs.

The NewtonScript heap (about 90K to 96K on current devices) is where all the runtime data from NewtonScript lives. Any result from the Clone family of calls will take up NewtonScript heap space. The view frame made at run time from your application templates will take up this heap space. NewtonScript heap space is very precious, so you should try to use as little of it as possible, especially when your application's base view isn't open.

The user store (192K in the MessagePad 100, larger on other devices) is where application packages stored internally live, and where soups are located. The entries in the soups are located in this space. While not quite as precious as the NewtonScript heap, this space can certainly run out. This is the space that's "extended" when a RAM PCMCIA card is inserted.

There is also some system heap space, which is used for, well, everything else. The viewCObjects and drawing objects live here. Recognition uses memory from here. You can run out of this space (in which case you get the Cancel/Restart dialog) but it's less of a programming issue.

Q *I have an application that uses a protoRollBrowser. When I expand the items, they have lines separating them. I can't seem to get rid of them. Is this a bug?*

A What you're seeing is part of the default definition of a protoRollItem. It includes a 1-pixel border. You can remove that border by modifying the viewFormat of your rollItems. In addition, you may want to set the fill to white.

Q *I'm using a protoRoll (not protoRollBrowser) in my application. But it never shows up. What's the problem?*

The llama is the unofficial mascot of Developer Technical Support in Apple's Newton Systems Group. 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.•

A You need to give it a viewFlags slot and make sure the Visible bit is checked. The default is Application and Clipping, but this won't make the protoRoll visible if it's included inside another view.

Q *I have a text view that the user can use to enter text. I wanted to extend a selection. I knew the insertion caret was at the end of the selection, so I called SetHilite(newPoint, newPoint, nil), where newPoint is the new position for the selection extension, but I got no highlight. What's going wrong?*

A The behavior is actually perfectly correct. There's a not quite obvious interaction between the caret and SetHilite. As shown in the table below, how SetHilite behaves depends on four things: the **start** and **end** character positions (the first two arguments) being equal, the value of **unique** (the third argument), the presence of a previously highlighted selection, and the presence of the caret. Note that the following explanation refers to the case of a single paragraph view, in which there can be only one selection; if there are multiple paragraph views, it's possible (with **unique** nil) to have multiple discontiguous selections.

Highlight and unique	**When start = end**	**When start <> end**
No previous highlight, **unique** true or nil	If there's a caret, move caret; otherwise, no effect	Create new highlight from **start** to **end**
Previous highlight, **unique** true	Clear highlight and, if there's a caret, move caret	Create new highlight from **start** to **end** (remove old highlight)
Previous highlight, **unique** nil	Extend highlight to include **start/end**	Extend highlight to include **start/end**

Q *I have an application that uses ADSP to connect to a server on the desktop. I want the server to handle multiple Newtons connected simultaneously. Unfortunately, if a connection fails after it's opened, the server doesn't seem to be able to identify it as a new connection when the Newton reconnects. This causes problems in the server's ability to handle multiple connections. Can you help?*

A We'll assume that the Newton tries to reconnect shortly after losing the connection. In that case, the Newton doesn't generate a new connection ID, so your server probably acts as if the connection didn't close, while the Newton is acting as if it's establishing a new connection. Currently the only solution is to force the Newton to wait three minutes after an improper disconnect before trying to reconnect.

Q *I have a communications program that always sends a string of the same size to the desktop. The string is quite large, and I would like to preallocate it and fill it with a particular value. What's the best way to do this?*

A As with all things in programming, the answer is a tradeoff between space and time. Let's assume that you want a string of 2K characters filled with the character A, and that you control the contents of the string (that is, if you get user input, you make sure the input is a string). The first option is to allocate the string at compile time. Note that you shouldn't allocate your string constant with a double-quoted string ("a string"), since typing 2K (less the terminator) characters is monotonous and error prone. The way to allocate the string is with the following SetLength trick:

```
constant kNumberOfUnicodeCharsForString := 2048;  // 2K chars
DefConst('kMyBigString, call func()
   begin
      // SetLength uses bytes; Unicode chars are 2 bytes each
      local aStr := SetLength("",
            2 * kNumberOfUnicodeCharsForString + 2);
      // initialize the string
      for i := 0 to k1KUnicodeChars - 1 do
         aStr[i] := $A;
      return aStr;
   end with ());
```

At run time you can clone kMyBigString and do what you need to fill it with characters. Note that the object is not a string; you would need to use StuffByte to put in individual characters.

The advantage of this method is that it's very fast: it averages less than one tick (60th of a second) for the clone. The disadvantage is that it puts a 4K object in your package (Unicode strings are two bytes per character). If you can't afford the 4K in your package, you need to generate the string at run time. Using the above code at run time averages 52 ticks.

Another possible runtime method is to use smart strings, which allow you to preallocate strings and concatenate them in a more efficient way. The first attempt at doing this seems to be inefficient, at an average of 175 ticks:

```
// defined constant somewhere in your project
constant  kNumberOfUnicodeCharsForString := 2048;

local s := SmartStart(2 * kNumberOfUnicodeCharsForString + 2);
local l := 0;
for i := 1 to kNumberOfUnicodeCharsForString do
   l := SmartConcat(s, l, "A");
SmartStop(s, l);
```

However, simply concatenating two characters at a time reduces the average to 88 ticks; four characters reduces it to 44; and so on. A lesson here is that testing and measurement are your friends.

Q *I'd like to train my dog to code in NewtonScript. How can I do that?*

A I'm afraid the prospect isn't promising. Dr. J. L. Fredericks at SITAP (Stanford Institute for Training Animal Programmers) has been trying for ten years to train different animal species to program computers. Although he's had some success training dogs to do simple programs, he says, "Anything more than a simple statement is beyond them. No loops, no conditionals." Besides which, paws don't work well for moving mice. For Newton programming the best he has been able to achieve is training a rat to reset the Newton on command. As Dr. Fredericks says, "Never underestimate the usefulness of a ratset."

Thanks to our Newton Partners for the questions used in this column, and to jXopher Bell, Bob Ebert, David Fedor, Neil Rhodes, Jim Schram, Maurice Sharp, and Bruce Thompson for the answers. Thanks especially to Bob Ebert for the Newton memory description.•

Have more questions? Take a look at Newton Developer Info on AppleLink.•

KON & BAL'S PUZZLE PAGE

Video Nightmare

See if you can solve this programming puzzle, presented in the form of a dialog between a pseudo KON (Ian Hendry) and BAL (Eric Anderson). 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. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.

IAN HENDRY AND ERIC ANDERSON

BAL I've got one for you, KON: I just updated to System 7.5 on my 8100/80 AV. Everything seemed OK for a while. I was comparing Scenery Animator to Vistapro and I noticed that my cool new desktop pattern had disappeared. It was there when I booted, but just as the Finder was coming up, the desktop changed to a black-and-white pattern.

KON That's easy. Go back to System 7.1 and the world will be fine again. Next.

BAL Hey, 7.5 is the source of much wonderment. It's really a lot of fun!

KON I don't know that much about 7.5. Metrowerks and THINK C seem to run fine on 7.1. Is this part of that new puzzle CDEV that was added to spruce up the system?

BAL Quit trying to change the subject. My desktop pattern went away and I'm not happy about it.

KON Hmm. Did you change anything on your machine?

BAL I turned on VM for the memory-hungry rendering stuff.

KON So turn off VM and see if the problem goes away.

IAN HENDRY (AppleLink HENDRY; Internet hendry@apple.com) gets paid by Apple to work on video stuff. His hobbies include shipping products and collecting new Engineering managers. He can be found skipping meetings to play Ultimate and working all hours to make up for it. Ian's going to be a dad soon, and though he has been in rigorous sleep-deprivation training for years, he's hoping (but still not certain) that he's ready for what he's gotten himself into.•

ERIC ANDERSON (AppleLink ERIC3) skipped out on the OS Services group at Apple to get away from the chore of working on VM and the Thread Manager. Now he works as Ian's evil twin on video-related Mac OS issues — and he gets bugs assigned to him that state, "When using a multisync monitor with my threaded test app while VM is on, this funny thing happens." Seeing that there's no escape, Eric wants more than ever to move to Hawaii and build boats.•

BAL Wow, that worked great. Now all I have to do is buy $1800 worth of tariff-enhanced RAM so I can render my flyby of the Pentagon.

KON I won't ask what you're up to these days. My recent stock dealings have left me low on bail money. Did you try it on 7.1.2?

100 BAL It didn't happen on straight 7.1.2, but I installed System Update 3.0 and it happened. VM must have changed in this update.

KON Sounds like a VM problem all right. Paste the older VM resources into the new system. (I love component software.)

BAL The problem is still there. Does this let VM off the hook?

KON VM is never off the hook, but if your only problem is that the desktop pattern is black and white, maybe you should stop whining and do your work.

95 BAL No, this is more serious. I opened the Monitors control panel and there was no depth list and no monitor tile in the rearrange section.

KON Well, this should be pretty straightforward. Does it happen every time?

90 BAL No such luck. This one is really evil. I've been trying to get a reproducible case for days. Sometimes it happens right away, sometimes it goes away for hours. Once it starts happening, it seems to keep happening across restarts. It doesn't happen as reliably across shutdowns. It seems to happen more in millions of colors but will happen at other depths too. Switching the display back and forth between a 13-inch and a multiple-scan display sometimes causes the problem to show up. Changing VM and RAM disk settings seems to affect the reproducibility.

KON Cool! The random bugs are always the most fun. Let's get our trusty MacsBug and see if we can find where it's going bad. Look at the video driver and GDevice.

85 BAL When I try to enter MacsBug, the mouse freezes but MacsBug doesn't come up.

KON Dang, I hate it when the tools don't work.

BAL Perhaps we should devote this column to model rocketry and video games instead.

KON Now there's a thought. If I type Command-G, does the mouse unfreeze, or do I have to reboot?

BAL The machine comes back when you hit Command-G.

KON So MacsBug is there; you just can't see it. I'll use **log foo** to dump the output to a file named **foo**, followed by

```
dm @@thegdevice gdevice
```

Then I'll use **drvr** to see if the video driver is alive.

BAL Nice **log** trick, KON! The GDevice is fine, and the driver looks as if it's loaded and active.

KON Drivers and VM sometimes don't get along. Maybe the driver is doing something wrong. Did you try other video cards?

80 BAL It seems to happen only on Power Macintosh AV models.

KON It's the nasty "*fung*us" problem from Issue 17 all over again, or maybe your card has gone bad.

BAL Well, I'm pretty sure there was no "*fung*us" around when the AV card was developed, so it's probably not that. Besides, we're sticking to production software in this column from now on. Anyway, I thought it might be my card, too, so I borrowed your card. Thanks for the loaner. I turned on VM and it worked like a champ . . . for a while. Now I've got the same problem again.

KON You killed my card?

BAL Admittedly, it's a computer that can do anything. But for the purposes of this column, that idea is pretty far-fetched.

KON Here's what might be happening: Early in the boot process, VM isn't present. For each card, the system calls the card's PrimaryInit code and creates a GDevice. When VM loads, it changes the logical address mappings. When the driver is called again, it assumes a one-to-one logical-to-physical mapping of RAM, so the card starts responding to bogus address cycles. This confuses the card's bus translator, and . . .

75 BAL Whatever. Any other stabs in the dark? Lose five points and try again.

KON OK. Perhaps there are subtle timing variations when VM is present, and the video card might have borderline hardware that's affected by these timing dependencies. Or maybe the card's controller gets into a state where it no longer responds to its address space.

70 BAL You're getting desperate. It's not a hardware problem. The declaration ROM is there and everything looks fine. You can't blame this on the hardware. Let's once again follow the software decision tree.

KON Yeah, you're probably right. Now that I think about it, those ideas seem really out there.

So what you're telling me is that the desktop pattern is black and white, MacsBug isn't working, and the Monitors control panel doesn't show depths or a monitor tile. Let's find out when MacsBug stops working.

65 BAL When the machine boots, MacsBug is working, but by the time you get to the Finder, it's gone. It seems to vanish early in the boot process.

KON See if you get to the first Display Manager call with an **atb DisplayDispatch** or **atb ABEB**.

BAL OK. MacsBug is still alive.

KON I'll set a breakpoint just after the first Display Manager call and then **go**.

BAL Yep. There doesn't seem to be a problem now. But the weird thing is, if you trace over the Display Manager call and then type **go**, MacsBug will eventually go away.

KON Wacky. Sounds like a Display Manager bug.

BAL Earlier you said it was a VM bug.

KON Both have been convicted criminals in the past, so you can't blame me for thinking they're suspects. I'll bet you a buck it turns out to be neither! Do an **atb** on the Display Manager call and trace from there until MacsBug goes away; it shouldn't take too long.

60 BAL Sorry, MacsBug never goes away. The problem isn't reproduced.

KON So what you're telling me is that if I trace over the Display Manager call and then **go**, I can't get into MacsBug after I'm done booting. But if I keep tracing, the machine boots fine and MacsBug is always available.

BAL That's right. Let me help you along a little bit. There's an SSecondaryInit call (which runs SecondaryInit code for the video cards) just a few 680x0 instructions after the first Display Manager call. Does that help at all?

KON What happens if we do an **atb** on SSecondaryInit?

55 BAL I can't reproduce the problem. If I set a breakpoint just after the Display Manager call and **go**, the problem disappears. If I do an **atb** on the Display Manager call, and either **go** from there or trace over it and **go**, the problem happens. If I trace over it for a few instructions and **go**, the problem doesn't happen.

KON So, what are the "few" instructions? It looks like they're the ones whacking the video driver.

50 BAL No, they're just a few MOVE instructions to innocuous RAM locations — nothing that should touch video.

KON What does this Display Manager call do? Could it be hosing anything in the slots?

45 BAL I don't think so. I used MacsBug to skip it entirely and the problem still happens.

KON This isn't getting us anywhere. Maybe the desktop pattern problem has some better clues. The General Controls control panel in System 7.5 has an INIT resource that calls HasDepth to decide whether to use the color pattern. It then sets a PRAM bit to remember whether to use a color pattern across restarts. When the desktop pattern is black and white, I'll use the **log** trick to find out what the HasDepth call is returning.

40 BAL It returns an error.

KON Aha! Since HasDepth returns an error, the INIT resource thinks it's on a display that can support only one bit per pixel (black and white), so it disables the color desktop pattern and resets the PRAM bit; the color desktop pattern is now gone forever.

BAL OK.

KON Let's trace HasDepth and find out what's wrong.

35 BAL It looks as if the Slot Manager returns the correct values for the active functional sResource of the card but fails to find the depth. It returns -316, an smInitStatVErr. According to Errors.h, this error indicates that the siInitStatusV field was "negative after primary or secondary init." This means the card's PrimaryInit or SecondaryInit code returned an error.

KON We can bet it's not PrimaryInit, because the GDevice is good. If the error had happened in PrimaryInit, QuickDraw would have gotten an smInitStatVErr when it called the Slot Manager to build the GDevice.

30 BAL You're finally making some progress!

KON MacsBug also makes Slot Manager calls (when it tries to switch depths), which explains why it fails. That means the problem must be with the SSecondaryInit call. Once the Slot Manager gets this error, most Slot Manager calls return errors.

BAL But this doesn't explain what's causing the Slot Manager to fail to begin with, or why the problem goes away every time we get close to it with MacsBug.

KON Maybe we should try this with a BootBug card. Can you still get one?

BAL Maybe, but we're doing pretty well here. Let's keep going a little longer.

KON Let's try to figure it out by brute logic. What does the SecondaryInit code look like on this card?

25 BAL MOVE.L A0,A0.

KON That's it? Two bytes? No RTS? Cool! A bug in the AV card ROM! Does this mean we all get new cards with the new 2.0 ROM? Maybe they can simplify that complicated Monitors control panel Options dialog at the same time. How does it boot at all?

20 BAL Good question. *Designing Cards and Drivers for the Macintosh Family* says that the SecondaryInit entry on a video card is an SExecBlock, which is a header followed by actual code. The Slot Manager validates the header before it executes the code. The first byte of an SExecBlock is the revision number, and the Slot Manager checks for a revision byte of 0x02. Since MOVE.L A0,A0 is 0x2048 in hex, the first byte of the AV card's SecondaryInit entry is 0x20, which is a bogus entry, and the Slot Manager will never try to execute the SecondaryInit code.

KON So it's pretty lame, but it should work, right?

BAL Yes. Remember, we added SecondaryInit to the boot process because some machines didn't have 32-bit QuickDraw in ROM. On a machine without 32-bit QuickDraw in ROM, video cards have to disable their functional sResources with direct bit depths (16 and 32) in their PrimaryInit code, because the PrimaryInit code runs before the disk is up and the cards can't tell if the system has 32-bit QuickDraw installed. SecondaryInit was added to give these cards a chance to reenable those direct depths after 32-bit QuickDraw was loaded from disk. Power Macs obviously have 32-bit QuickDraw in ROM, and this card only runs on a Power Mac, so it doesn't need SecondaryInit.

KON Let's walk through the SSecondaryInit call and see what it does. Why does VM make a difference? And why is MacsBug causing the problem to go away if you set breakpoints?

BAL You're just full of questions, aren't you? You're supposed to be giving the answers!

KON Let's walk through SSecondaryInit.

15 BAL For each card, it looks for a SecondaryInit entry in the card's ROM. The entry contains a header followed by the SecondaryInit code. If there's no SecondaryInit entry on the card, SSecondaryInit bails out early. If there is a SecondaryInit on the card, the Slot Manager tries to execute it with SExec and then checks the status from the SExec call. If the status is negative (an error), the Slot Manager marks the slot with that evil -316 error, and the slot is bad from there on out.

KON So who is responsible for setting the error?

10 BAL The code executed by SExec, in this case the SecondaryInit code, should set the status error. If the header is bad, the code never gets run and the status never gets set.

KON Let me guess: the boot code never initializes the status before calling SExec.

5 BAL Yep. And it's allocated on the stack as a local variable, which means that the status is set to whatever garbage is left on the stack. At this point in the boot process you're still in supervisor mode, so MacsBug is sharing your stack. When MacsBug is used, it pushes stuff onto the stack and then pops it off when it leaves (changing the garbage below the stack in the process). That's why setting breakpoints and tracing mask the problem. BootBug also uses the stack, so it too would have interfered with the bug.

Between the first Display Manager call and the SSecondaryInit, the system allocates stack space for the SPBlock parameter for the SSecondaryInit call. After the SPBlock is allocated, the stack pointer is very close to where the local variables for SSecondaryInit will be allocated. At this point MacsBug's stack usage will affect those never-initialized local variables.

This is something else to add to your list of gotchas for MacsBug: If you're in supervisor mode (as you are at this point in the boot process) and you set breakpoints, MacsBug is sharing your stack, and its use of the stack may affect uninitialized variables. Later in the boot process, VM switches the machine to user mode; from then on, MacsBug and applications use different stacks and MacsBug will not interfere with uninitialized variables on the stack.

KON The garbage that VM leaves on the stack (sometimes) happens to be negative. When the boot code gets to SecondaryInit and allocates variables on the stack, it happens to use an area of the stack affected by VM.

BAL Well, I never turn VM on, so I'm always in supervisor mode, and MacsBug always shares my stack. But now I've finally found a good use for VM: turn it on when I have a bug that's hard to reproduce when MacsBug gets involved, and see if it becomes reproducible.

KON That'll slow your machine down.

BAL Nasty.

KON Yeah.

SCORING

75–100	Excellent; you probably have a video-in jack built right into your head.
50–70	Maybe we should be working for you.
25–45	Maybe you should be working for us.
5–20	Maybe you should stick to television.•

Thanks to Rich Collyer, Kent Miller, Mike Puckett, John Yen, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•