QuickDraw 3D Tips • Advanced PowerPC Performance Profiling • MPW ToolServer

develop

The Apple Technical Journal

Issue 24 December 1995

Speeding Up whose Clause Resolution in Your Scriptable Application

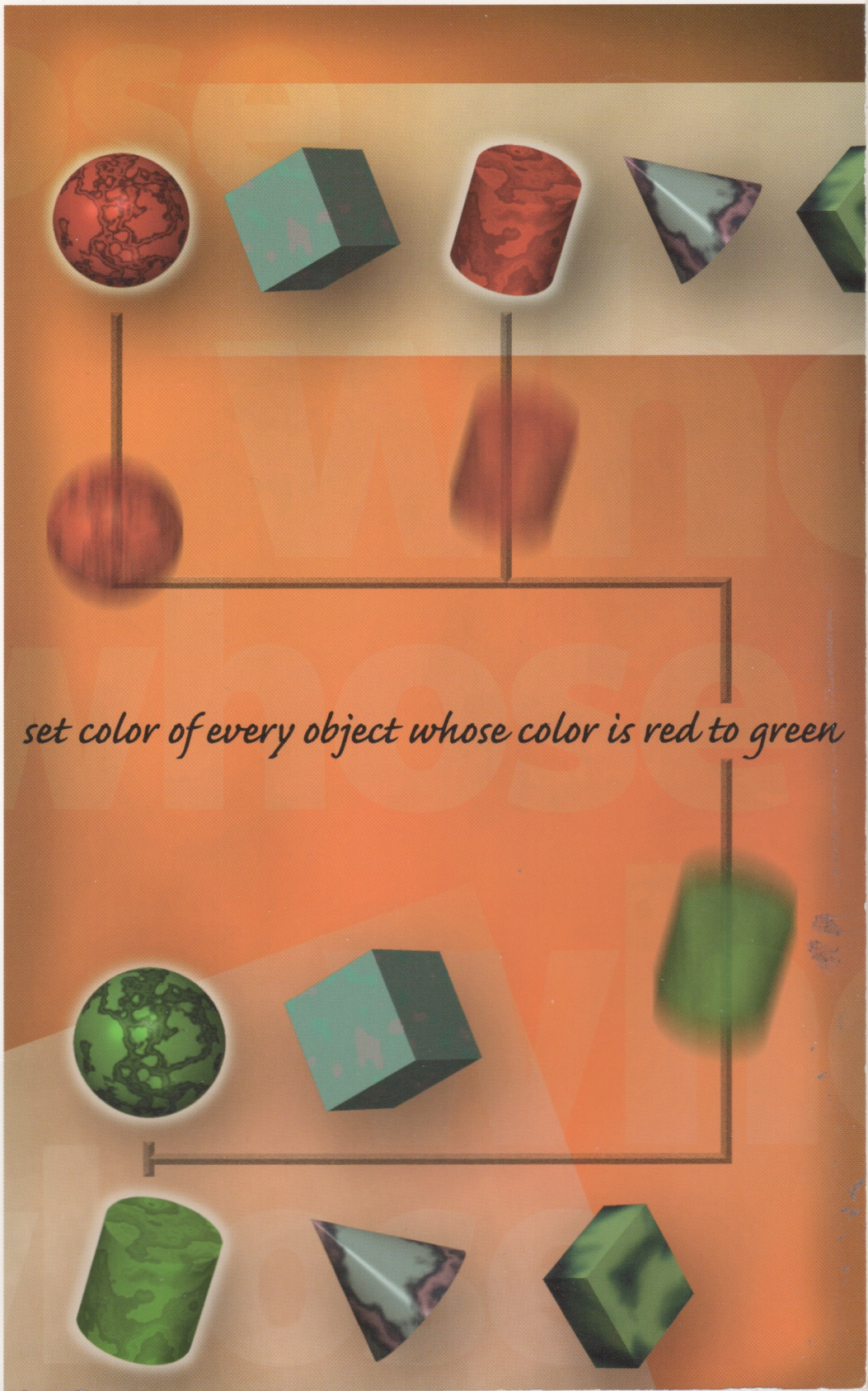
Getting Started With OpenDoc Storage

Sound Secrets

Guidelines for Effective Alerts

Printing Images Faster With Data Compression

The New Device Drivers: Memory Matters



$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.

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.

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.

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

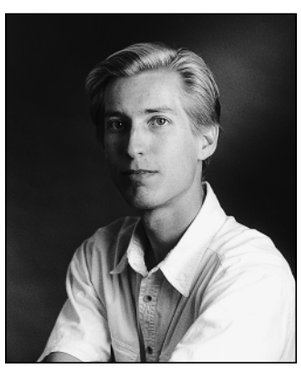
There's much more to the 10-bit model than you seem to realize. Have you ever had somebody tell you "You can eat as many Snickers bars as you can count on your hands"? Probably not. You wouldn't want to stop at 256, would you?

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,

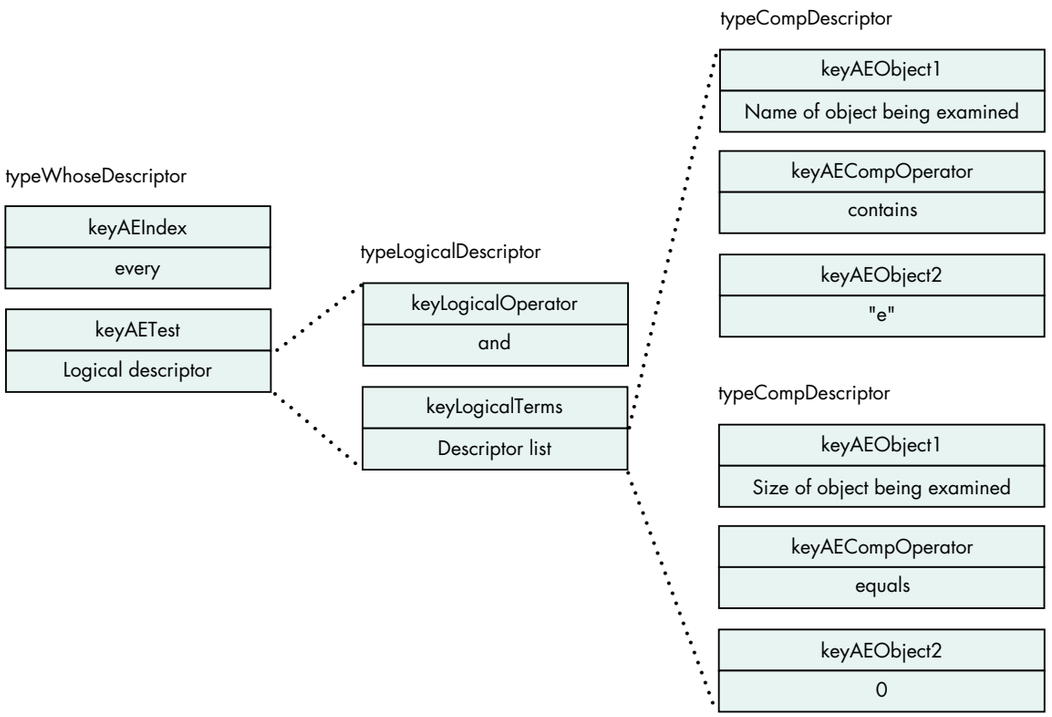


Figure 1. Contents of a **whose** descriptor

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();
}
```

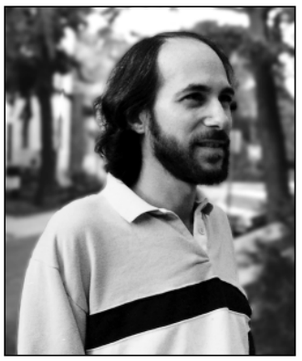
Given this definition for the Command method, some class derived from TBehavior could override the virtual method TBehavior::CommandDynamicBehavior, and call Inherited to execute the default action of the method it's overriding. This allows behaviors to do both pre- and post-processing. The cost to supporting 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.•



ACCORDING TO SCRIPT

Steps to Scriptability

CAL SIMONE

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.

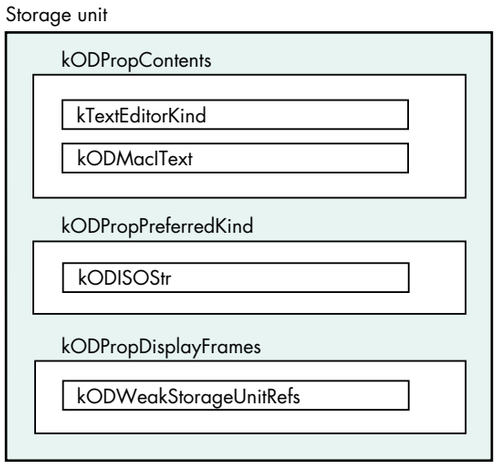


Figure 1. Structure of a storage unit

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.

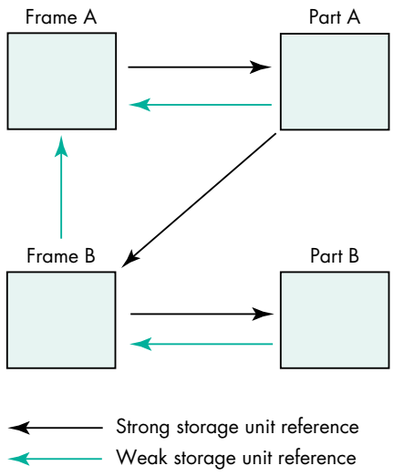


Figure 2. Strong and weak storage unit references

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

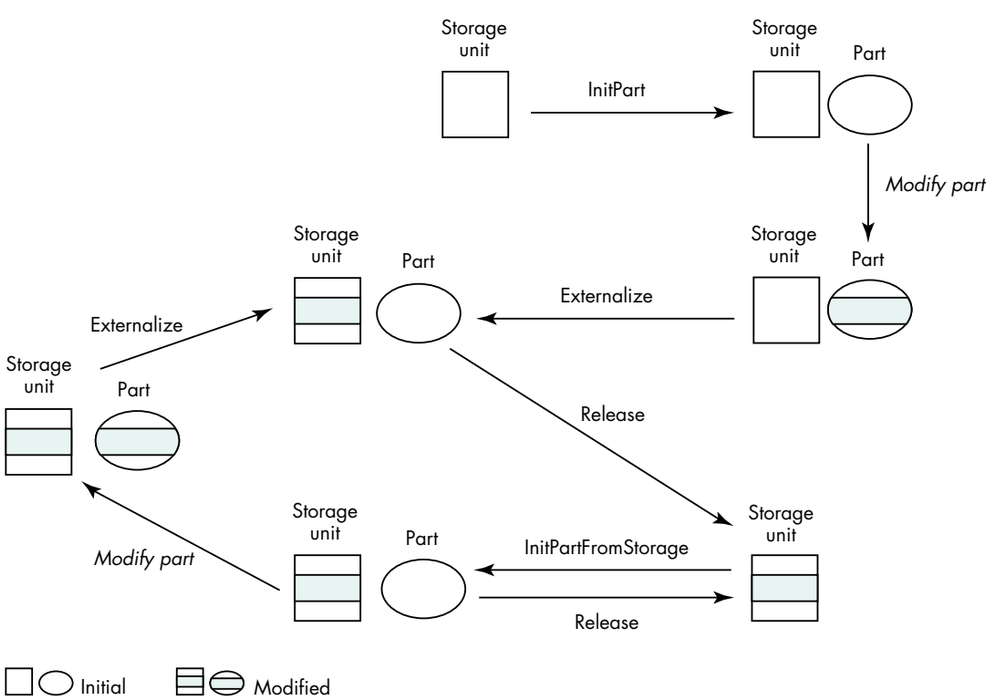


Figure 3. Life cycle of a part

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).

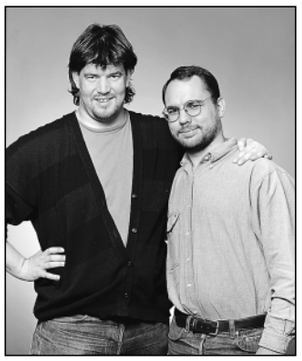
WHAT NEXT?

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);
```

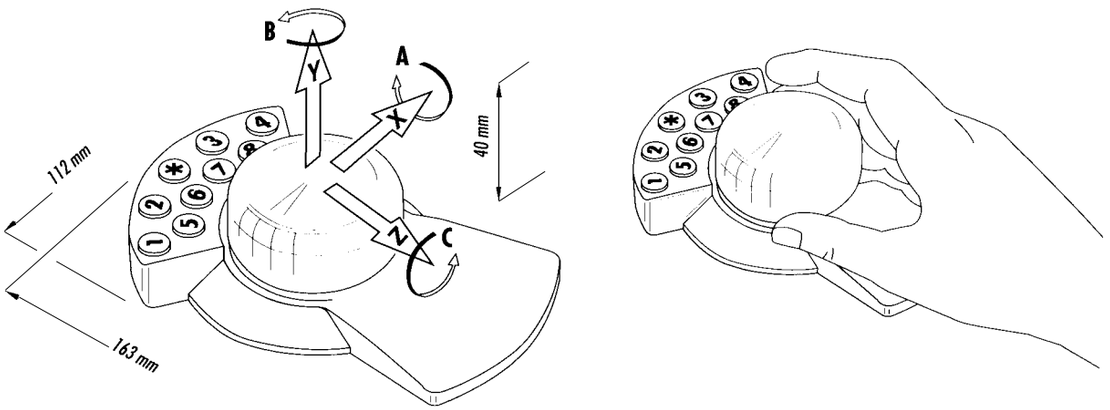


Figure 1. Magellan: a six-degrees-of-freedom input device (courtesy of Logitech)

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 Setting	Right Channel Decimal Value	Left Channel 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.

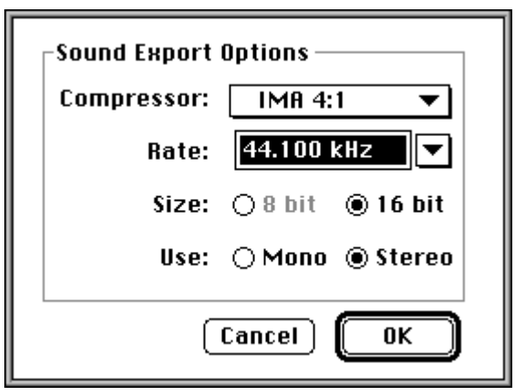


Figure 1. Sound Export Options dialog

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);
}
```

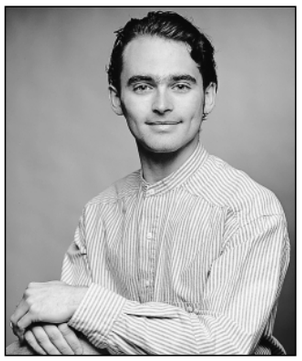
SOUNDING OFF

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.•



BALANCE OF POWER

Advanced Performance Profiling

DAVE EVANS

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.

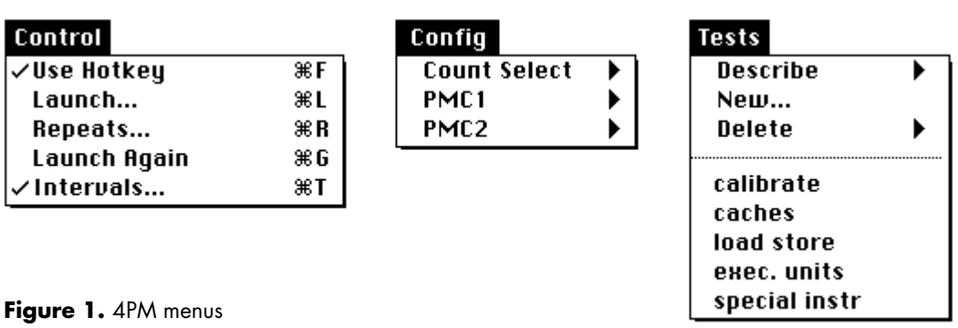


Figure 1. 4PM menus

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.

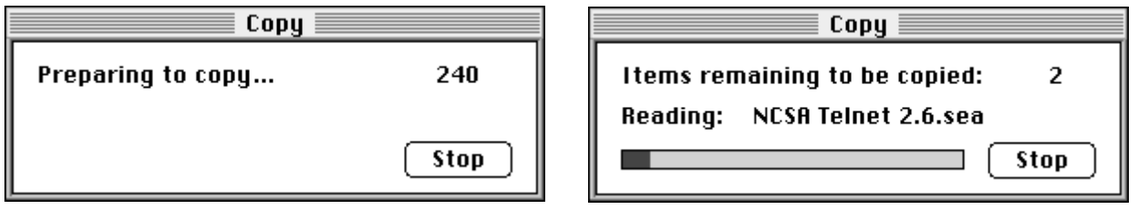
Similarly, if the chosen name is too long, it’s more helpful for the message to state the maximum number of characters a filename can have.

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).



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?

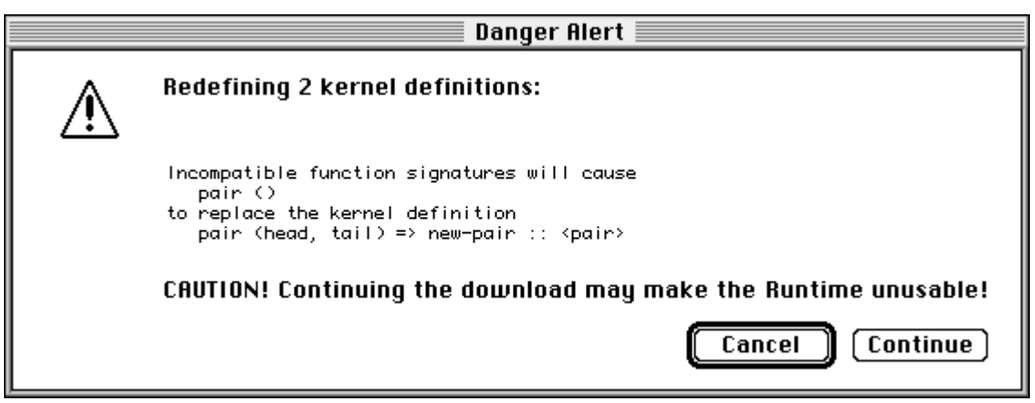


Figure 3. Poorly designed "danger alert"

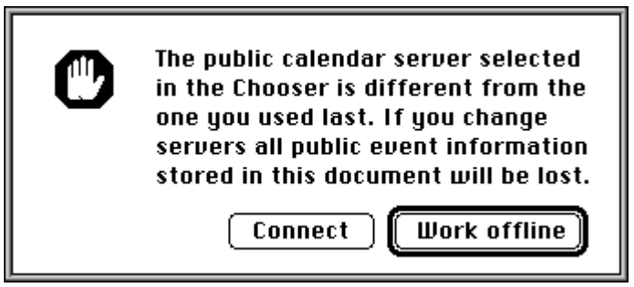


Figure 4. Poorly designed "server alert"

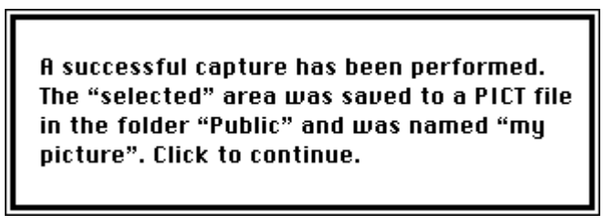


Figure 5. Poorly designed "finished alert"

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.

MPW TIPS AND TRICKS

ToolServer Caveats and Carping

TIM MARONEY

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 MARONEY 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.

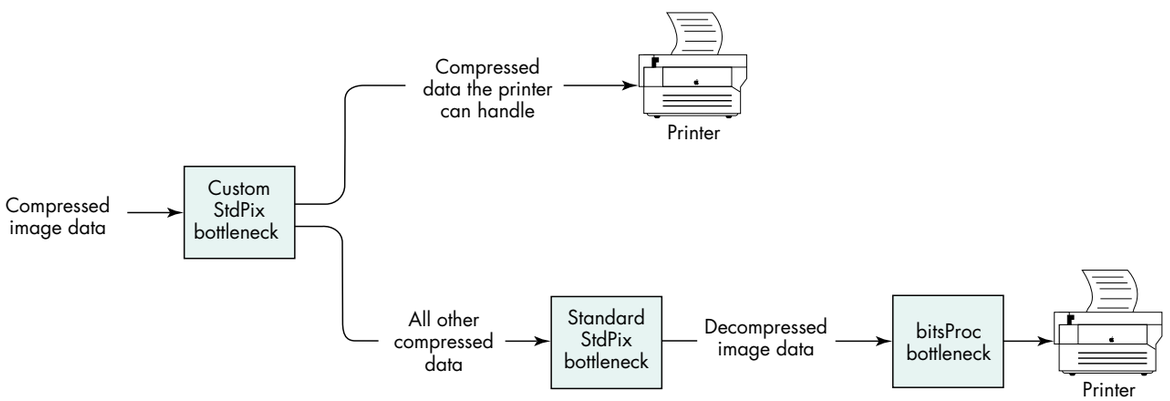


Figure 1. Special handling of compressed image data in the LaserWriter 8.3 driver

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	LocalTalk	288 seconds	129 seconds
	LaserWriter 16/600	EtherTalk	116 seconds	44 seconds
Portrait.pict	LaserWriter 320	LocalTalk	54 seconds	37 seconds
	LaserWriter 16/600	EtherTalk	22 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.

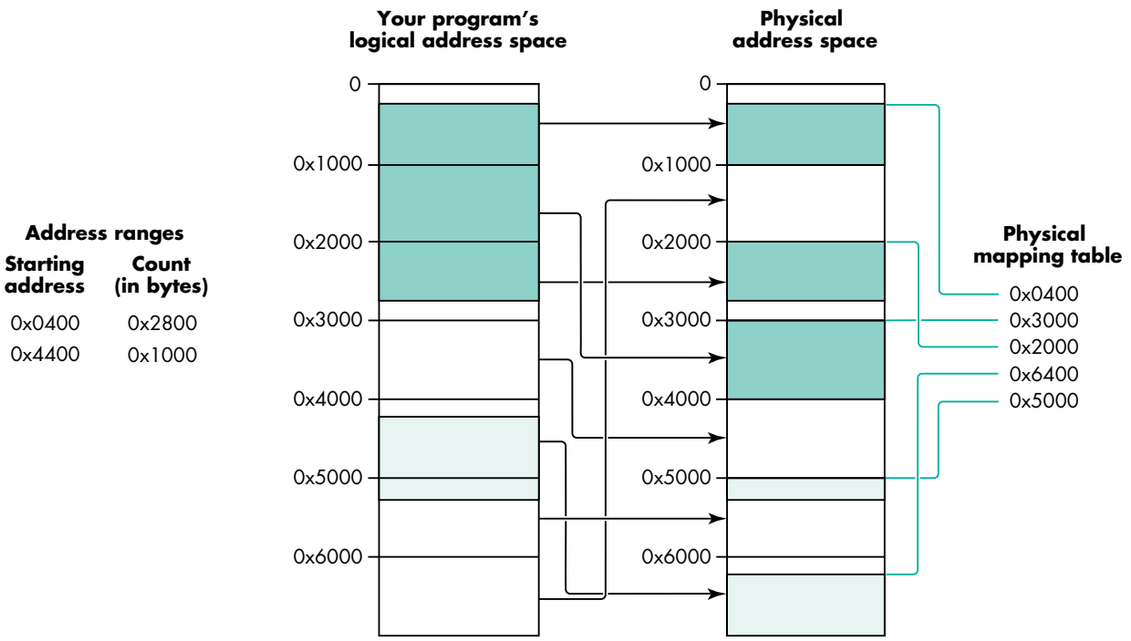


Figure 1. Mapping to multiple pages

- 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.

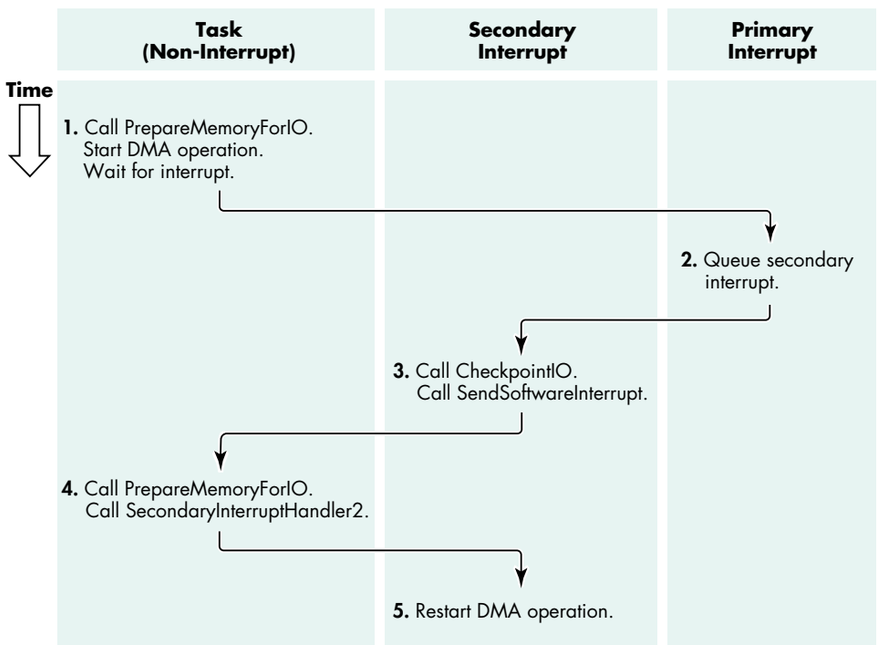


Figure 2. The progress of a data transfer with partial preparation

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.

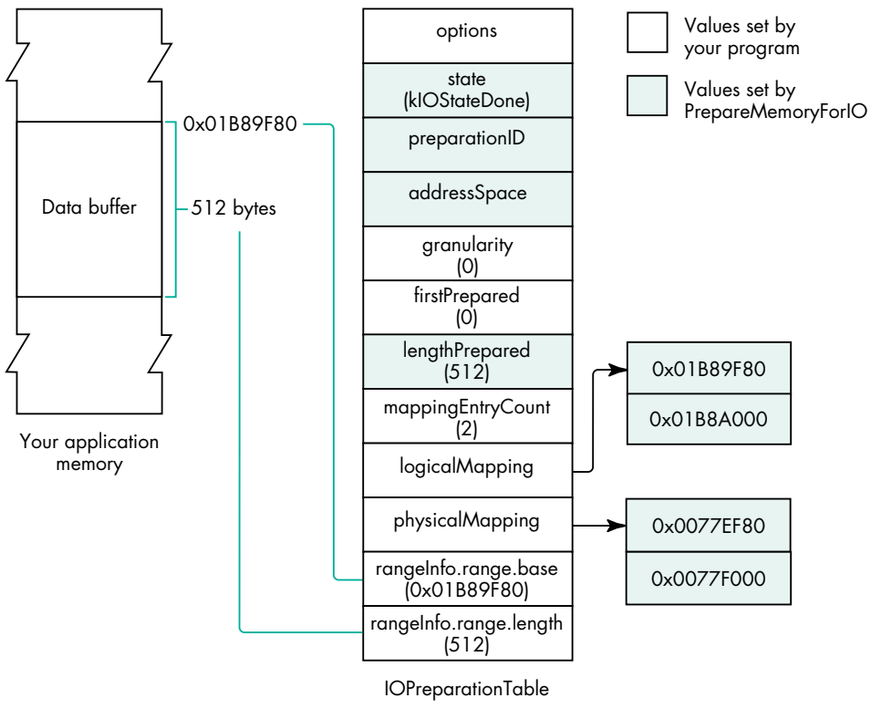


Figure 3. A simple IOPreparationTable

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.

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.

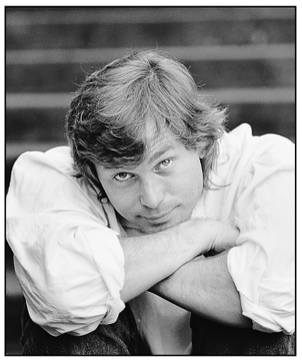
There's a kludgy workaround, but it will break under future systems. You could, however, use sequential calls to the Standard File Package instead of nesting them. This is a bit of a pain, but should accomplish what you want. Here's how: Put up the first dialog. In your filter routine, when the user 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.)•



THE VETERAN NEOPHYTE

The Right Tool for the Job

DAVE JOHNSON

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 ¡Xopher 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.•

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.

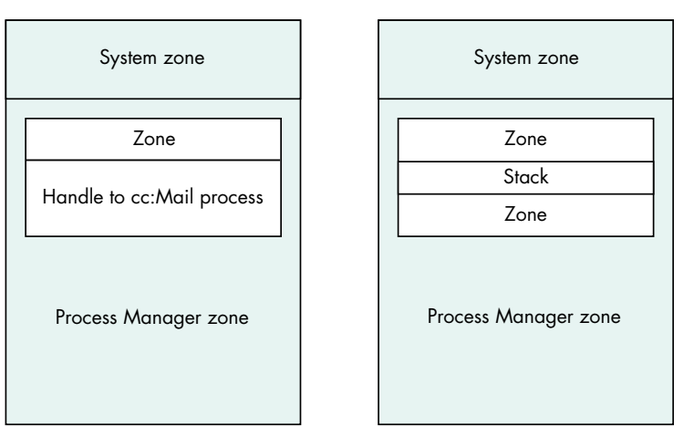
BAL I'll take the second question first. Zones are created by InitZone, and they're never explicitly destroyed. In the Modern Memory Manager, there's new logic in DisposeHandle that checks to see if the handle is a zone; if so, it assumes the zone is destroyed and updates the heap tree.

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.•

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W