develop
The Apple Technical Journal

Issue 22 June 1995

QuickDraw 3D: A New Dimension for Macintosh Graphics

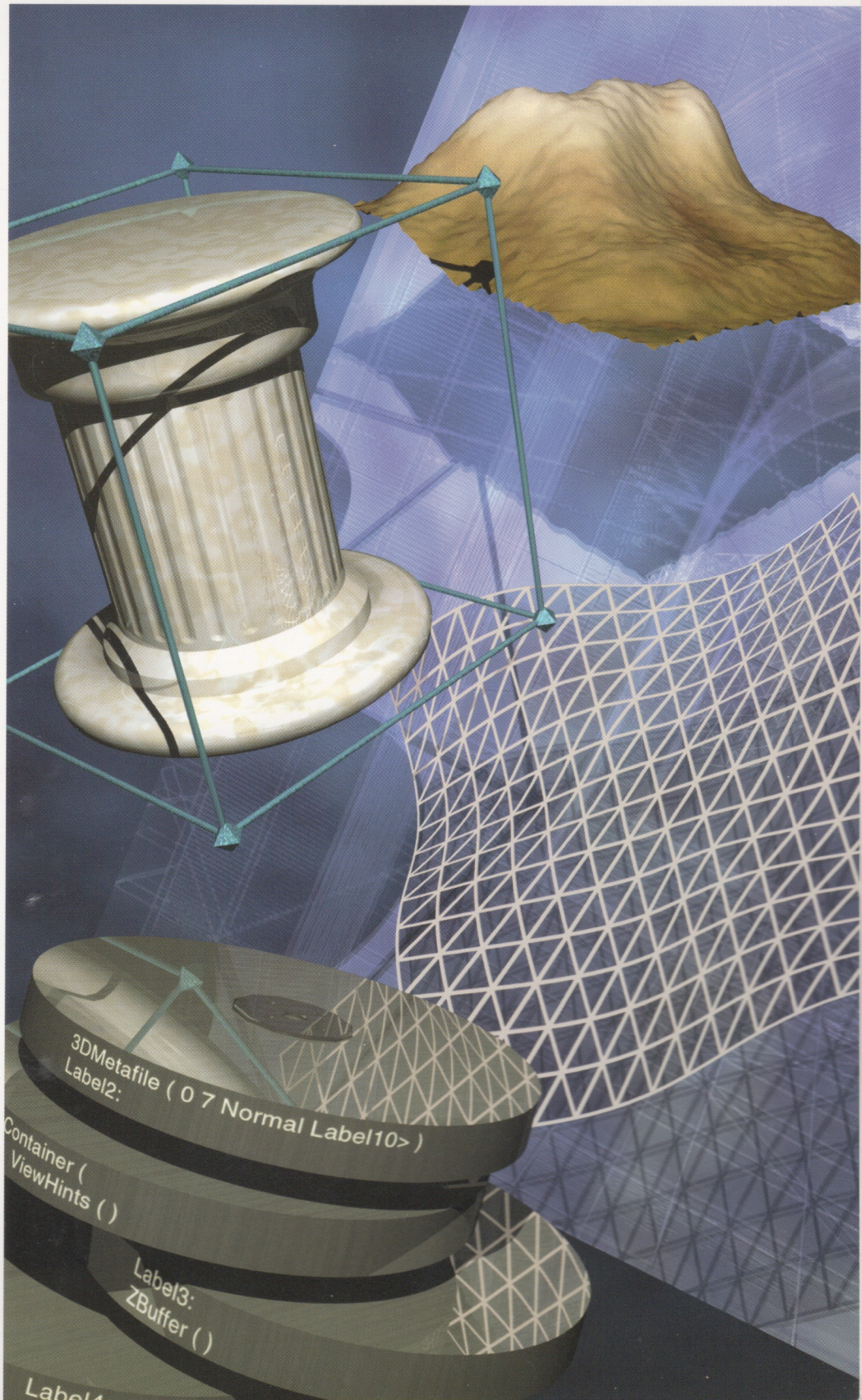
Copland: The Mac OS Moves Into the Future

Creating PCI Device Drivers

Custom Color Search Procedures

The OpenDoc User Experience

Futures: Don't Wait Forever



$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 *Greg Joswiak*

His Boss *Dennis Matthews*

Review Board *Pete "Luke" Alexander, Dave Radcliffe, Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*

Contributing Editors *Lorraine Anderson, Steve Chernicoff, Toni Haskell, Jody Larson, Cheryl Potter*

Indexer *Marc Savage*

ART & PRODUCTION

Production Manager *Diane Wilcox*

Technical Illustration *Deb Dennis, Shawn Morningstar, John Ryan*

Formatting *Forbes Mill Press*

Photography *Sharon Beals, Deb Dennis, Maggie Fishell*

Cover Illustration *Graham Metcalfe of Metcalfe/Shubert Design; modeled and rendered in Strata StudioPro*

Online Production *Cassi Carpenter*

ISSN #1047-0735.

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, also in the Developer Services area.)

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

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

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)974-6395. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)974-6395. Or write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014.

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

Subscriptions and back issues. You can subscribe to *develop* through APDA (see below) or use the subscription card in this issue. You can also order printed back issues. For subscription changes or queries or back issue orders, call 1-800-877-5548 in the U.S., (815)734-1116 outside the U.S. Or write AppleLink DEV.SUBS or Internet dev.subs@applelink.apple.com. *Be sure to include your name, address, and account number as it appears on your mailing label in all correspondence related to your subscription.* One-year U.S. subscription price is $30 for 4 issues of *develop* and the *develop Bookmark* CD; all other countries, $50 U.S. For Canadian orders, price includes GST (R100236199). Back issues are $13 each in the U.S., $20 all other countries.

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

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

This is a very forward-looking issue of *develop*. The cover article is on QuickDraw 3D, whose final release won't have shipped by the time you read this (though it should be soon). We've also got articles on Copland and OpenDoc, which aren't due for final release for a while yet. You'll learn how Copland will take the Mac OS into the future and how OpenDoc will affect the way users work with documents. There's an article on creating PCI device drivers that will be — as far as we can tell as of this writing — forward compatible with Copland. And we've got an article on the very subject of futures, a convenient way of implementing asynchronous interapplication communication, which will be especially valuable as more applications become scriptable and as component-oriented systems like OpenDoc become more prevalent. All in all, we're looking *ahead*.

Having articles on technology that hasn't shipped yet makes it tough for us to give you solid information and code that we know will withstand the test of time and not change in the future. QuickDraw 3D is shipping soon enough that we know *that* article and its code are reasonably solid — but the software is "beta" as of this writing, so changes can still happen. The Copland and OpenDoc articles provide only background information that should prove helpful as you consider how to use those technologies in your work; these articles provide no code, but only a context for the respective technologies. And although the PCI article does its best to tell you what may or may not work with Copland, there are limits to how far it can see into the future.

While it's certainly atypical to have *develop* articles that aren't based on good, solid code, we felt these articles would nonetheless be of interest to you. We'd really like to know whether you agree with our decision. Our Review Board meetings are driven by what we think you want; we periodically need a reality check from you.

There's yet another way that we're giving you a glimpse into the future: this issue's CD contains a Preliminary Drafts folder containing articles that we expect to publish in a future issue of *develop*. Again, we didn't want to keep you from getting information that you might find useful. This time we've got an article on implementing multipane dialogs and another on performing timing operations. Look in this folder from now on for "extra" articles.

So please, take a moment to give us your feedback on all of this (see the inside front cover to find out how to contact us). Why not stop by for a chat if you're at this year's Worldwide Developers Conference? Help us help you do a better job; that's why we're here.

Looking *forward* to hearing from you,

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) has been writing and editing for so long that she can do it in her sleep. In fact, she sometimes lies awake at night trying to solve writing problems — as she used to do for code bugs back when she was a programmer. To help get her mind off work, Caroline does Tai Chi and Ch'i Kung and curls up with her longtime feline companion, Cleo. But even then she can't get away from playing with words, as she continually adds to the long list of Cleo's nicknames; current favorites include Fuzz Bucket and The Purrmeister. Caroline agrees with Albert Schweitzer that there are two means of refuge from the miseries of life: music and cats.•

LETTERS

MORE ON HIDING DIALOG ITEMS

In *develop* Issue 20, I came across the Q&A (at the bottom of page 107) that recommends using AppendDITL and ShortenDITL to add or remove many dialog items at once rather than using ShowDItem and HideDItem on each item individually. I agree with what's said; however, there's an issue with using AppendDITL that I encountered recently and confirmed with Developer Support.

I've been involved in writing an application that uses 'ictb' resources to define the font for each dialog item. This is necessary for our application to allow globalization. When AppendDITL is called to append items to the dialog, the associated 'ictb' resource for the appended DITL resource isn't loaded. 'ictb' resources are loaded only when NewDialog is called. As a result, AppendDITL can't be used in this case; the show/hide items mechanism must be used instead.

I find *develop* informative; keep up the good work.

— Niall Quiggin

Ah, the inevitable exception to the rule. Thanks for pointing it out.

— Dave Johnson

PUZZLE PAGE ERROR: OPENRF

The solution for the Puzzle Page in *develop* Issue 20 is wrong. It says that the Finder should use OpenRF instead of OpenResFile. OpenRF allows the resource fork to be accessed only as a data stream, and so is useful only if the code wants to copy the entire resource fork without examining its contents. To look at bundles and icons and such, a routine such as HOpenResFile must be used — which is, in fact, what the Finder calls.

The cause of the bug isn't that the Finder uses fsRdWrPerm, but that it uses fsCurPerm. *Inside Macintosh: More Macintosh Toolbox* implies that fsCurPerm will work fine if the file is open for writing by someone else, and that read permission to the file will be granted in that case. But unfortunately, fsCurPerm will fail, just like fsRdWrPerm, if the file is already open for writing. To guarantee access to the resource fork of the file, fsRdPerm must be used instead of fsCurPerm. This change was made to the Finder in system software version 7.5.1.

Still, you can't blame Shelley and Byron for getting the wrong answer; they're just dogs, and most dogs don't have access to Finder sources.

— Greg Anderson, Apple Computer

I conferred with my dogs and they apologized profusely for assuming the inner workings of the Finder that they indeed did not understand. Thanks for the correction.

— Cary Clark

PUZZLE PAGE STINKS

Has it ever occurred to you how small must be the audience to which your regular contributors KON & BAL are playing? Their Puzzle Page is elitist and intellectually arrogant. Who do you imagine would be privy to the Apple-Eyes-Only knowledge necessary to solve some of these puzzles?

WHAT DO *YOU* THINK OF THE PUZZLE PAGE or the rest of *develop*, for that matter? We welcome timely letters to the editors, especially regarding articles published in *develop*. Letters should be addressed to Caroline Rose — or, if technical *develop*-related questions, to Dave Johnson — at AppleLink CROSE or JOHNSON.DK. Or you can write to Caroline or Dave at Apple Computer, Inc., 1 Infinite Loop, M/S 303-4DP, Cupertino, CA 95014. All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

As you progress further and further into their morass of micro-minutiae, they indicate that you're less and less clever due to your ever-reducing "score." The whole concept is punitive, pedantic, and boorish. And those invectives at the end of the article continue the process of belittling the reader with the suggestion that, due to your incredibly low score, "Maybe you'd better stick to AppleScript." Ouch! As it happens, AppleScript is an incredibly powerful technology that helps to differentiate the Mac OS from being just another pretty interface. Their attempt at being humorous isn't lost on me, but it failed nonetheless.

Those guys are certainly smart and Apple needs to have people like that on the payroll. But the average fellow in Kansas with a subscription to *develop* who has adopted Apple as his computing beacon is mocked by such articles and to no real good end. The Puzzle Page is wasted on all but the most inner circle of monks in the Apple sanctum sanctorum.

— Lance Drake

Your letter was surprising, since we get a lot of good feedback on the Puzzle Page. The puzzle format is just for fun (heh heh). The idea is that you learn something from the debugging techniques. Probably no one ever scores above 0, but that's not really the point. If you haven't already, you might want to take a look at the two letters in Issue 20 on the subject of the Puzzle Page.

Humor is a tricky thing: what some people find hilarious, others find repugnant. I'm sorry the Puzzle Page doesn't work for you. I certainly don't want any of our readers to feel mocked; maybe our publishing this letter will stimulate some dialog on this.

Regarding your specific comment about AppleScript: we couldn't agree more. We hope you'll be pleased with our new regular column, According to Script.

By the way, Apple does indeed need smart people like KON & BAL on the payroll, but they don't work for Apple anymore.

Thanks for writing.

— Caroline Rose

UNTIDY CODE (GIVE US A BREAK)

Greg Anderson's article in Issue 20 of *develop*, in the listing on page 67, gave me a probably unintentional insight into the deeper workings of Apple code. Apparently, constructions like this

```
while (true) {
   do something
   if (somethingelse) break;
}
```

are acceptable at Apple nowadays. Surely there must be a better, less sloppy and lazy way to do this. (Please don't ask me what's wrong with it; that would force me to go and buy Windoze machines next.)

— Joost Carpay

*You're right; the use of a **break** statement in conjunction with **while (true)** is generally considered poor style. Good style would be:*

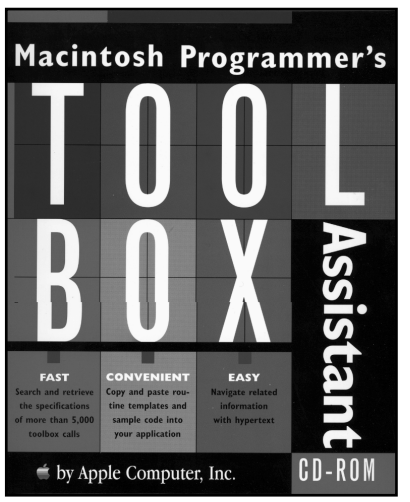
```
condition = true;
while (condition)
   condition = DoSomething();
```

While code that appears in develop *should of course use good style, the* develop *staff tells me that they are loath to enforce particular rules; they can, however, make suggestions, and will keep an eye out for this construct in the future. Apple's guidelines for software development recommend against using breaks inside loops and also against using **do/while** in place of a simple **while** loop.*

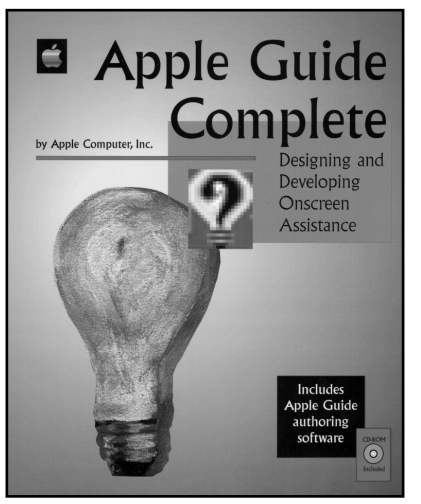
The ultimate metric used to judge code should be the clarity of the intent of the algorithm in question. Using good and consistent style certainly improves the readability of code, but I would hope that small infractions of style would be forgiven if the intent of the code remains clear. Code quality is important to Apple, and we're always working at improving the process used to produce system software.

— Greg Anderson

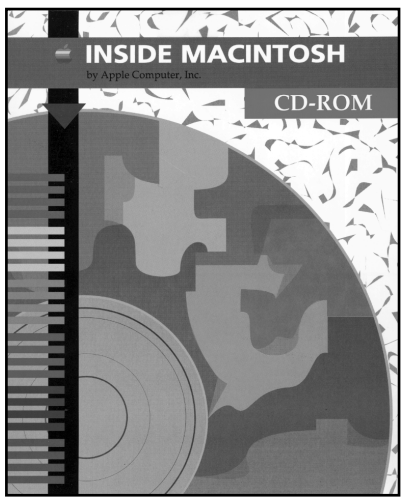
Indispensable Tools of the Trade



Macintosh Programmer's Toolbox Assistant



Apple Guide Complete



Inside Macintosh CD-ROM

JUST RELEASED!

Macintosh Programmer's Toolbox Assistant by Apple Computer, Inc., provides instant access to essential information for more than 5,000 toolbox calls that are at the heart of the Mac™ OS. Directly accessible from most of the popular development environments, this CD-ROM has been carefully designed to help you reduce the time it takes to develop your applications. With abundant hypertext links and the ability to copy and paste routine templates and sample code, *Macintosh Programmer's Toolbox Assistant* is **the** productivity tool for programmers.

ISBN 0-201-48342-4

JUST RELEASED!

Apple Guide Complete: Designing and Developing Onscreen Assistance by Apple Computer, Inc., is the official, complete kit for producing interactive tutorials with System 7.5. The book demonstrates how to develop a wide range of onscreen help systems that streamline everything from task-oriented procedures to quick tips and reference material. The accompanying CD contains the Apple Guide authoring software and will help instructional designers, scripters, and programmers really get the most out of this powerful help system.

ISBN 0-201-48334-3

Inside Macintosh CD-ROM has already become an invaluable reference for thousands of programmers since its publication just this past October. The CD-ROM contains more than 16,000 pages of the complete text from 26 volumes of *Inside Macintosh* library — the definitive reference for anyone writing software for Macintosh computers plus the text of *Macintosh Human Interface Guidelines*. No Macintosh programmer should be without this ultimate electronic resource.

ISBN 0-201-40674-8

Addison-Wesley Publishing Company

Available at fine technical bookstores in your area, or call 1-800-822-6339 for U.S. orders and 1-800-447-2226 for International orders.

QuickDraw 3D: A New Dimension for Macintosh Graphics

QuickDraw 3D is a new technology that helps developers bring 3D capabilities to their applications. It runs on all Power Macintosh computers and offers high-performance 3D rendering and other features that make working with 3D data easier. This article gives the basics you'll need to use QuickDraw 3D in your application, whether you're a consummate 3D developer, a classic 2D application developer, or a game developer.

PABLO FERNICOLA AND NICK THOMPSON

QuickDraw 3D is the newest enhancement to the Macintosh graphics architecture. Developers have been requesting a 3D library, supported at the system level, since the Macintosh was introduced. Although a number of Macintosh developers have produced some amazing 3D applications, 3D graphics capabilities were relegated to niche applications due to the lack of support at the core operating system level. QuickDraw 3D, which is expected to ship in mid-1995, brings the ability to deal with 3D graphics to all Power Macintosh applications: not only can traditional 3D applications take advantage of it, but it provides base functionality for general-purpose applications as well.

QuickDraw 3D is a Code Fragment Manager–based shared library, with a C-based API. Here we'll cover some concepts you need to know to get basic QuickDraw 3D support into your application. This issue's CD contains a prerelease version of the QuickDraw 3D shared library, the 3D Viewer shared library, programming interfaces, preliminary *Inside Macintosh: QuickDraw 3D* documentation, sample code, utility libraries, and other goodies. Two of the sample programs are discussed in this article.

The API described in the article is based on a beta version of QuickDraw 3D; although nearly final, the API may change before the final release of the software.•

In addition, we'll talk about reading and writing data in QuickDraw 3D metafile format, which is a way of representing 3D data in a consistent, transferable manner. But first we'll set the stage with some background information.

PABLO FERNICOLA (AppleLink PFF, eWorld EscherDude) After spending many years working in 3D graphics under operating systems named **IX, in a faraway land called Alabama, Pablo made the transition to real computers. After moving to Silicon Valley, he learned to beat the traffic jams by getting to work before 8 A.M. and going home after 10 P.M. Now he can be found staring out the window and wondering how he's going to get home on Interstate 280 after the next earthquake.•

NICK THOMPSON (AppleLink NICKT) is currently establishing himself as the Mountain Dew–guzzling fat fool of Developer Technical Support. Unable to work the winter blubber off due to killer waves that are preventing him from surfing on the California coast, Nick has been consoling himself with learning the wonder that is QuickDraw 3D. He was last seen wandering down one of the corridors at Apple mumbling to himself.•

QUICKDRAW 3D — SO, WHAT'S THE BIG DEAL?

As we'll explain further in this article, QuickDraw 3D provides developers with a number of benefits:

- a rich set of high-level geometries
- built-in renderers that cover the base functionality needed by developers
- immediate and retained graphics
- a common 3D file format
- human interface guidelines and widgets
- a 3D pointing-device manager that provides support for input devices with more than two degrees of freedom
- pointing and picking support that enables user selection of 3D data
- transparent access to graphics accelerators
- an extensible, plug-in shading and rendering architecture
- implementation advantages over other 3D libraries

We've made dealing with 3D data in applications easier with QuickDraw 3D. By creating a standard for data interchange, with a well-rounded metafile definition, we're enabling applications to read and write 3D data in a consistent format. The metafile specification addresses requests from both end users (who couldn't exchange data between applications in a common format) and developers (who had to write special-case code to deal with several different file formats).

QuickDraw 3D comes with a set of human interface guidelines to foster the adoption of a consistent look and feel between applications (see "The QuickDraw 3D Human Interface"). 3D applications today are geared toward the trained 3D expert; what you learn in one application is generally not transferable to another application. By following the QuickDraw 3D human interface guidelines, however, developers can help make 3D graphics an integral part of the user experience within their applications.

QuickDraw 3D technology has been made possible in part by the dramatic performance improvements in the Power Macintosh line of computers. The performance of QuickDraw 3D is scalable across the Power Macintosh line; we've put in a lot of effort to ensure that the performance on even entry-level computers is excellent. With hardware acceleration, these computers can easily compete (and win) against mid-range workstations costing a lot more money.

HOW QUICKDRAW 3D COMPARES WITH OTHER LIBRARIES

QuickDraw 3D offers many advantages over other 3D libraries. When using other graphics libraries, you're on your own if, for instance, you want to change the way a scene is rendered (say, by doing ray tracing or applying procedural shading): you have to reimplement all of the 3D architecture. With QuickDraw 3D, you only have to write code to deal with the specific area that you want to change. And, even better, the code you write can be used as a plug-in by other applications.

Unlike some libraries, QuickDraw 3D will be able to take advantage of a number of 3D hardware acceleration solutions, since acceleration was one of its design criteria. Another important criterion was cross-platform support. For example, a renderer could be written to take advantage of low-level 3D libraries, such as the Silicon Graphics OpenGL graphics library.

THE QUICKDRAW 3D HUMAN INTERFACE

BY DAN VENOLIA

QuickDraw 3D provides human interface guidelines (in version 1.0) and a toolkit for implementing the guidelines (to come in the second major release). A sample application on this issue's CD illustrates our current ideas for a 3D human interface. By getting a preview of our plans, you can start taking your applications along the common path.

Our main goal is to provide integration into the Macintosh experience. We feel that 3D graphics will be the next popular multimedia data type — in the way that 2D graphics, sound, and movies have been in the past — and users will want to incorporate 3D data into their documents in the same way that they can now incorporate other multimedia data types. To do this they'll need an interaction model built on the 2D principles that they're familiar with.

Our guidelines offer suggestions and examples of how things can be done. If your applications are targeted for a very specific audience, and you know that audience well, you may decide to communicate with them in a different way, and that's perfectly OK.

One of our guidelines, about direct manipulation through the use of a widget, is illustrated in Figure 1. Here we've appropriated the 2D grab handles that are popular in many "draw" programs and extended them to 3D. A widget is a set of handles for control of spatial parameters. Some widgets, such as the scale tool shown in Figure 1, indicate selection of a shape, while others make an invisible object, such as a light or a camera, visible.

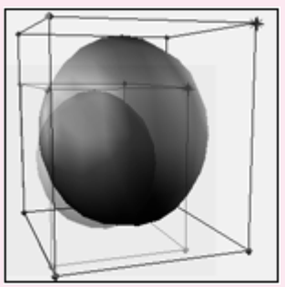


Figure 1. A scaling widget

Figure 2 shows what a full-featured 3D application might look like. The emphasis here is on what's the same as in 2D applications rather than on what's unique. The illustration shows a shape selected with a rotation widget, a material selection palette, a room metaphor, and a document containing multiple views of a scene.

WHAT YOU CAN DO WITH QUICKDRAW 3D

The 3D application development process can be broken down into four areas: creation of 3D data into a set of data structures, manipulation of that data in the human interface of the application, presentation of the data by displaying it, and transportation of the data (saving to and reading from files). QuickDraw 3D provides support in each of these areas. You can implement one or more of them in your application:

- QuickDraw 3D geometries — If you're planning to write an application to deal with the creation of models, QuickDraw 3D lets you define the representation of the objects to be modeled in 3D form.
- QuickDraw 3D human interface — Maybe you want to allow users to visualize 3D data and models in a standalone application or as part of an existing application. QuickDraw 3D's human interface guidelines and built-in widgets provide a consistent way of manipulating 3D objects.
- QuickDraw 3D rendering and shading — Rendering turns the 3D geometries into pixels; shading determines what color those pixels should be. Realism can be added by applying textures to objects: *texture mapping* takes a texture (usually from a picture source, such as a picture of a brick wall) and wraps it around an object. For

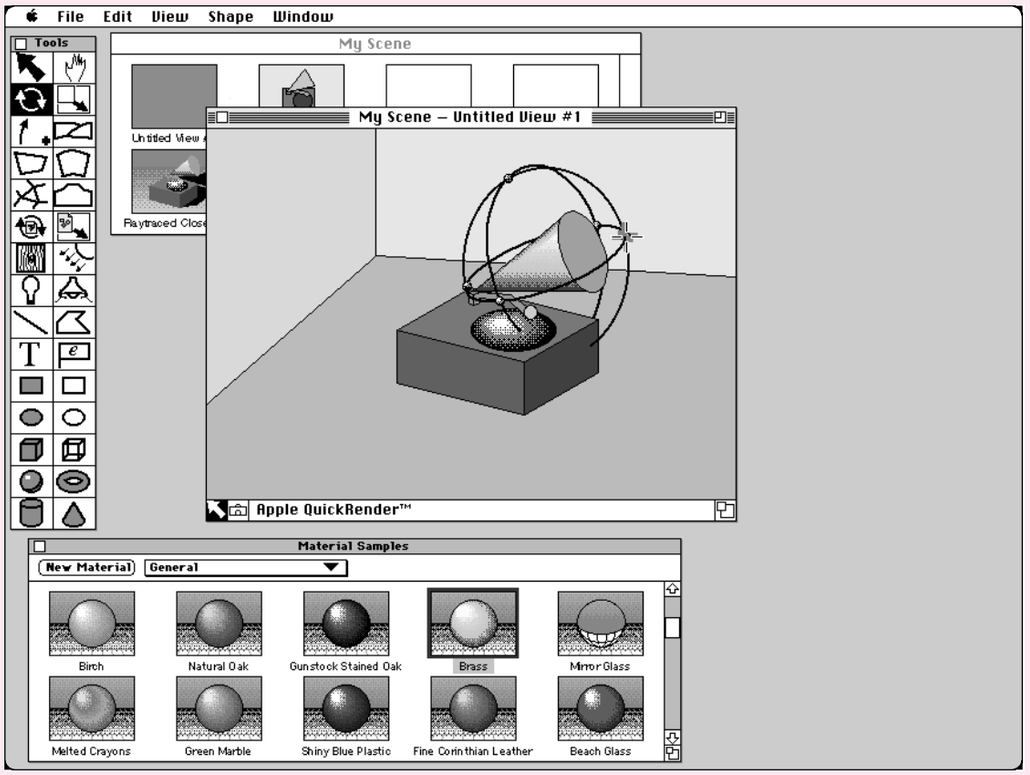


Figure 2. Conceptual sketch of a 3D application

example, Figure 3 shows a dinosaur mesh rendered with a skin texture picture as a texture map. In its second major release, QuickDraw 3D will enable you to write plug-in renderers and shaders and license them to other developers.

The dinosaur model was supplied in QuickDraw 3D metafile format courtesy of Viewpoint DataLabs Intl.•

- QuickDraw 3D metafile format — If you want to provide 3D clip art in the form of models, you'll really be pleased with QuickDraw 3D's metafile format. One of the common problems encountered by users when working with several 3D applications is that of data interchange, where one application's file is not readable by another due to the multitude of 3D data formats. QuickDraw 3D addresses this problem by providing a standard for the interchange of 3D data. This device- and platform-independent representation of 3D data is extensible, so your custom data gets preserved. And all of the elements for a scene can be stored in the metafile, including lighting, camera objects, texture maps, and shaders.

ROAD MAP FOR ADOPTION

Based on our experience working with developers, we've created a road map for adoption of QuickDraw 3D. Here we'll look at how different application developers might begin to adopt QuickDraw 3D, in order from the least to the greatest amount

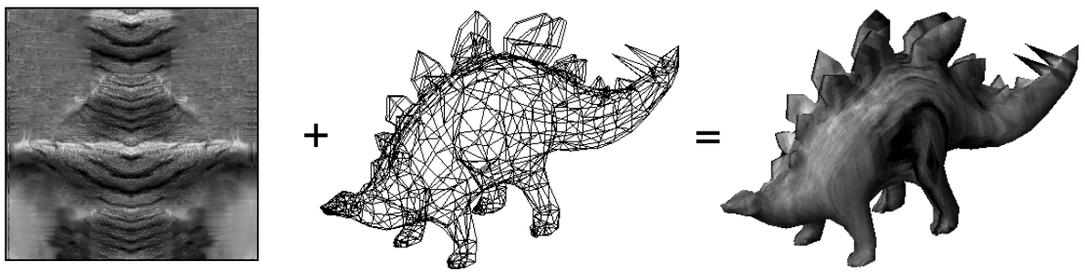


Figure 3. Dinosaur mesh mapped with a skin-like texture

of support. These categories provide you with a general strategy for bringing QuickDraw 3D into your applications.

- Developers of general-purpose 2D applications should add support for the metafile format, enabling users to read and save 3D data within an application. This can be achieved by using the 3D Viewer, which allows 3D objects derived from metafile data to be viewed and manipulated by the user.
- Developers who use other 3D libraries and may not be ready to move to QuickDraw 3D just yet should at least add support for the metafile format and additionally consider adopting the QuickDraw 3D human interface guidelines. Obviously, support for the metafile format requires writing a parser to convert metafile data to another internal representation (Apple will be supplying parser code). Implementing the human interface guidelines will make the application be compatible with and look consistent with other 3D applications available on the Macintosh. Note that an application that uses a 3D library other than QuickDraw 3D will have a harder time using the 3D Viewer.
- Developers of existing 3D applications who want to take the first step toward creating a QuickDraw 3D–savvy application should take advantage of QuickDraw 3D's rendering capabilities through the use of immediate-mode rendering (more on this later). This method provides not only fast rendering in software but also transparent access to hardware, while allowing the application to preserve its own data structures. In addition, these developers should plan to add support for the metafile format and the human interface guidelines.
- Developers who want to leave the low-level work to QuickDraw 3D, and concentrate on creating differentiating features within their applications, should make their applications as QuickDraw 3D–savvy as possible. This means taking advantage of the full API, including QuickDraw 3D's data structures and geometries (which provide metafile support virtually for free), rendering (both immediate and retained modes), and the human interface guidelines.

QUICKDRAW 3D ARCHITECTURE

The QuickDraw 3D architecture isolates in a layer within the system software those things that all developers have to do, leaving them to concentrate on the code that will allow their application to stand out. This architecture can be thought of as a sandwich filling that sits between your application and the hardware it's running on,

isolating you from having to deal with operating system and hardware issues directly. Like any good sandwich filling, if you examine it closely, you'll see that it's divided into a number of appetizing chunks. Figure 4 shows some of the functional blocks that make up QuickDraw 3D, with an emphasis on those areas that can be customized by developers.

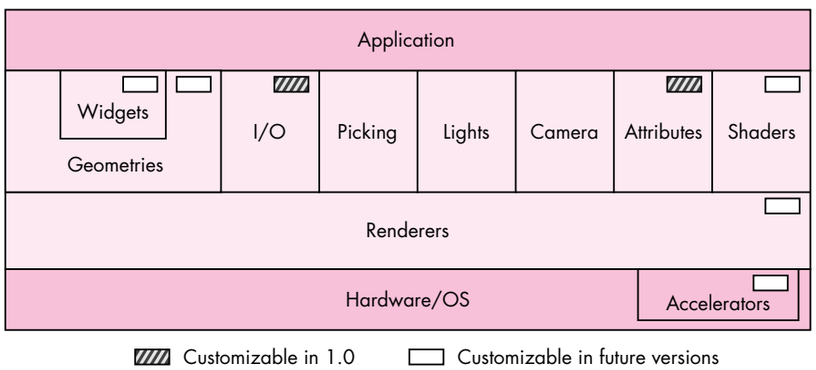


Figure 4. QuickDraw 3D architecture

Let's take a quick look at each of these functional areas, which we'll expand on later. Here we'll use the word *scene* to describe not only the objects being modeled, but also the lighting, camera settings, shaders, and other entities that affect the final appearance on output devices.

Widgets are used to enhance the user experience for 3D applications. For example, to allow the user to interact with an object, the application can draw grab handles, in the form of a translation widget, to allow the object to be manipulated.

Geometries are the encapsulation of data used to describe an object. Some geometries are provided as part of QuickDraw 3D, resulting in a very concise representation; for more information, see "QuickDraw 3D Geometries." (QuickDraw 3D uses geometries to draw widgets.)

The *I/O layer* provides support for metafiles. There are routines for reading and writing 3D data to Storage objects, which may be disk or memory based and are useful for providing Clipboard or drag and drop support in your application.

QUICKDRAW 3D GEOMETRIES

The QuickDraw 3D geometries that are currently available are as follows: line, polyline, triangle, point, simple polygon, general polygon, trigrid, mesh, box, marker, NURB curve, and patch.

In addition, the following geometries are planned for the second major release of QuickDraw 3D: torus, ellipse, ellipsoid, disk, cylinder, cone, and triangle strip. (In version 1.0, you can create any of these geometries by representing them as meshes.)

Where applicable, the geometries are parameterized so that they're ready for texture mapping or other shading effects.

Picking is used to determine which object a user chose. QuickDraw 3D's picking facilities are more extensive than in other 3D libraries, not only providing several different types but also returning quite a bit of information to the application beyond whether a hit took place.

Light objects supply the lighting for a scene. QuickDraw 3D provides four types of light sources: ambient, directional, point, and spot. Based on the light sources for a given scene and the illumination shader, the renderer makes intensity calculations for each object's surface and vertex contained in the scene.

Camera objects define a point of view into a particular scene. QuickDraw 3D provides three different camera types: view angle, orthographic, and view plane.

Attributes are used to specify different characteristics for each object (or parts of an object, such as its vertices or faces), and also to attach custom data to an object.

Shaders are used to modify or add data, on either a per vertex or a per pixel basis, as geometries are being processed by the renderer — for example, illumination and texturing shaders.

Renderers are the business end of QuickDraw 3D. A renderer is a set of routines used to create a shaded synthetic model of the scene, based on the information stored in the geometry and taking into account the lighting, surface attributes, and camera location. QuickDraw 3D provides two basic renderers: a wireframe and an interactive renderer. You can extend QuickDraw 3D by writing a plug-in renderer, developing an accelerator card, or implementing a combination of both — a renderer tied to a particular hardware setup.

IMPLEMENTING SUPPORT FOR THE 3D VIEWER

Now, on to the coding details. We realized that some application developers wouldn't want to get involved with the low-level details of a new API. We looked at the QuickTime model and saw that a lot of developers implemented support for viewing movie data by using movie controllers in their existing nonmultimedia applications. We likewise wanted to allow applications to support the viewing of QuickDraw 3D metafiles with minimal effort, so we've provided an additional shared library that implements a 3D Viewer. The Viewer allows users to view and have a basic level of interaction with 3D data without your having to make any QuickDraw 3D calls. Figure 5 shows a Viewer implementation in a modified version of the Scrapbook. (We used a preliminary version, so the Viewer interface may change.)

The car model was supplied in QuickDraw 3D metafile format courtesy of Viewpoint DataLabs Intl.•

Adding Viewer support is simple — it requires only about five function calls. Your application can check to see if the Viewer is available by calling Gestalt with the constant gestaltQuickDraw3DViewer.

We'll now look at how your application can create and use a QuickDraw 3D Viewer object. In the application named Simple 3D Viewer on this issue's CD, we create a window in which the only object is a Viewer.

As you read through the code samples, you'll notice that function names have a "Q3" prefix, data types have a "TQ3" prefix, and constants have a "kQ3" prefix. The part of a function name before the underscore indicates the object being operated on (the class), while the part after the underscore indicates the operation (the method). For example, to set the origin of a Box object, you'd call the function Q3Box_SetOrigin.•

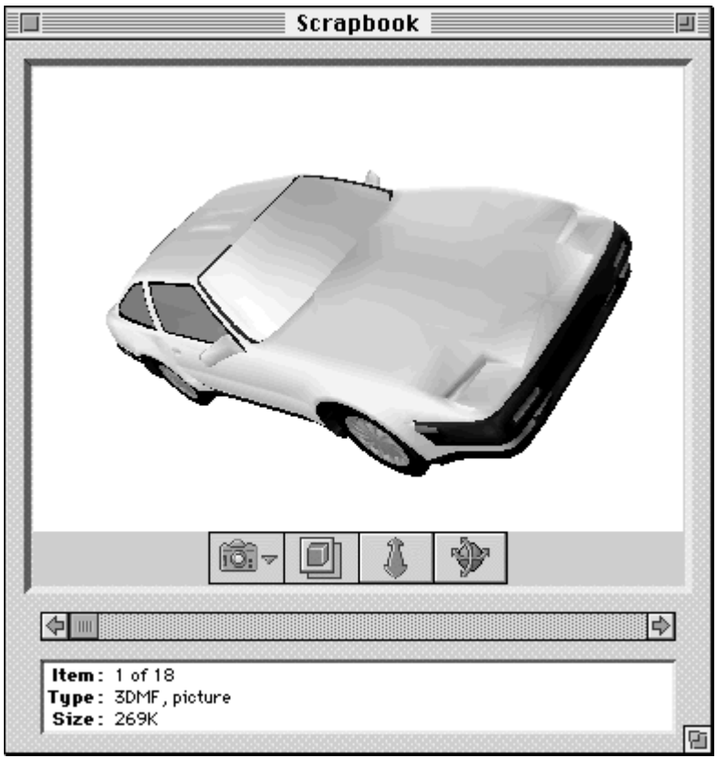


Figure 5. Viewer implementation in the Scrapbook

CREATING AND DISPOSING OF A VIEWER OBJECT

Creating and disposing of a Viewer object is very easy to do. You attach a Viewer to a window with the Q3ViewerNew function:

```
viewerObj = Q3ViewerNew((CGrafPtr)theWindow, &theRect, 0L);
```

This function takes a WindowPtr, a pointer to a Rect that describes the window area where you want the 3D scene to appear, and a long word containing flags for modifying the behavior of the Viewer. When you're finished with the Viewer, you need to dispose of it with the Q3ViewerDispose function:

```
Q3ViewerDispose(viewerObj);
```

ATTACHING DATA TO THE VIEWER

To display the contents of a metafile in your Viewer, you can use the Q3ViewerUseFile function:

```
Q3ViewerUseFile(viewerObj, fileRefNum);
```

Q3ViewerUseFile takes a reference to the Viewer object and a file reference to a previously opened QuickDraw 3D metafile. You can also display data from the Clipboard or data you created yourself, with the Q3ViewerUseData function:

```
Q3ViewerUseData(viewerObj, myDataPtr, myDataSize);
```

This function takes a reference to a Viewer object, a pointer to the data, and the size of the data in bytes. The data must be in metafile format.

HANDLING EVENTS

You need to modify your event loop slightly to give the Viewer the opportunity to handle events, as follows:

```
wasViewerEvent = Q3ViewerEvent(viewerObj, theEvent);
```

Q3ViewerEvent takes a reference to a Viewer object and a pointer to an event record (usually obtained from WaitNextEvent). This function allows the Viewer to respond to events, such as a mouse-down event in one of its controls. It returns a value of type Boolean that indicates whether the event was handled.

If the area occupied by the Viewer needs to be updated, you need to redraw the data in your update event handler by calling Q3ViewerDraw:

```
theErr = Q3ViewerDraw(viewerObj);
```

OTHER VIEWER FUNCTIONALITY

The Viewer allows access to the View object for the scene, which enables you to customize the Viewer's behavior by changing the renderer or lighting for the scene (more on Views later). Also, the Viewer provides support for cut, copy, and paste; see the Simple 3D Viewer sample on the CD for an example.

PROGRAMMING WITH THE QUICKDRAW 3D API: ERROR CHECKING AND INITIALIZATION

Now let's look at programming with the QuickDraw 3D API, starting with error checking and initialization. First, the QuickDraw 3D shared library needs to be installed in the Extensions folder or in the same folder as your project. During your development cycle you should use the debugging version of the library for extensive error checking.

Error checking may seem like a weird place to start, but checking and responding to what QuickDraw 3D is trying to tell you will save a great deal of trouble and strife during development. The QuickDraw 3D error manager provides several levels of error checking along with functions for checking the last error that occurred. The error checking, which is similar to that in QuickDraw GX, has three levels: errors, warnings, and notices.

- *Errors* are the most severe indication of a problem and can be divided into two kinds, fatal and nonfatal. You can determine whether an error is fatal with the call

  ```
  TQ3Boolean Q3Error_IsFatalError(TQ3Error theError);
  ```

  For a complete list of errors provided by QuickDraw 3D, look in the QuickDraw 3D header files.

- *Warnings* are less severe than errors, but you should be prepared to handle them. If the system generates a warning based on a recoverable situation that you ignore, often an unrecoverable error may occur later.

- *Notices* indicate problems that may exist with the way you're using the QuickDraw 3D library. Although they're less severe than warnings, you should take note of what notices are telling you, to prevent problems from occurring later in your application's execution. Notices are generated only in the debugging version.

You can install your own error, warning, and notice handlers, which can write the error information to a window or file or present a dialog or alert. Presenting too many alerts can be annoying to the user, so you should probably log errors, warnings, and notices to a file or a status window, and present a dialog or an alert only for fatal errors from which no recovery is possible.

DEFINING AND INSTALLING AN ERROR HANDLER

Handlers for errors, warnings, and notices are all similar — they're functions that take an error code of type TQ3Error and have no return value. Listing 1 shows a definition of an error handler.

Listing 1. Error handler

```
static void MyErrorHandler(TQ3Error firstError, TQ3Error lastError,
                           long refCon)
{
   char  buf[512];

   sprintf(buf, "ERROR %d - %s\n", lastError,
         getErrorString(lastError));     // Get the error as a C string.
   if (gErrorFile == NULL)
      gErrorFile = fopen("error.output", "w+");
   if (gErrorFile != NULL)
      fputs(buf, gErrorFile);
}
```

Once handlers have been defined, it's a snap to install them. For example, you would install the error handler defined in Listing 1 as follows:

```
Q3Error_Register(MyErrorHandler, 0L);
```

INITIALIZING QUICKDRAW 3D

Before you can use QuickDraw 3D, you need to call Gestalt to see if the library is installed, using the constant gestaltQuickDraw3D. You then need to initialize the library as shown in Listing 2. You call the Q3Initialize function to ensure that the QuickDraw 3D library gets a chance to allocate its internal data structures and to initialize any subcomponents (such as plug-in shaders) that it needs to call. You then do other initialization as needed, such as installing an error handler. The return value indicates whether the call was successful.

When your application is about to quit, you should shut down your connection to the QuickDraw 3D library by calling Q3Exit, also shown in Listing 2. (Obviously a real application would have more sophisticated error handling here.)

CREATING AND DRAWING A SIMPLE 3D OBJECT: THE BOX APPLICATION

The Box application on this issue's CD is a simple QuickDraw 3D program that opens a window, displays 3D boxes in the window, and rotates the boxes (see Figure 6). While this isn't a useful application as such, it covers all the basics needed to create and display objects using QuickDraw 3D. It also illustrates double buffering support, which helps an application provide flicker-free drawing when animating

Listing 2. Initializing and closing the connection to the library

```
void Initialize3DStuff(void)
{
   if (Q3Initialize() == kQ3Failure) {
      // Handle the error.
      StopAlert(kQD3DInitFailed);
      ExitToShell();
   }
   MyErrorInit();
}

void Exit3DStuff(void)
{
   if (Q3Exit() == kQ3Failure) {
      // Handle the error.
      StopAlert(kQD3DExitFailed);
      ExitToShell();
   }
}
```

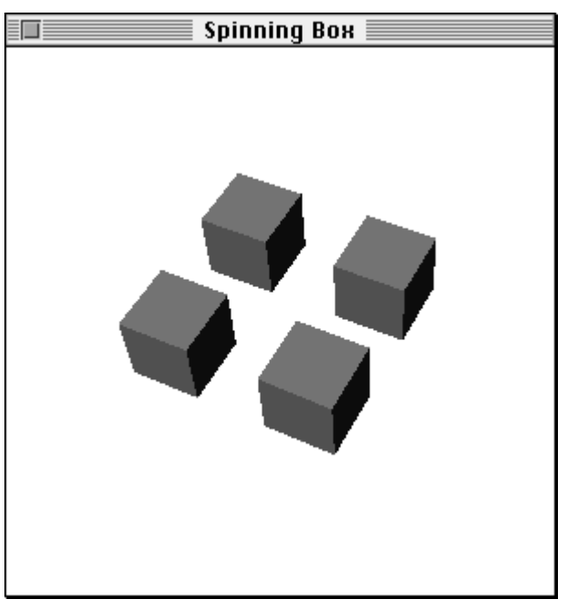


Figure 6. A window from the Box sample program

geometries; QuickDraw 3D's double buffering takes advantage of hardware double buffering when available.

For a more complex example, see the Modeller program on the CD, which shows most of the things a QuickDraw 3D application needs to do, such as reading and writing metafiles, texture mapping, and using interpolation styles.•

We define the following data structure to store the information that QuickDraw 3D needs to model and render our scene:

```
struct _documentRecord {
   TQ3ViewObject     fView;           // The view for the scene
   TQ3GroupObject    fModel;          // Object in scene being modeled
   TQ3StyleObject    fInterpolation;  // Style used when rendering
   TQ3StyleObject    fBackFacing;     // Whether to draw shapes that face
                                      // away from the camera
   TQ3StyleObject    fFillStyle;      // Drawn as solid filled objects or
                                      // decomposed to components
   TQ3Matrix4x4      fRotation;       // The transform for the model
};
typedef struct _documentRecord DocumentRec, *DocumentPtr, **DocumentHdl;
```

We can create a new instance of this type, initialize it with the required values, and store a reference to it in each window's refCon field.

OBJECT CREATION

Creating a simple object — like a box — is straightforward. We'll make four copies of the box, each with its own transform. The code to create these boxes is shown in Listing 3. We can store the boxes in our document simply by storing the value returned by this function in our document's fModel field.

Notice that we dispose of the boxes after adding them to the document group. QuickDraw 3D will create references to the boxes in the document group, so we can safely dispose of them. To be good QuickDraw 3D citizens and to make more effective use of memory, we need to dispose of each QuickDraw 3D object as soon as we're done with it. QuickDraw 3D keeps track of the reference count of each object to help detect memory leaks. If you're using the debugging version of QuickDraw 3D, it will tell you when you call Q3Exit if there are any objects remaining that need to be disposed of.

RETAINED AND IMMEDIATE MODE RENDERING

We talked earlier about retained and immediate modes. Which mode to use is the subject of big philosophical arguments in the world of 3D graphics. Some developers prefer one over the other as a matter of principle; other developers make a choice based on the type of application being developed. QuickDraw 3D offers the best of both worlds: not only does it support both ways of rendering geometric data, it also allows you to mix these types in the same rendering loop.

> **Retained and immediate modes are** simply methods of rendering, without the usual connotation of the term "mode" (a state that you must exclusively remain in once you get into it). Although this terminology has become common in the field of 3D graphics, retained rendering and immediate rendering calls can in fact be freely mixed.•

In *retained mode*, the definition and storage of the geometries are kept internal to QuickDraw 3D (as an abstract object). This mode provides convenient features for caching, rejection of entire objects based on clipping and culling, preservation of tessellated surfaces, multiple instantiation of objects (drawing multiple versions of an object but storing its definition only once), and conversion between geometry types. Retained mode is useful when the geometry has to be passed around to different modules within the application or to plug-in renderers. Extensive geometry editing functions are provided as part of the QuickDraw 3D API, which makes it easy to alter the data associated with each geometric object.

In *immediate mode*, the application keeps the only copy of the geometry. This is particularly useful when your application needs to reference data that's in a format

Listing 3. Creating four boxes

```
TQ3GroupObject MyNewModel()
{
   TQ3GroupObject       myGroup;
   TQ3GeometryObject    myBox;
   TQ3BoxData           myBoxData;
   TQ3GroupPosition     myGroupPosition;
   TQ3ShaderObject      myIlluminationShader;
   TQ3Vector3D          translation;
   TQ3SetObject         faces[6];
   short                face;

   // Create a group for the complete model.
   if ((myGroup = Q3DisplayGroup_New()) != NULL) {
      // Define a shading type for the group and add the shader to
      // the group.
      myIlluminationShader = Q3PhongIllumination_New();
      Q3Group_AddObject(myGroup, myIlluminationShader);

      // Set up the colored faces for the box data.
      myBoxData.faceAttributeSet = faces;
      myBoxData.boxAttributeSet = nil;
      MyColorBoxFaces(&myBoxData);

      // Create the box itself.
      Q3Point3D_Set(&myBoxData.origin, 0, 0, 0)
      Q3Vector3D_Set(&myBoxData.orientation, 0, 1, 0);
      Q3Vector3D_Set(&myBoxData.majorAxis, 0, 0, 1);
      Q3Vector3D_Set(&myBoxData.minorAxis, 1, 0, 0);
      myBox = Q3Box_New(&myBoxData);

      // Put four references to the box into the group, each one with
      // its own translation.
      translation.x = 0; translation.y = 0; translation.z = 0;
      MyAddTransformedObjectToGroup(myGroup, myBox, &translation);
      translation.x = 2; translation.y = 0; translation.z = 0;
      MyAddTransformedObjectToGroup(myGroup, myBox, &translation);
      translation.x = 0; translation.y = 0; translation.z = -2;
      MyAddTransformedObjectToGroup(myGroup, myBox, &translation);
      translation.x = -2; translation.y = 0; translation.z = 0;
      MyAddTransformedObjectToGroup(myGroup, myBox, &translation);
   }

   // Dispose of the objects we created here.
   if (myIlluminationShader != NULL)
      Q3Object_Dispose(myIlluminationShader);
   for (face = 0; face < 6; face++) {
      if (myBoxData.faceAttributeSet[face] != NULL)
         Q3Object_Dispose(myBoxData.faceAttributeSet[face]);
   }
   if (myBox != NULL)
      Q3Object_Dispose(myBox);
   return myGroup;
}
```

different from the one used by QuickDraw 3D or when a large number of vertices that make up the geometry are being edited continuously — for example, in the animation of a stress analysis for mechanical design.

The code in Listing 3 creates the boxes in retained mode, by creating objects that encapsulate the box data; QuickDraw 3D then manages the box data for us. If you want to add QuickDraw 3D rendering and drawing to an existing application with its own 3D data structures, you can draw in immediate mode instead. To draw a box in immediate mode, you simply initialize the values in the TQ3BoxData structure to the appropriate values and then draw the data directly in a rendering loop (described later) by calling the following function:

```
myStatus = Q3Box_Submit(&myBoxData);
```

Because you never create a QuickDraw 3D object, there's no need to call Q3Object_Dispose.

Notice that in Listing 3 we initialize an object using a data structure of type TQ3BoxData. This structure contains all of the information required to draw a Box geometry, but is not an object in itself. Because of this we don't call Q3Object_Dispose on the box data structure, but we do call it on the Box object.•

THE DRAW CONTEXT

All window system dependencies are isolated to a layer we call the *draw context*. This makes porting your application easier (and it also makes it easier for us to port QuickDraw 3D to other platforms). Although QuickDraw 3D is platform independent, of course at some point you'll need to deal with the realities of a particular platform's window system, in this case the Mac OS.

This is where the concept of a draw context comes in. It's a means for QuickDraw 3D to interface with the host environment. There's a special draw context for the Mac OS, called a *Macintosh draw context*; information describing this context is stored in a TQ3MacDrawContext object, which contains the information necessary for QuickDraw 3D to image the data on a computer running the Mac OS.

Listing 4 is a routine from the Box application that creates a Macintosh draw context the size of a window that we pass in. We're telling QuickDraw 3D to create a buffer in which to image the data; this is referred to as the *back buffer*. If we're using double buffering (that is, we set the doubleBufferState field of the Macintosh draw context to true), the *front buffer* will be the window associated with the draw context. The data is copied from the back buffer to the front buffer when Q3View_EndRendering is called. This helps provide flicker-free animation if you're animating the object being viewed.

Sometimes you might want to be able to get at the back buffer yourself; for example, you might want to create a picture preview of some metafile data to place on the Clipboard along with the metafile data, so that applications that don't support metafiles can display the picture. QuickDraw 3D makes this possible by providing a different type of draw context, called a *pixmap draw context*, which can be based on a GWorld. First you need to create a GWorld the size of the window area; then you can create a pixmap draw context as shown in Listing 5.

When using a pixmap draw context, you must keep the GWorld's PixMap locked all the time (which implies that you need to call LockPixels on it, to help avoid heap fragmentation). Also, the PixMap must be 32 bits deep — other depths are not supported.

Listing 4. Creating a Macintosh draw context

```
TQ3DrawContextObject MyNewDrawContext(WindowPtr theWindow)
{
   TQ3DrawContextData      myDrawContextData;
   TQ3MacDrawContextData   myMacDrawContextData;
   TQ3DrawContextObject    myDrawContext;
   TQ3ColorRGB             clearColor;

   Q3ColorRGB_Set(&clearColor, 1, 1, 1);
   myDrawContextData.clearImageState = kQ3True;
   myDrawContextData.clearImageMethod = kQ3ClearMethodWithColor;
   myDrawContextData.clearImageColor = clearColor;
   myDrawContextData.paneState = kQ3False;
   myDrawContextData.maskState = kQ3False;
   myDrawContextData.doubleBufferState = kQ3True;
   myMacDrawContextData.drawContextData = myDrawContextData;
   myMacDrawContextData.window = (CGrafPtr) theWindow;  // The window
                                        // associated with the view
   myMacDrawContextData.library = kQ3Mac2DLibraryNone;
   myMacDrawContextData.viewPort = nil;
   myMacDrawContextData.grafPort = nil;

   // Create draw context and return it; if nil, caller must handle it.
   myDrawContext = Q3MacDrawContext_New(&myMacDrawContextData);
   return myDrawContext;
}
```

THE CAMERA

A camera is a QuickDraw 3D object used to project a 3D scene onto a 2D plane. It defines a point of view on the scene and a method of projection onto the viewing plane. QuickDraw 3D provides three types of camera:

- *View angle* or *perspective* — This type of camera is defined in terms of a viewing angle and horizontal-to-vertical aspect ratio. It's the most common camera type because it provides a natural-looking perspective.
- *Orthographic* — This is a parallel projection, where the direction of projection is perpendicular to the projection plane. Orthographic projections are generally less realistic than perspective projections; however, they're popular for engineering drawings because parallel lines remain parallel in the projection, rather than converging to a single point on the horizon.
- *View plane* — This is a perspective projection defined in terms of an arbitrary viewing plane. This type of camera is useful for providing an off-axis view, which is used for scrolling.

We use a view angle camera for the Box application, creating the camera with the routine in Listing 6.

LIGHTING

QuickDraw 3D includes a number of different light objects that can be used to provide illumination to the surfaces in a scene. Lighting is *additive*, meaning that the

Listing 5. Creating a pixmap draw context

```
TQ3DrawContextObject MyNewPixmapDrawContext(GWorldPtr theGWorld)
{
   TQ3PixmapDrawContextData  myPixmapDCData;
   TQ3ColorRGB               clearColor;
   PixMapHandle              hPixMap;
   Rect                      srcRect;

   Q3ColorRGB_Set(&clearColor, 1, 1, 1);

   // Fill in the draw context data.
   myPixmapDCData.drawContextData.clearImageState = kQ3True;
   myPixmapDCData.drawContextData.clearImageMethod =
      kQ3ClearMethodWithColor;
   myPixmapDCData.drawContextData.clearImageColor = clearColor;
   myPixmapDCData.drawContextData.paneState = kQ3False;
   myPixmapDCData.drawContextData.maskState = kQ3False;
   myPixmapDCData.drawContextData.doubleBufferState = kQ3False;
   hPixMap = GetGWorldPixMap(theGWorld);
   LockPixels(hPixMap);
   srcRect = theGWorld->portRect;
   myPixmapDCData.pixmap.width  = srcRect.right  - srcRect.left;
   myPixmapDCData.pixmap.height = srcRect.bottom - srcRect.top;
   myPixmapDCData.pixmap.rowBytes  = (**hPixMap).rowBytes & 0x7FFF;
   myPixmapDCData.pixmap.pixelType = kQ3PixelTypeRGB32;
   myPixmapDCData.pixmap.pixelSize = 32;
   myPixmapDCData.pixmap.bitOrder  = kQ3EndianBig;
   myPixmapDCData.pixmap.byteOrder = kQ3EndianBig;
   myPixmapDCData.pixmap.image     = (**hPixMap).baseAddr;

   return Q3PixmapDrawContext_New(&myPixmapDCData);
}
```

Listing 6. Creating the camera

```
TQ3CameraObject MyNewCamera(WindowPtr theWindow)
{
   TQ3ViewAngleAspectCameraData perspectiveData;
   TQ3CameraObject              camera;

   TQ3Point3D     from = { 0.0, 0.0, 13.0 };
   TQ3Point3D     to   = { 0.5, 0.5, -1.5 };
   TQ3Vector3D    up   = { 0.0, 1.0, 0.0 };
   float          fieldOfView = 0.523593333;
   float          hither      = 0.001;
   float          yon         = 1000;

   perspectiveData.cameraData.placement.cameraLocation = from;
   perspectiveData.cameraData.placement.pointOfInterest = to;
   perspectiveData.cameraData.placement.upVector = up;
   perspectiveData.cameraData.range.hither = hither;
```

(continued on next page)

Listing 6. Creating the camera *(continued)*

```
    perspectiveData.cameraData.range.yon = yon;
    perspectiveData.cameraData.viewPort.origin.x = -1.0;
    perspectiveData.cameraData.viewPort.origin.y = 1.0;
    perspectiveData.cameraData.viewPort.width  = 2.0;
    perspectiveData.cameraData.viewPort.height = 2.0;
    perspectiveData.fov = fieldOfView;
    perspectiveData.aspectRatioXToY =
       (float) (theWindow->portRect.right - theWindow->portRect.left) /
       (float) (theWindow->portRect.bottom - theWindow->portRect.top);
    camera = Q3ViewAngleAspectCamera_New(&perspectiveData);

    return camera;
}
```

amount of lighting applied to a particular surface will be the sum of the lighting from all sources. There are four light types:

- *Ambient* — This is the amount of light added to all surfaces in a scene. Since this light type has no location, it doesn't cast shadows.
- *Directional* — Sometimes referred to as an "infinite" light, this light source emits parallel rays of light in a specific direction. The intensity of this light source doesn't change as the distance from the light changes.
- *Point* — This light source emits rays of light in all directions from a particular point location. A point light is *attenuated*, meaning that the intensity of the light decreases as the distance from the light increases; QuickDraw 3D provides a set of constants to control this behavior.
- *Spot* — This type of light emits a circular cone of light from a point source in a particular direction. A spot light is attenuated both by the distance from the source and by the position across the cone; the intensity of light at the center of the cone is greater than the intensity at the edge of the cone.

Listing 7 shows an extract from our sample's MyNewLights routine; here we create a point light and add it to a light group.

THE VIEW

Once you've added the light to a group, you can associate the group with the View object for your scene. A View object keeps track of the information necessary to render an entire scene, tying together the different parts of QuickDraw 3D. In our simple example it ties together the draw context, camera, lights, and renderer. Listing 8 shows the code we use to create the View object for the Box program.

THE RENDERING LOOP

All drawing must be done in a rendering loop. This is necessary because we don't know in advance how much memory is required to render a complex model. The loop should fit neatly into your application, because most Macintosh applications will localize drawing in the update event–handling code, which is where you'll call your rendering loop for QuickDraw 3D. A simple rendering loop will look like Listing 9.

Listing 7. Creating a point light in a light group

```
lightGroup = Q3LightGroup_New();

pointData.lightData.isOn = kQ3True;
pointData.lightData.brightness = 0.80;
pointData.lightData.color.r = 1.0;
pointData.lightData.color.g = 1.0;
pointData.lightData.color.b = 1.0;
pointData.location.x = -10.0;
pointData.location.y = 0.0;
pointData.location.z = 10.0;
pointData.castsShadows = kQ3False;
pointData.attenuation = kQ3AttenuationTypeNone;
light = Q3PointLight_New(&pointData);

Q3Group_AddObject(lightGroup, light);
Q3Object_Dispose(light);
```

Listing 8. Creating the View object

```
TQ3ViewObject MyNewView(WindowPtr theWindow)
{
   TQ3Status               myStatus;
   TQ3ViewObject           myView;
   TQ3DrawContextObject    myDrawContext;
   TQ3RendererObject       myRenderer;
   TQ3CameraObject         myCamera;
   TQ3GroupObject          myLights;

   myView = Q3View_New();

   // Create and set the draw context.
   myDrawContext = MyNewDrawContext(theWindow);
   myStatus = Q3View_SetDrawContext(myView, myDrawContext);
   Q3Object_Dispose(myDrawContext);

   // Create and set the renderer. Use the interactive software renderer.
   myRenderer = Q3Renderer_NewFromType(kQ3RendererTypeInteractive);
   myStatus = Q3View_SetRenderer(myView, myRenderer);
   Q3Object_Dispose(myRenderer);

   // Create and set the camera.
   myCamera = MyNewCamera(theWindow);
   myStatus = Q3View_SetCamera(myView, myCamera);
   Q3Object_Dispose(myCamera);

   // Create and set the lights.
   myLights = MyNewLights();
   myStatus = Q3View_SetLightGroup(myView, myLights);
   Q3Object_Dispose(myLights);

   return myView;
}
```

Listing 9. The rendering loop

```
TQ3Status DocumentDraw3DData(DocumentPtr theDocument)
{
   Q3View_StartRendering(theDocument->fView);
   do {
      Q3Style_Submit(theDocument->fInterpolation, theDocument->fView);
      Q3Style_Submit(theDocument->fBackFacing, theDocument->fView);
      Q3Style_Submit(theDocument->fFillStyle, theDocument->fView);
      Q3MatrixTransform_Submit(&theDocument->fRotation,
                                              theDocument->fView);
      Q3DisplayGroup_Submit(theDocument->fModel, theDocument->fView);
   } while (Q3View_EndRendering(theDocument->fView)
                    == kQ3ViewStatusRetraverse);
   return kQ3Success;
}
```

Recall that earlier we set up our Macintosh draw context to use double buffering; this causes all drawing to take place in the back buffer. The calls in the rendering loop draw into the active buffer, which we have set up to be the back buffer. The image data is copied from the back buffer to the front buffer (in this case the window) when Q3View_EndRendering is called.

A rendering loop for a pixmap draw context would be similar to the routine in Listing 9, except you would need to copy the data from your PixMap to the screen yourself, generally with CopyBits.

THE QUICKDRAW 3D METAFILE

Here we'll take a brief look at the architecture of QuickDraw 3D's metafile format (file type '3DMF') and at how you can provide metafile support in your application.

The QuickDraw 3D metafile comes in two forms: plain-text (ASCII) and binary. Table 1 shows the differences between these two forms. The plain-text form is more useful for debugging purposes; once your application is debugged, it's more efficient to use the binary form, which may be read and written much faster and may require less storage space on disk.

Table 1. Differences between plain-text and binary metafiles

Primitive	Plain-text	Binary
Integer	Text representation	Int 8/16/32/64
Unsigned	Text representation	Uns 8/16/32/64
Float	Text representation	Float 32/64
Object type	ObjectName	4-byte code
Sizes	Parentheses delimited	Uns32
File pointer	Label>, Label: pairs	Uns64
Enumerated types	EnumName	Uns32
Bit fields	Mask1 | Mask2 | ...	Uns32
String	"Quoted String"	Padded C string
Raw data	Hex string (e.g., 0xAB02)	Stored raw

The metafile format supports a wide range of primitive data types, including 1-, 2-, 4-, and 8-byte signed and unsigned integers and 4- and 8-byte IEEE floating-point numbers, together with a range of types for describing 3D data. In addition, metafiles may contain big- or little-endian numbers, making them ideal for storing data that may be used in a cross-platform manner.

METAFILE ORGANIZATION

There are three distinct types of metafile organization: normal, stream, and database. The organization of the file can affect both the size of the file and the time it takes to access the data in the file. Let's look at a simple example in which a single Box object is drawn four times at different positions by means of four different Transform objects, as was shown in Figure 6. The three types of organization are illustrated in Figure 7. (Note that **#** marks the beginning of a comment.) These types are as follows:

- *Normal* — This is the most compact representation. Referenced objects are listed in a Table of Contents (TOC). In our example, only the Box object is listed in the TOC. The Transform objects don't appear in the TOC because they were referenced only once. Note that random access to the file is needed to resolve references, since after reading a reference, the metafile parser needs to skip forward to the TOC, and back to resolve the references.
- *Stream* — There is no TOC, and references to objects are written as copies of the objects themselves. This may result in a larger file if a lot of object references were used, but it allows for a sequential search. A sequential search is very useful for reading from the file and imaging to a printer, since each object can be read, imaged, and disposed of. This organization is also useful as a wire protocol for imaging on remote machines.
- *Database* — Every object is logged into the TOC, even if it's not referenced. Each TOC entry contains the type of the object. Accessing the TOC lets you see all the information contained in the file without having to read in all of the file and create objects. This would be useful for creating a catalog of textures, for example.

USING METAFILES

The simplest way to access data in metafiles is to use the QuickDraw 3D API. First, there are two types of objects you need to understand:

- TQ3FileObject — Objects of this type maintain state information and provide an interface between a given file format and a Storage object. File objects are used to read and write data in metafile format from and to Storage objects.
- TQ3StorageObject — Objects of this type act as an interface to a type of physical stream-based storage (for example, memory and files). Storage objects are used to represent a piece of physical storage.

Why have this two-stage approach? The answer is that all the machine dependencies are localized in the Storage object, which allows files to be used to read and write data from differing types of physical storage with the same set of routines. For example, you can use the same File object to write to a Storage object that represents a file on your hard disk and to write to another Storage object that represents a block of memory that will be passed to the Scrap Manager.

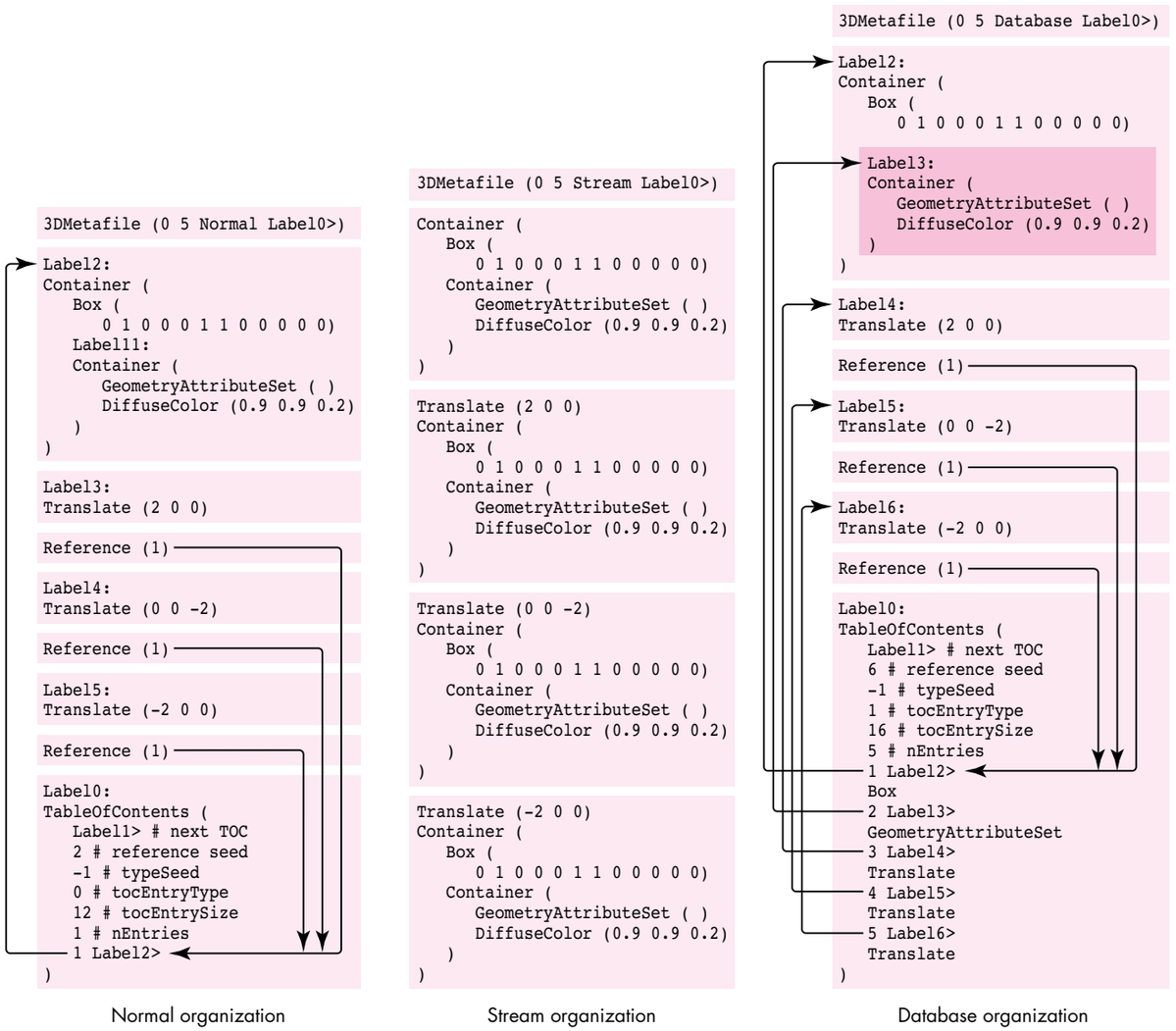


Figure 7. Three types of metafile organizations (representing Figure 6)

The usual method for using File and Storage objects is to create a new instance of a Storage object and attach it to a newly created File object using Q3File_SetStorage, as shown in Listing 10.

Reading data from metafiles. There are three routines that you can use to help with reading the data: Q3File_GetNextObjectType, Q3File_ReadObject, and Q3File_SkipObject. Listing 11 illustrates the technique used to read drawable data from a metafile. The code loops through the file, getting each object and checking to see if the object is drawable; if so, it adds the object to a group object.

Because we're isolating the implementation details of how the metafile data is stored in the Storage object that we associated with the File object at its creation time, we don't care how the metafile data we're reading is physically stored. What this means is that we could use the routine above to read data from the scrap, from a handle supplied by the Drag Manager, or from a file, as long as the storage object attached to the file is set up properly.

Listing 10. Attaching a Storage object to a file

```
TQ3FileObject MyGetNewFile(FSSpec *myFSSpec, TQ3Boolean *isText)
{
   TQ3FileObject     myFileObj;
   TQ3StorageObject  myStorageObj;
   OSType            myFileType;
   FInfo             fndrInfo;

   // We assume the FSSpec passed in was valid and get the file
   // information. We need to know the file type; this routine may get
   // called by an Apple-event handler, so we can't assume a type -- we
   // need to get it from the FSSpec.
   FSpGetFInfo(myFSSpec, &fndrInfo);
   myFileType = fndrInfo.fdType;

   if (myFileType == '3DMF')
      *isText = kQ3False;
   else if (myFileType == 'TEXT')
      *isText = kQ3True;
   else
      return NULL;

   // Create a new Storage object and new File object.
   if (((myStorageObj = Q3FSSpecStorage_New(myFSSpec)) == NULL)
         || ((myFileObj = Q3File_New()) == NULL)) {
      if (myStorageObj != NULL)
         Q3Object_Dispose(myStorageObj);
      return NULL;
   }

   // Set the storage for the File object.
   Q3File_SetStorage(myFileObj, myStorageObj);
   Q3Object_Dispose(myStorageObj);

   return myFileObj;
}
```

Writing data to metafiles. Data is written to files similarly to the way it's drawn in a rendering loop. Depending on the available memory and the complexity of the model, QuickDraw 3D may need to traverse the model in the group more than once in order to write all the data out (this is the same reason that the rendering needs to be done in a loop). As shown below, you need to preface your file-writing loop with a call to Q3File_BeginWrite, and test the value returned by Q3File_EndWrite to see if there's a need to traverse the data again.

```
Q3File_OpenWrite(file, kQ3FileModeNormal);
Q3File_BeginWrite(file);
do {
   Q3Object_Write(group, file);
} while (Q3File_EndWrite(file) == kQ3FileStatusRetraverse);
Q3File_Close(file);
```

Listing 11. Reading from a metafile

```
TQ3Status MyReadModelFromFile(TQ3FileObject theFile, TQ3GroupObject
                              myGroup)
{
   if (theFile != NULL) {
      TQ3Object      myTempObj;
      TQ3Boolean     isEOF;

      // Read objects from the file.
      do {
         Q3File_ReadObject(theFile, &myTempObj);
         if (myTempObj != NULL) {
            // We want the object in our main group only if we can
            // draw it.
            if (Q3Object_IsDrawable(myTempObj))
               Q3Group_AddObject(myGroup, myTempObj);
            // We either added the object to the main group, or we don't
            // care, so we can safely dispose of it.
            Q3Object_Dispose(myTempObj);
         }
         // Check to see if we've reached the end of the file yet.
         Q3File_IsEndOfFile(theFile, &isEOF);
      } while (isEOF == kQ3False);
   }
   if (myGroup != NULL)
      return kQ3Success;
   else
      return kQ3Failure;
}
```

GO TO IT!

QuickDraw 3D lowers the bar for application developers who want to put support for 3D data into their applications. By providing support for the features that all developers need to have in applications — geometries, metafile support, rendering, and human interface — QuickDraw 3D allows you to concentrate on the features and facilities that set your application apart.

Thanks to our technical reviewers Kent Davidson, Eiichiro Mikami, Don Moccia, and Dan Venolia, and to all the members of the QuickDraw 3D team. Special thanks to Kent and Dan for supplying information used in this article and to David Vasquez for his Viewer sample. Thanks also to the Shawn and John team (Shawn Hopwood, Apple's 3D evangelist, and our marketing weenie, John Alfano) for their input.•

Copland: The Mac OS Moves Into the Future

The Macintosh operating system has continually evolved since the days when the Macintosh was a home appliance with 128K of RAM and a floppy disk drive — but now the time has come for radical change. The next generation of the Mac OS, code-named Copland, was designed specifically to serve computers with a fast processor running several tasks and processing large quantities of data. This preview describes Copland's major features and suggests how you might get ready for it.

TIM DIERKS

Since the first Macintosh operating system and Toolbox were developed in the early 1980s, the needs of users and developers alike have evolved significantly. Newer technologies, such as MultiFinder and the PowerPC™ processor, have appeared on the scene. Users have come to expect greater ease of use, more capabilities, and enhanced productivity. Although the Mac OS has evolved along with the times, a more radical advance is now required to take advantage of the great increases in power afforded by the PowerPC processor.

Enter Copland, a new generation of the Mac OS to be released by Apple in mid-1996. Copland will provide a radically new architecture that includes technologies such as preemptive multitasking and protected memory. For one thing, it's based on a microkernel that moderates between individual tasks and arbitrates access to the machine's resources. A number of other services have been updated and improved, both to fulfill the requirements this change implies and to take advantage of the new capabilities it provides. For example, the file system has been updated to be accessible from several processes running simultaneously in several address spaces. Similarly, the networking system has been enhanced, as have a number of the auxiliary operating system managers such as the Process Manager.

With Copland will also come a number of enhancements to the user experience, including a Finder that can perform several tasks simultaneously, changes to the appearance and feel of the interface, and advances that will make it easier to locate and access information. (See "Moving the Mac OS Interface Into the Future" for more details.) All of this new functionality is glued together in a runtime model based on the Code Fragment Manager, the dynamically linked library mechanism introduced with the first Power Macintosh computers.

TIM DIERKS, who is known for having the messiest office on the Apple R&D campus, has been bumming around Apple for several years, including stints working with the Macintosh Developer Technical Support group as well as on the Copland project. Currently, he's hard at work on Apple's interactive television solution, which gives him an excuse for watching *Rocco's Modern Life* at work. He shares his office with two lizards, a corn snake, and a pinball machine — which helps explain at least some of the mess.•

MOVING THE MAC OS INTERFACE INTO THE FUTURE

BY B. WINSTON HENDRICKSON

Copland will not only radically change the foundation of the Mac OS, it will also introduce some of the most significant changes to the user experience since 1984. For the user, this means new personalization capabilities, built-in assistance with tasks, and improved access to information. For the developer, it means a robust foundation for constructing consistent and compelling interfaces that are easier to use.

The new managers in Copland directly concerned with enhancing the user experience are the Appearance Manager, the Assistance Manager, and Navigation Services. The following brief descriptions of these will give you an inkling of things to come.

THE APPEARANCE MANAGER

The Appearance Manager defines how standard user interface elements should be presented and enables users to personalize the appearance of these elements by choosing one of a number of graphical designs called *themes*. Applications can use the Appearance Manager's capabilities to draw custom interface elements in the style of the current theme.

The Appearance Manager provides you with

- a Pattern Manager that returns the appropriate PixPats for use as dialog backgrounds, control colors, and other aspects of the interface
- a set of drawing primitives for rendering common interface elements such as window title bars and dialog separator lines
- new standard interface elements, including sliders, progress indicators, and icon buttons
- event notification when the current theme is changed, allowing you to resync any cached appearance data

To prepare now for Copland's dynamic system appearance, be sure not to make assumptions about interface specifics (such as assuming that the dialog or menu background is white). Also, don't hard-code the appearance of your application's interface elements; for instance, avoid the use of custom definition procedures wherever possible.

THE ASSISTANCE MANAGER

The Assistance Manager supports the implementation of active user assistance, enabling the computer to accomplish specific tasks with little or no direction from the user. The Assistance Manager provides support for the following:

- task delegation, allowing the creation and management of automated activity controlled by a condition or event, such as time or mail delivery
- the ability to create templates from which tasks are created and executed
- user "interviews" for task configuration, based on Apple Guide's interaction engine

Since this active assistance is built on existing technologies, you can start to prepare for it today. The first and most important step is to make your application scriptable, so that it can be automated. Second, you should provide task-based assistance using Apple Guide. Finally, if your application provides any task delegation, you should factor out the related code now so that you can take advantage of the Assistance Manager under Copland.

NAVIGATION SERVICES

Navigation Services replaces System 7's Standard File Package, providing a set of tools for opening, saving, and naming documents as well as for navigating a hierarchical information space containing such documents. These tools will increase consistency between applications and the Finder and will enable integration with the Finder's new and improved searching mechanism. The new capabilities provided by Navigation Services include these:

- support for a favorite items list, file list position recall (rebound), and a more intuitive browser
- the ability to browse diverse containers such as a mailbox and return a general-purpose reference value to documents in those containers
- one-step calls for common operations such as selecting a file or directory
- support for easy customization, including an extensible list of information "panels" (based on Copland's new dialog panels)
- automatic dialog layout adjustment for active script systems

You can do a few things now to get ready for Navigation Services. First, when customizing the Standard File dialog, render only inside your dialog items, as they may be rearranged. Second, don't assume you're drawing into the desktop port, because you won't be. And finally, don't try to control Standard File by posting events to dialog items; use the documented interface instead.

Because of the large number of advances and changes in Copland, some software will be incompatible. For instance, applications that have inappropriately incestuous relationships with the operating system might run into problems. But there *are* some things you can do now to prepare yourself for this release and ensure that your applications will be as compatible as possible. I'll tell you about those things as I give you a tour of Copland's microkernel, runtime model, File Manager, and I/O architecture.

THE MICROKERNEL: A NEW FOUNDATION

The present Macintosh operating system is somewhat too trusting: it doesn't take charge of restricting software's actions or balancing the use of the machine's resources. Any piece of code can write all over memory, retain control forever, and even turn off interrupts for any period of time. This model, while once appropriate, has shortcomings in a computer with a fast processor running several tasks and processing large quantities of data.

That's where Copland's microkernel, developed by Apple specifically for use in the Mac OS, comes in. The microkernel serves as a referee for the system. It moderates between many individual tasks so that none can hog the processor and so that special code need not be implemented to share it. It also arbitrates access to the machine's resources, including memory, preventing software from being able to see or change data unrelated to its task.

The microkernel provides a number of services — most of them familiar to those conversant with kernel-based systems — including task control, address space management, virtual memory management, interrupt control, synchronization primitives, and intertask messaging. These services, which we'll look at in more detail in the following pages, serve as the basic building blocks of the system. In most cases, your software won't use any of these kernel services directly but will instead take advantage of them through other APIs — APIs that are part of System 7 but that have been reimplemented in Copland.

TASK CONTROL

The Copland kernel provides full support for a variety of tasking services. While applications will normally be cooperatively scheduled by the Process Manager (just as in System 7), applications will also be able to create tasks that are preemptively scheduled. Preemptive tasks are scheduled in the order of their assigned priority and according to kernel scheduling rules; the Process Manager doesn't manage them in the way it does applications. Such tasks behave as threads behave in other systems. At any time, almost anything in the system — including the currently running application — can be preempted to run such a task. Interrupt handlers can't be preempted, however.

You'll be able to set the priority of preemptive tasks that you create; higher priority tasks will run in preference to lower priority ones. By giving an I/O-intensive task higher priority than your main application thread, you'll gain performance very similar to that made possible today by chained completion routines. During the relatively long I/O delays when your task is blocked, your main application thread will execute freely. Whenever its I/O requests do complete, the task will regain control immediately so that it can issue its next I/O request, resulting in maximum throughput without unnecessary blocking of other computing tasks. Similarly, you'll be able to assign a higher priority to general application tasks than to background tasks that can afford to wait or proceed slowly while the machine is in use (such as a background renderer for a network-distributed 3D software package). This will ensure responsiveness in your application and allow you to use otherwise idle CPU time.

Chained completion routines are discussed in the article "Asynchronous Routines on the Macintosh" in *develop* Issue 13.•

To prepare for this opportunity, you can work to make your application easier to factor. If you remove dependencies between different portions of your application, it'll be easier to take full advantage of Copland's multitasking capabilities. When a Copland preemptive thread runs, the file system, networking, and device I/O will be available, similar to the environment when a Time Manager or Deferred Task Manager task runs in System 7. One addition is that synchronous calls can be made; your thread will just block until the I/O has completed.

ADDRESS SPACE AND VIRTUAL MEMORY MANAGEMENT

In System 7.5, there's only one address space. A particular address always refers to the same part of memory, and data located there can be accessed by every part of the operating system. In Copland, by contrast, multiple address spaces can be created, allowing code and data to be hidden from some processes. For compatibility reasons, Macintosh applications will continue in this release to share a single address space, while components of the operating system and third-party software can create fully protected memory areas in which code that's not dependent on the Macintosh Toolbox can execute. The kernel, the file system, and several other components will create such areas to protect their private data structures.

Each address space is divided into areas. An area can be either private (accessible only to tasks executing in that address space) or global (accessible at the same location in all address spaces). In addition, an area can be either read/write to all tasks (most global areas fit into this category) or read-only in user mode and read/write in supervisor mode. (Most code runs in user mode; only code that needs special abilities, such as drivers and parts of the operating system, runs in supervisor mode.) This latter protection is used for most kernel and file system data structures; they're located in global memory for fast and easy access (without the need to switch to another address space) but can't be damaged by code executing in user mode. Only the privileged clients of the system can change these structures.

In addition to having the ability to map RAM into a variety of address spaces, Copland also uses virtual memory to provide more room than is available in physical RAM, moving data between RAM and disk as needed. In fact, virtual memory is always on. It's dramatically better than System 7's virtual memory in these ways:

- The new file system and better integration between the file system and virtual memory will improve performance.
- Your application will be able to provide hints to the operating system to allow it to tune for best performance. For example, you'll be able to tell it that you're about to access a significant portion of a large array, and it will asynchronously begin to bring in the pages that the array resides on.
- Fewer limitations on what can be paged will increase the available RAM for the system. In System 7, the system heap is always held in memory and can't be paged onto disk. In Copland, virtually all of the system — aside from the kernel, the file system, and the disk driver — will be pageable.
- The disk cache will be integrated with the virtual memory system, and the size of the disk cache will dynamically adjust based on current operations in order to optimize performance.

- Best of all, Copland's virtual memory will dynamically expand the amount of address space in the system as needed, giving users much more flexibility than in System 7, where they must decide beforehand how much memory they'll need, adjust the Memory control panel accordingly, and reboot. In Copland, if they need to open additional applications, they can do so without going through any rigmarole; space will be created on the fly to support their needs, provided that sufficient disk space is available for use as a backing store.

Because Copland will make available a full gigabyte of address space in which to run applications (subject, of course, to the limits imposed by the amount of disk space available for paging), two other limitations of the System 7 memory allocation system should be alleviated. First, Copland will reduce (though not entirely eliminate) the need for the user to configure and reconfigure the application's memory partition to accommodate changing needs. Also, problems with applications fragmenting the available memory for launching more applications should be eliminated. Thus, no longer will users always have to deal with the complex issues of memory allocation and organization to make best use of their machines.

In addition to swapping space, Copland will support memory-mapped files. This technology allows an application to map a file against an area of address space; accessing locations in the address space causes the appropriate portions of the file to be read into RAM. In system software version 7.1.2, this technology is used by the system for paging PowerPC code when virtual memory is on, but it's not available to applications. In Copland, it will be available to applications; data files can be mapped for read-only or read/write access. An application will be able to read a document just by walking through the address space without having to manually stream it into buffers.

To be prepared for Copland's use of virtual memory, applications today should be able to operate well in a virtual memory environment. For purposes of performance, this means keeping a tight locality of reference; code that uses contiguous data structures rather than structures spread all over memory will require fewer pages to be resident for any operation. Also, take care when allocating variable-sized buffers. Don't always attempt to allocate the largest possible buffer, sizing it down till it fits — a popular but potentially wasteful habit; instead, cap buffers at points beyond which they won't gain from more RAM. For example, if reading a file, you might cap the size of a data buffer at 64K, because there's little to gain by reading the file in larger chunks.

INTERRUPT CONTROL

On Power Macintosh computers running System 7.5, interrupts are handled by the 680x0 instruction emulator, incurring a large overhead. Even if the overhead of the actual interrupt handler is small, a significant price is paid in invoking the emulator, especially if a mostly native application is executing (in which case the emulator has to be pulled into the cache on each interrupt, then flushed out as native code is reloaded after the interrupt returns). Since a Macintosh can easily take several hundred interrupts per second (thanks to interrupts caused by video retrace, the old-style VBL Manager, ADB, and the like), this can have a significant performance impact.

By contrast, Copland's I/O system, including interrupt handlers, is entirely native; this, along with an improved architecture, should mean significantly lower interrupt latency and better overall performance. Because of the flexibility of the execution control available to the kernel, it will be easy for an interrupt handler to do the absolute minimum to deal with an interrupt (often all that's required is to acknowledge it). After control is returned from the interrupt, another piece of code

called a *secondary interrupt handler* can be invoked; although under the same constraints as a hardware interrupt handler, this handler results in the best system performance by enabling the soonest possible exit from the hardware handler. If significant processing needs to be done right away, this secondary handler can wake a high-priority task to do that work, thus keeping the system from being bottlenecked by any individual set of handlers.

Some native interrupt functionality will be delivered before Copland; see the article "Creating PCI Device Drivers" in this issue of *develop* for more information.•

SYNCHRONIZATION AND INTERTASK MESSAGING

As mentioned earlier, applications under Copland will be able to create tasks that are preemptively scheduled. You'll be able to assign priorities to your preemptive tasks, but this in itself won't prevent the tasks from preempting each other at inopportune times. What you'll need in order to ensure correct behavior from your tasks is a mechanism to synchronize access to shared resources.

Copland provides several synchronization mechanisms, each useful in a different situation. Any operating systems textbook includes a variety of them, and most can be implemented in combination with others. The ones implemented in Copland — atomic operations, simple locks (mutual exclusions, or *mutexes*), read/write locks, event flags, and event queues — are meant to efficiently solve problems common in Macintosh applications and the Mac OS and to provide building blocks to implement other synchronization mechanisms if necessary.

The kernel also has an intertask message system that provides data transfer as well as synchronization, although for basic synchronization of shared data it's probably more than you'll need. It can move arbitrary amounts of data across address spaces synchronously or asynchronously, by value or by reference. In cases in which the system uses messages to implement functionality, the message will be hidden inside an API library, so you generally won't have to deal with the details of how the message system works.

THE RUNTIME MODEL

Copland's runtime model is based on the Code Fragment Manager (CFM). Instead of a monolithic binary file, the operating system consists of a number of individual libraries that combine to provide the API and system functionality. This mechanism allows software to be built in a much more modular fashion. In addition, the CFM provides a much more consistent context and activation model than does the statically linked, trap-dispatched runtime model used in 680x0-based computers. Rather than having to be concerned with setting up and restoring the A5 register to provide access to data when executing in an interrupt task, the CFM provides a standardized transition to make sure the data appropriate to the executing code is always available.

Although applications under Copland will continue to be cooperatively scheduled within a single address space, developers can, as already mentioned, use kernel services to create tasks that take advantage of preemptive multitasking and protected memory. Tasks running in address spaces outside the Toolbox will have a limited number of services available to them, similar to those available to background-only applications in System 7: they'll be able to allocate memory, communicate with other processes, and use the kernel services, the file system, and the network, but they won't be able to draw on the screen or interact directly with the user. Tasks that are I/O or computation intensive running in separate address spaces will get the benefits of preemptive multitasking, and protected memory will separate these tasks from applications, providing an increased level of stability and reliability.

For example, Copland will include an implementation of the personal file sharing server that runs in a separate address space. This allows the server — which takes networking traffic and converts it to file system calls, serving files from the local drive to remote clients — to share the system with the user and foreground processes as efficiently as possible. It will use concurrent I/O to interleave its requests with those of foreground processes, and it will get compute time at any moment when the processor is otherwise idle, even if the foreground process is waiting for a page fault to be completed.

Figure 1 illustrates Copland's architecture, showing the separation between the Toolbox environment, other tasks, and the operating system. Core portions of the operating system such as the kernel and the file system run in an address space that's protected from the Toolbox environment and other processes; similarly, the Toolbox environment is protected from other applications running in their own address spaces. Each of these areas, including the kernel and other services, can have one or several threads of execution; the kernel preemptively switches between them. Some services, such as the file system, can have several active threads of control, each responsible for a single outstanding file system request. All applications running in the Toolbox environment, including the Finder, run as a single task, inside a single address space. However, Copland-aware applications can use preemptive threads to best handle CPU- and I/O-intensive tasks.

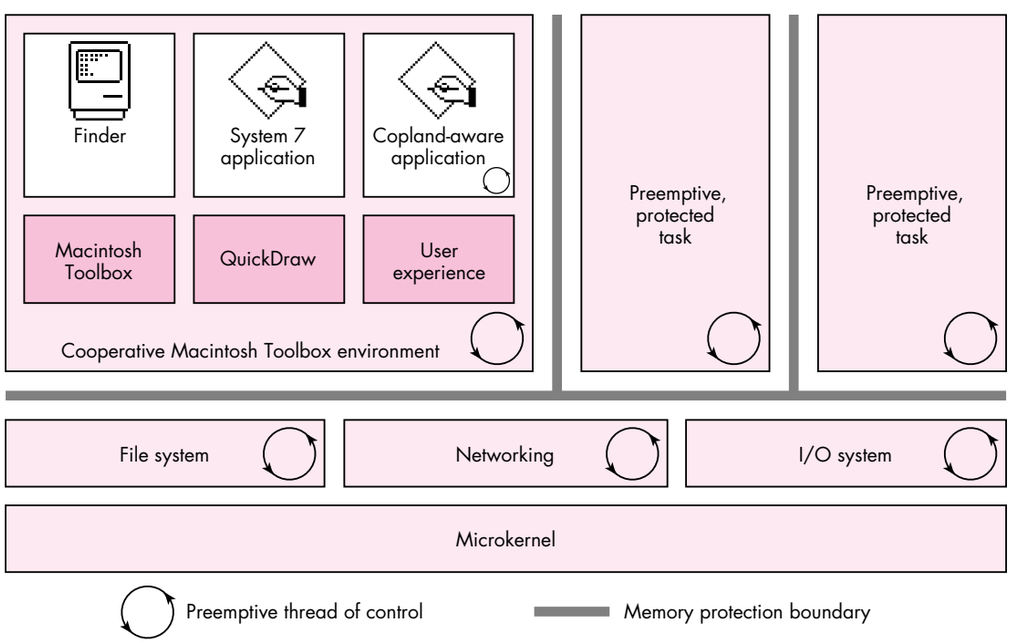


Figure 1. The Copland architecture

With the new runtime model will come a number of new ways to monitor and modify system operations without patching, which is clunky and difficult to maintain. Currently, extensions have no formal presence in the system; they live in the cracks between the system and applications. By providing a better-defined environment for extensions, Copland will make them easier to write and more stable. For example, extensions currently must patch the file system — or use the inefficient alternative of polling — if they need to track file usage; the new File Manager will let software install notification procedures that can be called whenever a particular event, such as creating or renaming a file, occurs. Patching will still be available; a new Patch

Manager will allow software to patch CFM entry points and will give much more control over the installation and removal of patches, including where they fall in the chain of execution.

To run under Copland, extensions you've developed will need to be revised. You'll make the transition easier for users if your applications that ship with extensions are able to run without these extensions installed. Also, desk accessories will no longer be supported in Copland; if you depend on any desk accessories, you should rewrite them as small applications.

THE FILE MANAGER

When the Macintosh was first introduced, it had a flat file system that was appropriate only for floppy disks. Since then, a number of advances have been made, including the introduction of the hierarchical file system (HFS) in 1986. But the System 7 File Manager has these limitations:

- The File Manager implementation is closely tied to the HFS volume format, making it difficult to support other volume formats.
- As HFS volumes grow in size, they become less efficient due to HFS's limitation of 2^{16} (65536) allocation blocks, making it difficult to extend the HFS volume format.
- The File Manager can process only one operation at a time, restricting performance when several tasks are contending for file I/O.
- The File Manager is implemented entirely in 680x0 assembly code, limiting performance on the Power Macintosh platform.

Copland will introduce a new File Manager that addresses these limitations, directly or indirectly. For instance, the new File Manager has been divorced from HFS implementation details and thus imposes no limitations on volume formats; arbitrary volume formats can be developed and plugged in. This will allow the Macintosh to properly support any file system, including ones that feature larger volumes, more files, or larger files than the HFS disk format. It will even be possible to create components that provide access to distributed network file systems or other data stores that don't easily map onto the HFS block storage model. In fact, HFS itself will be implemented as one of these plug-in modules.

The new File Manager will also support concurrent data transfer, so that several file system requests can be in progress at any one time. This will dramatically increase throughput in a number of cases. For example, copying files from a fast file server to a hard drive now involves an entirely serialized read over the network followed by a write to the hard drive; in Copland, the read and write operations can be overlapped, so the copy can be completed in as little as half the time. Throughput will even be increased in cases where two accesses share a communications channel, such as transfers involving several devices on the local SCSI bus or several file servers, because most communications channels won't be filled by a single device. A significant portion of the time it takes to read or write data to a SCSI disk is spent waiting for the disk, not actually transferring data; in the new model, that time can be used to transfer data to or from a different device.

Another real enhancement to the file system is the introduction of a new API, designed to be easy to learn and use. A new API was necessary because the new file system supports files and volumes larger than 2^{31} bytes, meaning that more than 32

bits are needed to store various values. The System 7 File Manager API has already been through several stages of evolution, from the original file system calls through the HFS calls to the calls taking FSSpecs in System 7; the new API is in lieu of reworking it one more time. In addition, the HFS API is composed of a number of calls that take huge parameter blocks where it isn't obvious which fields need to be set to what at any moment; wherever possible in the new API, parameter blocks have few fields and can easily be reused.

THE I/O ARCHITECTURE

With Copland comes a new I/O architecture designed with the following objectives in mind:

- improved performance
- ability to support concurrent data transfer
- sufficient abstraction to enable Apple to license the Mac OS to manufacturers who build a variety of hardware configurations
- increased ease of configuration
- independence from the 68000 microprocessor and its runtime model

The I/O architecture is organized around a number of services, each of which can be associated with a set of drivers in a unit known as a *family*. For example, the SCSI Manager can be associated with a number of SCSI interface modules (SIMs), each of which describes a single SCSI bus. Similarly, each of the drivers in the block storage family can provide block storage functionality to a file system agent. Drivers in the volume format family (through the File Manager) will manage a number of volume format agents, including the HFS agent.

Thus, the I/O architecture is structured in a hierarchy of layered components, which pass control and data among themselves. For example, an application might make a file system call, which would be passed to the HFS file system agent, which would then make a request of the appropriate block device driver. That driver could then use the SCSI Manager to transfer data to a SIM, over a SCSI bus, to a specific SCSI device. This modularity means that other data flows are easily constructed. For example, the file system request might be passed to an AFP file system agent, which would result in data being transferred over the network using Open Transport. (AFP stands for AppleTalk Filing Protocol — the protocol used to talk to AppleShare file servers.) These relationships are diagrammed in Figure 2.

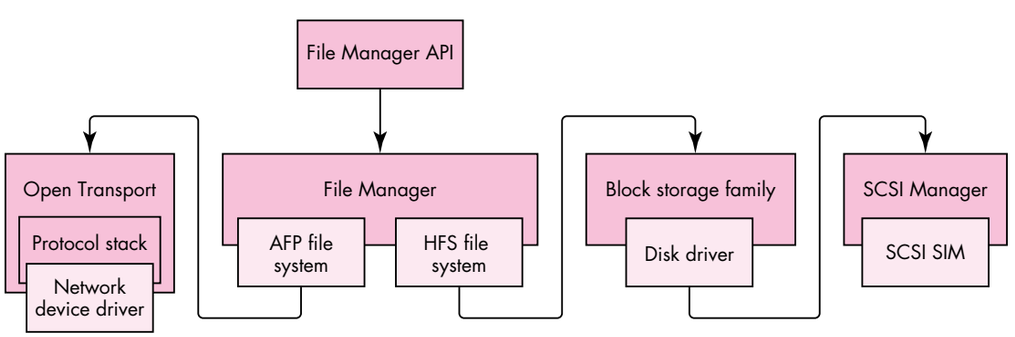


Figure 2. An example of relationships and control flow in the new driver model

This layered architecture permits a flexible dependency chain, where no component has too much knowledge about the implementation details of its dependencies or its clients. A block storage driver, for instance, doesn't need to know the details of the SIM's SCSI bus implementation or which volume format it's being used for; it just passes requests up and down the chain. This modular architecture should make it easier for Apple and developers to introduce new ways of solving problems.

The driver model for PCI cards, described in this issue of *develop* in the article "Creating PCI Device Drivers," was designed with this new I/O architecture in mind. Copland will support drivers developed for PCI cards in accordance with the guidelines presented in that article, so you would do well to familiarize yourself with them. Old drivers will need to be revised because they read and write to hardware locations directly; protection in the new kernel requires that this ability be reserved to specially privileged software. But this doesn't mean that everything packaged into Device Manager drivers will break. The Device Manager will continue to support code packaged as a driver that doesn't actually touch hardware and that isn't otherwise incompatible with Copland, such as drivers that some programs use for interapplication communication.

Note also that Open Transport, which is now available for development on System 7.5, will be the native networking protocol in Copland. Applications that use it will make best use of the native networking stack and will be prepared to run in a separate address space.

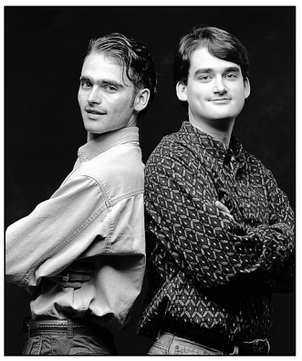
WHERE DOES THIS LEAVE YOU?

The transition to Copland will be the most significant operating system transition in the history of the Macintosh. You can make this transition easier for yourself and users if you do these things:

- Begin factoring your applications now.
- Make sure your applications can operate well in a virtual memory environment.
- Avoid reading or modifying low-memory globals and system data structures if at all possible in your applications.
- Avoid patching.
- Ensure that any of your applications that ship with extensions can run without these extensions installed.
- Rewrite as small applications any desk accessories you depend on.
- Familiarize yourself with the new driver model for PCI cards.
- Make use of Open Transport for networking.

With its vastly expanded functionality, Copland will offer an unprecedented number of new opportunities for developers. In providing a stronger foundation for third-party products and future Mac OS releases, Copland will lay the groundwork for years of advances on the Macintosh platform.

Thanks to our technical reviewers Jeff Cobb, Dave Evans, John Iarocci, Wayne Meretsky, Mike Neil, Steve Szymanski, and Russell Williams. Special thanks to Russell Williams for information on synchronization services for preemptively scheduled tasks.



BALANCE OF POWER

MacsBug for PowerPC

DAVE EVANS AND JIM MURPHY

Where would we be without our venerable friend MacsBug? Those long debugging sessions, with soda cans piling up, would be so lonely. Like a trusted buddy, only a key press away, MacsBug has helped us solve the toughest problems and has taken us into the very core of the Macintosh.

Yet now the core has changed. The Macintosh has transmuted into a RISC powerhouse, and its new runtime environment is a foreign place to the old MacsBug. But instead of leaving our friend in the past, Apple is developing a next-generation MacsBug. The next MacsBug, version 6.5, supports PowerPC debugging with the same commands that you already know. An early release of the new MacsBug is provided on this issue's CD.

A QUICK LOOK

When we're confronted with a MacsBug display that heralds the beginning of a new debugging challenge, our first questions are "Where are we?" and "How did we get here?" MacsBug has always provided tools and information to help us quickly answer these questions. The stack crawl commands, for example, show not only what code you're executing but also how you got there. On a Power Macintosh, however, the old MacsBug often failed at tasks like the stack crawl. The PowerPC architecture changes fundamental structures — such as stack frame formats — and introduces new hurdles like mixed-mode switches and native code. The key goal of the new MacsBug is to restore the functionality you need in order to debug on a Power Macintosh.

The new MacsBug adds many features to satisfy this goal. It lets you disassemble, trace, and step through PowerPC code. You can set breakpoints in PowerPC code, and easily find out what code fragment you've interrupted and what native symbol is closest. And you can display a stack crawl that includes both PowerPC and 680x0 stack frames. Let's take a closer look at these and other new features.

DISASSEMBLING POWERPC CODE

The new MacsBug will disassemble both PowerPC and 680x0 code. Some commands are now sensitive to which type of code is executing. The **td** (total display) command will show the PowerPC register set during execution of native code, and the 680x0 emulated registers during execution of 680x0 code. When used without an address, the **il** and **ip** commands will disassemble the code currently being executed, whether PowerPC or 680x0; however, when an address is specified with **il** and **ip**, MacsBug will disassemble 680x0 code at the specified address.

We've introduced a few new commands to force disassembly of PowerPC native code. They include the following:

ilp	Instruction list of PowerPC code
ipp	Instruction half-page list of PowerPC code
idp	Instruction display one word as PowerPC code
dhp	Disassemble hex as PowerPC code
brp	Set breakpoint in PowerPC code

To demonstrate, we'll set a native breakpoint in the Memory Manager with **brp __SetHandleSize**. After we type **g** and wait for a second, MacsBug interrupts with this display:

```
PowerPC breakpoint at 000A06C4 __SetHandleSize
```

We type the **il** command and see the following (notice the familiar breakpoint bullet and the asterisk showing the current program counter location):

```
Disassembling PowerPC code from 000A06C4
  __SetHandleSize
     +00000 000A06C4 ·*mflr    r0
     +00004 000A06C8   stmw    r26,-0x0018(SP)
     +00008 000A06CC   stw     r0,0x0008(SP)
```

DAVE EVANS denies rumors that he modeled for the Mac OS logo. Although he works in the Mac OS team at Apple and does perpetually smile — and his favorite color is blue — Dave could never sit still long enough for the pose. You're more likely to spot him racing off in his hunter green Jeep Sahara. He'll probably be late, after fitting too many activities into his day, and he'll certainly be en route to some new adventure.•

JIM MURPHY (AppleLink MURPH, Internet murph@apple.com) dislikes one thing more than the Pittsburgh Penguins hockey club without superstar center Mario Lemieux, and that's a Macintosh without MacsBug. When he's not dodging a tedious meeting at Apple, blearily staring at a logic analyzer, or hacking the Mac OS boot process, he can be found plying the backroads of the Santa Cruz mountains in his trusty Miata.•

```
+0000C 000A06D0   stwu    SP,-0x0170(SP)
+00010 000A06D4   lwz     r30,0x0000(RTOC)
+00014 000A06D8   addic   r7,SP,72
+00018 000A06DC   lwz     r26,0x0000(r30)
+0001C 000A06E0   ori     r31,r4,0x0000
+00020 000A06E4   stw     r7,0x0000(r30)
+00024 000A06E8   ori     r29,r5,0x0000
+00028 000A06EC   lwz     r3,0x0004(RTOC)
+0002C 000A06F0   bl      __HSetStateQ+013C
+00030 000A06F4   lwz     RTOC,0x0014(SP)
+00034 000A06F8   ori     r28,r3,0x0000
+00038 000A06FC   addic   r3,SP,72
```

From here we can display any PowerPC register, step or trace through the code, or ask for our location using the improved **wh** command. The **wh** command now lists which code fragment contains the address, using the code fragment export symbols to display the nearest earlier symbol. In this example, we interrupted at __SetHandleSize in the MemoryMgr code fragment, so **wh** produces this:

```
Address 000A06C4 is in the System heap at
   00002800 at __SetHandleSize
This address is within the code fragment
   named “MemoryMgr”.
It is 000021D4 bytes into this heap block:
  Start    Length       Tag  Mstr Ptr  Lock
· 0009E4F0 00009050+04   R   00002BF0   L
```

CRAWLING AROUND TOGETHER

MacsBug version 6.5 distinguishes between PowerPC and 680x0 stack frames but will display them as a unified stack crawl. This makes it very easy to determine where you are and where you've been. Since the PowerPC version of the Mac OS still contains a substantial amount of 680x0 code, you'll often see references to 680x0 callers in the stack crawl. Otherwise, expect to see the nearest earlier code fragment symbol to each caller. A sample stack crawl, displayed with the **sc** command, would look like this:

```
Frame     Caller    ISA
01D87FB2  00013CA0  PPC  'MyApp'+1A0
01D87F4A  00043050  PPC  'MyApp'+2F450
01D87EF0  4085FF06  68K  ComponentDispatch+26
01D87EC8  4085FFE6  68K  ComponentDispatch+106
01D87E50  00063144  PPC  'NativeComponent'+40
```

Here you can see that we're executing a native application with the exported main symbol MyApp. The application calls a subroutine at MyApp+1A0, leaving the first stack frame that we see here. Then at MyApp+2F450 the subroutine appears to call the 680x0-based Component Manager. We assume this because the next two stack frames are marked “68K” and appear within the ComponentDispatch code. The Component Manager then calls native code with the symbol NativeComponent. The last frame is generated when NativeComponent calls a subroutine.

EVEN MORE NEW FEATURES

Besides support for native Power Macintosh debugging, the new MacsBug adds other exciting new features, including:

- multiple debugger preferences files
- better ROM symbols using ROM map files
- an improved dcmd format
- significant performance improvements on 68040-based Macintosh computers

We once used ResEdit to construct a single MacsBug preferences file with all our favorite dcmds and templates. Those days are over: the new MacsBug version will load up to 32 preferences files that you provide. And if you haven't discarded the ROM list files provided with MPW, you can build them and use the resulting ROM map files with the new MacsBug. Put the map files in the new MacsBug preferences folder to use the map's symbol information. When disassembling or displaying addresses in the ROM, MacsBug will then display better symbols.

The improved dcmd format adds new action calls and an expanded parameter block structure, which provides full access to the PowerPC register set and machine state. Although this doesn't give you special support for developing PowerPC-native dcmds, you now have access to valuable internal state information. With this new information, your dcmds can do things that were previously reserved for Apple dcmds, such as **tdp** (total dump PowerPC), which was introduced in the article “Debugging on PowerPC” in *develop* Issue 17. This dcmd, among others from Apple, has intimate knowledge of how the PowerPC system software works. With the new dcmd format, this intimate knowledge is no longer necessary since MacsBug provides access to everything you'll need.

The new dcmd format has been designed with maximum flexibility in mind. Your dcmds can check at run time for the availability of MacsBug features. When new callbacks are defined, you can check whether MacsBug supports the calls, rather than being tied to a specific MacsBug version.

On 68040-based Macintosh computers, MacsBug will perform most tasks much faster, and with an important side effect. The new MacsBug doesn't flush the 68040 processor cache nearly as often, greatly improving

performance for most commands. The key side effect is for bugs related to cache flushing: in the past, MacsBug would flush the cache frequently enough to make these bugs hard to reproduce when you're stepping or using breakpoints; the new MacsBug, with its selective flushing, should allow you to more readily reproduce this type of problem.

IT'S LOG, LOG, LOG!

Here at Apple, bugs are usually found by test engineers using automated scripts and manual testing. When they find a bug in our code, we're rarely nearby to analyze it immediately. Therefore, they collect key information so that we can later reproduce the problem. One useful piece of information they collect is called a *standard log*.

The standard log is a sequence of MacsBug commands that are run after a crash or interrupt — for example, a register display and stack crawl. These commands are logged in a text file, and the result is copied into a report that describes the problem. Having this information in the problem report saves significant time and sometimes provides enough detail to resolve the problem immediately. MacsBug version 6.5 makes this log useful on the Power Macintosh. Its improved stack crawl, native disassembly, and PowerPC register display provide key information for later analysis. We recommend that you incorporate a standard log into your testing process; you'll find ours included as the StdLog macro.

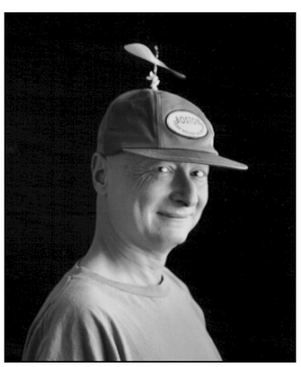
DON'T HESITATE

You can read the release notes provided on the CD for detailed descriptions of these and other improvements. We hope you'll install the new version of MacsBug without delay!

Thanks to Brian Bechtel, Dave Lyons, and Greg Robbins for reviewing this column.

Creating PCI Device Drivers

The new PCI-based Power Macintosh computers bring with them a subset of the functionality to be offered by the next generation of I/O architecture. New support for device drivers makes it possible to develop forward-compatible drivers for PCI devices, while at the same time making them much easier to write and eliminating their dependence on the Macintosh Toolbox. Key features of the new driver model are described in this article and illustrated by the accompanying sample PCI device driver.



MARTIN MINOW

Writing Macintosh device drivers has always been something of a black art. Details of how to do it are hidden in obscure places in the documentation and often discovered only by developers willing to disassemble Macintosh ROMs and system files. But this art that's flourished for more than a decade is about to get a lot less arcane.

The PCI-based Power Macintosh computers are the first of a new generation of computers with support for a driver model that's independent of the 68000 processor family and the Macintosh Toolbox. Existing 680x0 drivers will continue to work on the PCI machines (although this may not be true for future systems); a third-party NuBus™ adapter enables the use of existing hardware devices and drivers without change. But drivers for PCI hardware devices must be written in accordance with the driver model supported in the new system software release, which makes them simpler to develop and maintain.

This article will give you an overview of the new device driver model, without attempting to cover everything (which would fill a book and already has). After discussing key features, it suggests how you might go about converting an existing driver to drive a PCI device. The remainder of the article looks at some of the individual parts of a forward-compatible PCI device driver. The sample code excerpted here and included in its entirety on this issue's CD offers a complete device driver that illustrates most of the features of the new driver model. Of course, you won't be able to use the driver without the hardware, and you'll need updated headers and libraries to recompile it.

How to write device drivers for PCI-based Macintosh computers is explained in detail in *Designing PCI Cards and Drivers for Power Macintosh Computers*.•

MARTIN MINOW recently sneaked away to England from his job at Apple for a (too) brief vacation. The high point was at the Kew Bridge Steam Museum outside of London, where he stood inside the oldest, or perhaps the largest, working steam engine in the world. The four-story-high, 50-foot-long engine was used to pump water from the Thames for more than 100 years and is now the centerpiece of a large collection of working steam engines. And speaking of working, Martin's been doing too much of it and already needs another vacation.•

KEY FEATURES OF THE NEW DRIVER MODEL

The following list of features will give you some idea of the rationale behind the move away from a device driver architecture that's served the Macintosh operating system for more than a decade. Some of these features address problems of the old architecture, while some anticipate new requirements.

A simplified set of driver services independent of the Macintosh Toolbox
The existing Device Manager design is closely tied to specific features of the Macintosh Toolbox. The new system software release supports only a small set of driver services, which are independent of the Toolbox and are limited to just those things that drivers need to do; they don't let drivers display dialogs, open files, read resources, or draw on the screen. This greatly simplifies both the driver's task (the driver interacts only with the actual hardware) and the operating system's task (the OS needn't have a file system or screen available when starting up drivers).

Independence from the 68000 processor family
The old device driver architecture is highly dependent on specific features of the 680x0 processor architecture. For example, the way code segments are organized and the conventions for passing parameters depend on the 680x0 architecture and make the old driver code different from other code modules. This means that drivers can't be written in native PowerPC code — or must make use of computationally expensive mixed-mode switches.

Also, in the 680x0 architecture, critical sections and atomic operations use assembly-language sequences to disable interrupts. The PowerPC processor has a completely different interrupt structure, effectively making these techniques impossible to transport directly to native PowerPC code.

In the new system software, support for the driver model is independent of any particular processor, hiding processor-specific requirements in operating system libraries. Drivers can be compiled into native PowerPC code and can be written in a high-level language such as C. Because they're standard PowerPC code fragments, they aren't bound by the segment size limitations of the 680x0 architecture; they can be created with standard compilers and debugged with the Macintosh two-machine debugger.

A more flexible configuration facility
Driver configuration in the old architecture requires the ability to read resources from a parameter file, or from a 6-byte nonvolatile RAM area indexed by NuBus slot. These ad hoc configuration mechanisms based on the Resource Manager, File Manager, and Slot Manager are replaced in the new system software by a more flexible configuration facility that's used throughout the system.

Drivers use a systemwide name registry for configuration. Each device has an entry in the Name Registry containing properties pertinent to that device. Device drivers can also store and retrieve private properties. Device configuration programs (control panels and utility applications) should use the registry to set and retrieve device parameters.

System-independent device configuration
Devices can use Open Firmware to provide operating system configuration as well as system-independent bootstrap device drivers. Open Firmware is an architecture-independent IEEE standard for hardware devices based on the FORTH language. When the system is started up, it executes functions stored in each device's expansion ROM that provide parameters to the system. A device can also provide FORTH code to allow the system to execute I/O operations on the device. This means a card can be

used to bootstrap an operating system without having operating system–specific code in its expansion ROM.

Open Firmware and the bootstrap process are described in detail in IEEE document *1275 — 1994 Standard for Boot (Initialization, Configuration) Firmware.*•

Grouping by family

Drivers are grouped into general *families*, and family-specific libraries simplify their common tasks. Currently, four families are defined: video, communications, SCSI (through SCSI Manager 4.3), and NDRV (a catch-all for other devices, such as data acquisition hardware). The sample code is for a device driver in the NDRV family.

Direct support for important capabilities

The existing Device Manager doesn't directly support certain capabilities, such as concurrent I/O (required by network devices) and driver replacement. Driver writers who need these capabilities have had to implement them independently, which is difficult, error-prone, and often dependent on a particular operating system release. The new system software supports these capabilities in a consistent manner.

A choice of storage

Drivers can be stored in the hardware expansion ROM or in a file of type 'ndrv' in the Extensions folder. A later driver version stored in this folder can replace an earlier version stored in the hardware expansion ROM.

Forward compatibility

Device drivers written for the new system software will run without modification under Copland, the new generation of the Mac OS forthcoming from Apple, if they use only the restricted system programming interface and follow the addressing guidelines in *Designing PCI Cards and Drivers for Power Macintosh Computers*.

For more on Copland, see "Copland: The Mac OS Moves Into the Future" in this issue of *develop*.•

CONVERTING AN EXISTING DRIVER

To illustrate how you'd go about converting an existing device driver to drive a PCI device, let's suppose you've developed a document scanner with an optical character recognition (OCR) facility. The document scanner is currently controlled by a NuBus board that you designed, and you're building a PCI board to support the scanner on future Macintosh machines.

A useful way to approach the conversion effort is to conceptualize the device driver as consisting of three generally independent layers:

- A high-level component that connects the device driver to the operating system and processes requests.
- A mid-level component that has the device driver's task-specific intelligence. For example, this might contain OCR algorithms. This part is unique to each driver and generally hardware independent.
- The low-level bus interface "hardware abstraction layer" that directly manipulates the external device and thus is always device dependent.

At the same time, you might also organize the code in each of these three layers into the following functional groups:

- data transfer operations (Read, Write)
- interrupt service routines
- initialization and termination
- configuration and control (power management, parameterization)

Let's look at what you would do to each of these layers and groups.

First, you would throw out the high-level component in your driver that interacts with the Device Manager and replace it with the considerably simpler request processing of the new system software release. You would need to add support for the Initialize, Finalize, Superseded, and Replace commands (discussed later), as they have no direct counterpart in the existing Device Manager. You would also need to revise the way you complete an I/O request: instead of storing values in 68000 registers and jumping to jIODone, your driver would call IOCommandIsComplete.

The mid-level component in your driver would include scanner management and, in particular, OCR algorithms. These algorithms comprise the intelligence that sets your product apart from its competition. To convert your driver to a PCI device driver, you would recompile (or rewrite) the algorithms for the PowerPC processor. If the algorithms were in 68000 assembly language, you could get started by making mixed-mode calls between the new driver and the existing functions; however, this won't work with Copland, and I would recommend "going native" as soon as possible.

You would replace the low-level bus interface that manipulates registers on a NuBus card with code that manipulates PCI registers. Because this is specific to a particular hardware device, it won't be discussed in this article, but the sample driver on the CD shows you how to access PCI device registers.

You would also create Open Firmware boot code to allow your card to be recognized during system initialization. Because the new driver model doesn't use Macintosh Toolbox services, you would have to redesign your driver to (1) use the Name Registry for configuration instead of resources and parameter files, and (2) use the new timer services, replacing any dependency on the accRun PBControl call (the sample code shows how to call timer services, although it's not discussed here).

How your new driver code would look will become clearer in the next sections, where we examine key parts of the sample device driver. To get the whole picture, see the sample driver in its entirety on the CD.

The remainder of this article introduces a number of new operating system functions, as well as a few new libraries, managers, and such. "A Glossary of New Operating System Terms" will help you navigate through the new territory.

A LOOK AT THE SAMPLE DRIVER: CONFIGURATION AND CONTROL

Now we'll look at key pieces of the sample driver, starting with the code for configuration and control. As mentioned earlier, the sample driver is a member of the NDRV family. To the operating system, an NDRV driver is a PowerPC code fragment containing two exported symbols: TheDriverDescription and DoDriverIO. (Although all drivers have a TheDriverDescription structure, the particular driver family they belong to determines which other exported symbols are required.)

A GLOSSARY OF NEW OPERATING SYSTEM TERMS

CheckpointIO. A function that releases memory that had been configured by PrepareMemoryForIO.

DoDriverIO. A function provided by the driver that carries out all device driver tasks. When you build a driver, it must export this function to the Device Manager.

DriverDescription. An information block named TheDriverDescription that the Driver Loader Library uses to connect a device driver with its associated hardware. When you build a driver, it must export this block to the Driver Loader Library.

Driver Loader Library. A library of functions used by the Device Manager to locate and initialize all drivers. It uses the DriverDescription structure to match a driver with the hardware actually present on a machine.

Driver Services Library. A family-independent library of driver services limited to just those things that drivers need to do.

Expansion Bus Manager. A library that provides access to PCI configuration registers.

GetInterruptFunctions. A function that retrieves the current interrupt service functions established for this device.

GetLogicalPageSize. A function that retrieves the size of the physical page. Normally called once when the driver is initialized.

InstallInterruptFunctions. A function that replaces the current interrupt functions with functions specific to this device driver.

IOCommandIsComplete. A function that completes the current request by returning the final status to the caller, calling an I/O completion routine if provided, and starting the next transfer if necessary.

MemAllocatePhysicallyContiguous. A function that allocates a contiguous block of memory whose address can be passed, as a single unit, to a hardware device. This is essential for frame buffers and similar memory areas that must be accessed by both the CPU and an external device.

Name Registry. A database that organizes all system configuration information. Each device's entry in the registry contains a set of properties that can be accessed with RegistryPropertyGet and RegistryPropertyGetSize.

PoolAllocateResident. A function that allocates and optionally clears memory in the system's resident pool. This replaces NewPtrSys, which isn't available to forward-compatible PCI device drivers.

PoolDeallocate. A function that frees memory allocated by PoolAllocateResident.

PrepareMemoryForIO. A function that converts a logical address range to a set of physical addresses and configures as much as possible of the corresponding physical memory space for subsequent direct memory access.

QueueSecondaryInterrupt. A function that runs a secondary interrupt service routine at a noninterrupt level.

RegistryPropertyGet, RegistryPropertyGetSize. Functions that retrieve, respectively, the contents and the size of a property, given its name and a value that identifies the current Name Registry entity.

Software task. An independently scheduled software module that can call driver services, including PrepareMemoryForIO. Software tasks can be used to replace time-based processing that previously used the PBControl accRun service.

SynchronizeIO. A function that executes the processor I/O synchronize (**eieio**) instruction.

TheDriverDescription is a static structure, shown in Listing 1, that provides information to the operating system about the device that this driver controls. The driver will be loaded only if the device is present. TheDriverDescription also indicates whether the driver is controlled by a family interface (such as Open Transport for the communications family) and specifies the driver name to be used by operating system functions to refer to it. The Driver Loader extracts TheDriverDescription from the code fragment before the driver executes; thus it must be statically initialized.

Listing 1. TheDriverDescription

```
DriverDescription TheDriverDescription = {
   /* This section lets the Driver Loader identify the structure
      version. */
   kTheDescriptionSignature,
   kInitialDriverDescriptor,
   /* This section identifies the PCI hardware. It also ensures that the
      correct revision is loaded. */
   "\pMyPCIDevice",                    /* Hardware name */
   kMyPCIRevisionID, kMyVersionMinor,
   kMyVersionStage, kMyVersionRevision,
   /* These flags control when the driver is loaded and opened, and
      control Device Manager operation. They also name the driver to the
      operating system. */
   ( (1 * kDriverIsLoadedUponDiscovery)  /* Load at system startup */
   | (1 * kDriverIsOpenedUponLoad)       /* Open when loaded */
   | (0 * kDriverIsUnderExpertControl)   /* No special family expert */
   | (0 * kDriverIsConcurrent)           /* Driver isn't concurrent */
   | (0 * kDriverQueuesIOPB)             /* No internal IOPB queue */
   ),
   "\pMyDriverName",                   /* PBOpen name */
   0, 0, 0, 0, 0, 0, 0, 0,             /* For future use */
   /* This is a vector of operating system information, preceded by
      an element count (here, only one service is provided). */
   1,                                  /* Number of OS services */
   kServiceTypeNdrvDriver,             /* This is an NDRV driver */
   kNdrvTypeIsGeneric,                 /* Not a special type */
   kVersionMajor, kVersionMinor,       /* NumVersion information */
   kVersionStage, kVersionRevision
};
```

DoDriverIO is a single function called with five parameters to perform all driver services (see Table 1). The overall organization of the driver thus is very simple, as shown in Listing 2.

Table 1. DoDriverIO parameters

Parameter Type	Usage
addressSpaceID	Used for operating system memory management. Currently, only one address space is supported; future systems will support multiple address spaces.
ioCommandID	Uniquely identifies this I/O request. The driver passes it back to the operating system when the request completes.
ioCommandContents	Varies depending on the ioCommandCode value. For example, for Read, Write, Control, Status, and KillIO commands, it's a pointer to a ParamBlockRec.
ioCommandCode	Defines the type of I/O request.
ioCommandKind	Specifies whether the command is synchronous or asynchronous, and whether it's immediate.

Listing 2. DoDriverIO

```
OSErr DoDriverIO(AddressSpaceID       addressSpaceID,
                 IOCommandID          ioCommandID,
                 IOCommandContents    ioCommandContents,
                 IOCommandCode        ioCommandCode,
                 IOCommandKind        ioCommandKind)
{
   OSErr    status;

   switch (ioCommandCode) {
      case kInitializeCommand:
         status = DriverInitialize(ioCommandContents.initialInfo);
         break;
      case kFinalizeCommand:
         status = DriverFinalize(ioCommandContents.finalInfo);
         break;
      case kSupersededCommand:
         status = DriverSuperseded(ioCommandContents.supersededInfo);
         break;
      case kReplaceCommand:
         status = DriverReplace(ioCommandContents.replaceInfo);
         break;
      case kOpenCommand:
         status = DriverOpen(ioCommandContents.pb);
         break;
      case kCloseCommand:
         status = DriverClose(ioCommandContents.pb);
         break;
      case kReadCommand:
         status = DriverRead(addressSpaceID, ioCommandID, ioCommandKind, ioCommandContents.pb);
         break;
      case kWriteCommand:
         status = DriverWrite(addressSpaceID, ioCommandID, ioCommandKind, ioCommandContents.pb);
         break;
      case kControlCommand:
         status = DriverControl(addressSpaceID, ioCommandID, ioCommandKind,
                  (CntrlParam *) ioCommandContents.pb);
         break;
      case kStatusCommand:
         status = DriverStatus(addressSpaceID, ioCommandID, ioCommandKind,
                  (CntrlParam *) ioCommandContents.pb);
         break;
      case kKillIOCommand:
         status = DriverKillIO();
         break;
   }

   /* Force a valid result for immediate commands. Other commands return noErr if the operation
      completes asynchronously. */
   if ((ioCommandKind & kImmediateIOCommandKind) == 0) {
      if (status == kIOBusyStatus)    /* Our "in progress" value */
         status = noErr;              /* I/O will complete later */
```

(continued on next page)

Listing 2. DoDriverIO *(continued)*

```
      else
         /* To prevent a subtle race condition, the driver must not store final status in the
            caller's parameter block. This prevents a problem where the caller can reuse the
            parameter block before the caller's completion routine is called. */
         status = IOCommandIsComplete(ioCommandID, status);
   }
   return (status);
}
```

The driver must ensure that immediate operations (those that must complete without delay) return directly to the caller and that completed synchronous and asynchronous requests call IOCommandIsComplete. (The sample driver handler functions return the final status if they handled the request, and a private value, kIOBusyStatus, if an asynchronous interrupt will eventually complete the operation.)

In the sample driver, individual subroutines carry out the functions. I'll describe the administration routines first, then the process of carrying out an I/O operation.

INITIALIZATION AND TERMINATION

Currently, drivers perform all of their initialization when called with PBOpen and generally ignore PBClose. The new system software provides six commands for initialization and termination, as shown in Table 2. Since drivers are code fragments, they can also use the Code Fragment Manager initialization and termination routines, although this probably isn't necessary.

For details on the Code Fragment Manager, see *Inside Macintosh: PowerPC System Software.*•

Table 2. Driver commands for initializing and terminating

ioCommandCode Value	Usage
kInitializeCommand	Carries out normal initialization. Called once when the driver is first loaded.
kReplaceCommand	Indicates that this driver is replacing a currently loaded driver for the device (for example, a ROM driver is being replaced by a driver loaded from the system disk).
kOpenCommand	Begins servicing of device requests.
kCloseCommand	Stops servicing of device requests.
kSupersededCommand	Indicates that this driver will be replaced by another.
kFinalizeCommand	Shuts down the device and releases all resources. Called once just before the driver is to be unloaded.

When you look at the sample driver, you'll see that most of the work is done by Replace and Superseded, with Open and Close having no function there.

Here are the tasks that a driver needs to perform when initialized, whether by Initialize or Replace:

1. Initialize its global variables and fetch systemwide parameters, such as the memory management page size.
2. Fetch the device's physical address range (either memory address or PCI I/O addresses) from the Name Registry.
3. Enable memory or I/O access and use the DeviceProbe function to verify that the device is properly installed.
4. Fetch the interrupt property information from the Name Registry and initialize the interrupt service routine.
5. If all initializations complete correctly, use device-specific operations to reset the hardware.

Listing 3 shows how to extract the physical addresses of your device and use the "AAPL,address" property to get the corresponding logical addresses. Unlike address space assignments on NuBus machines, where the slot number directly corresponds to the device's 32-bit address range, PCI address space assignments are dynamic. Devices define a set of registers, and the system initialization process (Open Firmware) uses this information, together with information about buses and PCI bridges, to bind the device to its 32-bit physical address range. (Actually, although addresses use 32 bits, the low 23 bits select the physical address, while the high 9 bits select between main memory and PCI bus address spaces. The device driver uses the logical address to reference device registers.) Open Firmware code updates the Name Registry to show the device's binding. Note that the driver must search for the required address register and can't rely on any particular address being in a specific location within the property.

Listing 3. Fetching the device's logical address range

```
typedef struct AssignedAddress {
      UInt32      cellHi;     /* Address type */
      UInt32      cellMid;
      UInt32      cellLow;
      UInt32      sizeHi;
      UInt32      sizeLow;
} AssignedAddress, *AssignedAddressPtr;

#define kAssignedAddressProperty "assigned-addresses"
#define kAAPLAddressProperty     "AAPL,address"
#define kIOMemSelectorMask       0x03000000
#define kIOSpaceSelector         0x01000000
#define kMemSpaceSelector        0x02000000
#define kDeviceRegisterMask      0x000000FF

OSErr GetDeviceAddress(UInt32            selector,
                       UInt32            deviceRegister,
                       LogicalAddress    *logicalAddress)
{
   OSErr                  status;
   RegPropertyValueSize   size;
   AssignedAddressPtr     addressPtr;
   LogicalAddress         *logicalAddressVector;
```

(continued on next page)

Listing 3. Fetching the device's logical address range *(continued)*

```
   int                     nAddresses, i;
   UInt32                  cellHi;

   addressPtr = NULL;
   logicalAddressVector = NULL;
   status = GetThisProperty(kAssignedAddressProperty,
         (RegPropertyValue *) &addressPtr, &size);  /* See Listing 6. */
   if (status == noErr) {
      /* GetThisProperty returned a vector of assigned-address records.
         Search the vector for the desired address type. */
      status = paramErr;           /* Presume "no such address." */
      nAddresses = size / sizeof (AssignedAddress);
      for (i = 0; i < nAddresses; i++) {
         cellHi = addressPtr[i].cellHi;
         if ((cellHi & kIOMemSelectorMask) == selector
          && (cellHi & kDeviceRegisterMask) == deviceRegister) {
            if (addressPtr[i].sizeLow == 0)
               /* Open Firmware was unable to assign an address to this
                  memory area. We must return an error to prevent the
                  driver from starting up (status is still paramErr). */
               break;
            /* This is the desired address space. Find the corresponding
               LogicalAddress by resolving the "AAPL,address" property.
               We want the i'th LogicalAddress in the vector. */
            status = GetThisProperty(kAAPLAddressProperty,
               (LogicalAddress *) &logicalAddressVector, &size);
            if (status == noErr) {
               nAddresses = size / sizeof (LogicalAddress);
               if (i < nAddresses)
                  *logicalAddress = logicalAddressVector[i];
               else status = paramErr;
            }
            break;   /* Exit the for loop. */
         } /* Check for the requested register. */
      } /* Loop over all address spaces. */
      DisposeThisProperty((RegPropertyValue *) &addressPtr);
      DisposeThisProperty((RegPropertyValue *) &logicalAddressVector);
   } /* If we found our "assigned-addresses" property */
   return (status);
}
```

When the driver reads the "assigned-addresses" property, it looks at the address type (I/O or memory) and may also need to examine other information to make sure the address range is appropriate. For example, a device may have two memory address ranges — one for the device's registers and a separate range for its on-card firmware. The GetDeviceAddress function in Listing 3 uses the register number to determine which of several address ranges to use, but this may not work for all hardware. This function also resolves the logical address range that corresponds to the device's physical address range using an Apple-specific property that records device logical addresses. This is important for devices that require I/O cycles: using the logical address lets the driver treat these devices as if they used normal memory addresses, eliminating the overhead of the Expansion Bus Manager routines.

Listing 4 shows how a driver might use the Expansion Bus Manager to enable a device to become bus-master and respond to either memory or I/O accesses. It also shows how to read a device register with the DeviceProbe function. While the actual values are specific to the NCR 53C825 chip, the technique is generally useful. Note that the command word was changed using a read-modify-write sequence.

Listing 4. Checking for the correct hardware device

```
OSErr InitializeMyHardware(void)
{
   OSErr             status;
   UInt8             ctest3;
   UInt16            commandWord;

   status = ExpMgrConfigReadWord(
               &gDeviceEntry,            /* kInitializeCommand param */
               (LogicalAddress) 0x04,    /* Command register */
               &commandWord);            /* Current chip values */
   if (status == noErr)
      status = ExpMgrConfigWriteWord(
               &gDeviceEntry,            /* kInitializeCommand param */
               (LogicalAddress) 0x04,    /* Command register */
               commandWord | 0x0147);    /* New chip values */
   if (status == noErr)
      status = DeviceProbe(
               gDeviceBaseAddress + 0x9B,/* Chip Test 3 register */
               &ctest3,                  /* Store value here */
               k8BitAccess);
   if (status == noErr && (ctest3 & 0xF0) != 0x20)
      status = paramErr;                 /* Wrong chip revision */
   return (status);
}
```

The code for initializing the interrupt service routine, including connecting the primary interrupt service routine to the operating system, is shown in Listing 5. This code installs a single interrupt handler; if your device supports multiple interrupts (for example, if it supports several serial lines), you may want to use the new interrupt management routines in the Driver Services Library to build a hierarchy of interrupt service routines.

Listing 5. Initializing the interrupt service routine

```
#define kInterruptSetProperty   "driver-ist"

OSErr InitializeInterruptServiceRoutine(void)
{
   OSErr                  status;
   OSStatus               osStatus;
   RegPropertyValueSize   size;
```

(continued on next page)

Listing 5. Initializing the interrupt service routine *(continued)*

```
   InterruptSetMember      *interruptSetMember;

   status = GetThisProperty(kInterruptSetProperty,
        (RegPropertyValue *) &interruptSetMember, &size);
   if (status == noErr) {
      if (size < (sizeof (InterruptSetMember)) {
         DisposeThisProperty((RegPropertyValue *) &interruptSetMember);
         status = paramErr;
      }
   }
   if (status == noErr) {
      /* We have the interrupt set ID and member number. Save the
         current interrupt set and get the current functions for this
         interrupt set. */
      gInterruptSetMember = *interruptSetMember;  /* Save globally */
      DisposeThisProperty((RegPropertyValue *) &interruptSetMember);
      osStatus = GetInterruptFunctions(gInterruptSetMember.setID,
            gInterruptSetMember.member, &gOldInterruptSetRefCon,
            &gOldInterruptServiceFunction, &gOldInterruptEnableFunction,
            &gOldInterruptDisableFunction);
      if (osStatus != noErr)
         status = paramErr;
   }
   if (status == noErr) {
      /* We have the information we need. Install our own interrupt
         handler function. If successful, call the old enabler to
         enable interrupts (we don't install a private enabler). */
      osStatus = InstallInterruptFunctions(
               gInterruptSetMember.setID,
               gInterruptSetMember.member,
               NULL,                  /* No refCon */
               DriverInterruptServiceRoutine,  /* See Listing 11. */
               NULL,                  /* No new enable function */
               NULL);                 /* No new disable function */
      if (osStatus != noErr)
         status = paramErr;
   }
   if (status == noErr)
      (*gOldInterruptEnableFunction)(gInterruptSetMember,
               gOldInterruptSetRefCon);
   return (status);
}
```

Interrupt management routines are described in Chapter 9 of *Designing PCI Cards and Drivers for Power Macintosh Computers*.•

GetThisProperty (Listing 6) is a generic utility function that retrieves a property from the Name Registry, storing its contents in the system's resident memory pool. This is useful for retrieving configuration information. The driver must, of course, return the memory to the pool when it's no longer needed, using DisposeThisProperty, also shown in Listing 6.

Listing 6. Retrieving properties from the Name Registry

```
OSErr GetThisProperty(RegPropertyNamePtr    regPropertyName,
                      RegPropertyValue      *resultPropertyValuePtr,
                      RegPropertyValueSize  *resultPropertySizePtr)
{
   OSErr                  status,
   RegPropertyValueSize   size;

   *resultPropertyValuePtr = NULL;
   status = RegistryPropertyGetSize(
            &gDeviceEntry,            /* kInitializeCommand param */
            regPropertyName,
            &size);
   if (status == noErr) {
      *resultPropertyValuePtr =
               (RegPropertyValue *) PoolAllocateResident(size, FALSE);
      if (*resultPropertyValuePtr == NULL)
         status = memFullErr;
   }
   if (status == noErr)
      status = RegistryPropertyGet(
               &gDeviceEntry,         /* kInitializeCommand param */
               regPropertyName,
               *regPropertyValuePtr,
               &size);
      if (status != noErr)
         DisposeThisProperty(regPropertyValuePtr);
   }
   if (status == noErr)
      *resultPropertySizePtr = size;  /* Success! */
   return (status);
}

/* DisposeThisProperty disposes of a property that was obtained by
   calling GetThisProperty. Note that applications would call DisposePtr
   instead of PoolDeallocate. */
void DisposeThisProperty(RegPropertyValue *regPropertyValuePtr)
{
   if (*regPropertyValuePtr != NULL) {
      PoolDeallocate(*regPropertyValuePtr);
      *regPropertyValuePtr = NULL;
   }
}
```

Applications can use the functions in Listing 6 but must replace calls to PoolAllocateResident and PoolDeallocate with calls to NewPtr and DisposePtr. The latter aren't available to PCI device drivers.•

CARRYING OUT AN I/O OPERATION

There are two parts to starting an asynchronous I/O operation: the driver must carry out the operations unique to the particular hardware device and it must configure memory so that hardware direct memory access (DMA) operations can

take place. Completing an operation requires responding to hardware interrupts, updating user parameter block fields, selecting the proper status code, and calling IOCommandIsComplete to inform the Device Manager that the driver has finished with this I/O request. The sequence for a complete, but somewhat simplified, I/O transaction might be as follows:

1. Use parameter block information to configure device-specific information.
2. Compute the logical addresses that are needed and call PrepareMemoryForIO to compute the corresponding physical addresses. PrepareMemoryForIO replaces the LockMemory and GetPhysical functions and handles virtual memory considerations.
3. With all memory ready for DMA, configure the hardware to start the transfer.
4. When the device completes its operation, it will interrupt the PowerPC processor. The operating system kernel will call your driver's primary interrupt service routine.
5. When the device request is complete, memory structures prepared by PrepareMemoryForIO for this operation are released with CheckpointIO, and the interrupt service routine calls IOCommandIsComplete to return final status to the caller.

This sequence represents an idealized and somewhat simplified situation. For example, display frame buffers generally don't interrupt when written to but might interrupt at the end of a display cycle.

I won't say much about the Read, Write, Control, Status, and KillIO handlers: they carry out tasks that are specific to the particular driver. Often, they initiate an operation that will be completed by a device hardware interrupt. Control and Status handlers must process PBControl csCode = 43 (driverGestalt) requests. These provide a systematic way to query device capabilities and are also used for power management. KillIO replaces the PBControl csCode = 1 (killCode) used for desk accessories; it stops all pending I/O requests.

Before jumping into the complexities of PrepareMemoryForIO and interrupt service, I need to mention one small task: setting and reading values in the device registers.

SETTING AND READING DEVICE REGISTER VALUES

The PCI bus architecture gives hardware developers two methods for setting and reading values in the device registers: memory-mapped I/O and I/O cycle operations (described in more detail in "Methods of I/O Organization"). A device advertises its I/O organization through bits in its configuration register and by providing a PCI-standardized "reg" property. When the system starts up, it assigns each device a range of physical addresses in the system's 32-bit physical address space. The driver can retrieve the device's physical addresses by resolving the "assigned-addresses" property and can use the Apple-specific "AAPL,address" property to translate the values in an "assigned-addresses" property to logical addresses, as was shown in Listing 3. Your driver should use these values when accessing your device's registers. Ranges of logical addresses are assigned to PCI bus memory and I/O cycles; thus, your driver can perform I/O cycles without calling operating system functions.

For example, the sample driver's hardware device has a test register (byte) at offset 0xCC from the start of its memory base address. Suppose the logical address retrieved by GetDeviceAddress was stored in the global gDeviceBaseAddress, defined as

METHODS OF I/O ORGANIZATION

Memory-mapped I/O and I/O cycle operations represent two ways of designing a computer architecture.

Using memory-mapped I/O, device hardware responds to normal memory operations in a particular range of addresses. For example, PDP-11 computers without memory management hardware reserved 8K for peripheral hardware registers, limiting the memory available to programs to 56K.

I/O cycle operations effectively place external devices in an independent address space. This gives programs additional memory but requires special instructions to access peripheral devices. The Intel 80x86 series uses this organization.

To the programmer, memory-mapped I/O has the advantage of allowing direct device operations without special instructions, making it relatively easy to write device drivers in high-level languages. As bus widths and memory size limitations have eased, the inability to use part of the address space for programs has become less of an issue.

Apple's PCI-based machines use only memory-mapped I/O. However, the bus interface hardware generates PCI I/O cycles for a subset of the physical address space.

```
volatile UInt8    *gDeviceBaseAddress;
```

The driver could then read the test register with

```
testRegister = gDeviceBaseAddress[0xCC];
```

The **volatile** keyword is important, as it prevents the compiler from removing what appear to be unnecessary operations. Drivers will also need to call the SynchronizeIO function in the Driver Services Library to force the PowerPC processor to flush its data pipeline. While the sample device driver appears to use only memory operations, the PCI hardware issues either memory or I/O addresses depending on the particular logical address reference. To issue I/O addresses, your device driver would have to retrieve the "AAPL,address" property shown in Listing 3.

While byte accesses are straightforward, word (16-bit) and long word (32-bit) accesses are more complex. This is because the PCI bus is little-endian (the address of a multibyte entity is the address of the low-order byte), whereas the Mac OS and the PowerPC chip are big-endian (the address of a multibyte entity is the address of the high-order byte). To access 16-bit and 32-bit data, then, your driver must swap bytes in memory, either by using the PowerPC **lwbrx** instruction or by calling the library functions EndianSwap16Bit or EndianSwap32Bit. The Expansion Bus Manager routines handle "endian swapping" internally. *Failing to swap bytes was the most frequent error when I wrote the sample driver; you would be wise to check this thoroughly in your code.*

PREPARING THE MEMORY

Before starting a DMA operation, the operating system must ensure that the data accessed by the operation is in physical memory and that any data in the processor cache has been written to memory. This is done with the PrepareMemoryForIO and CheckpointIO routines. Because the process is complex, I'll break it down into smaller pieces to describe it. Let's assume your driver will prepare two areas: a permanent shared-memory area used to communicate with the device (this could be used for a display frame buffer) and a request-specific area used for a single I/O request.

Preparing the shared area is fairly straightforward: your driver allocates a physical mapping table, initializes an IOPreparationTable, and calls PrepareMemoryForIO. Listing 7 shows how to prepare a shared area and Listing 8 shows several related

Listing 7. Preparing a shared memory area

```
IOPreparationTable   gSharedIOTable;
LogicalAddress       gSharedAreaPtr;

OSErr PrepareSharedArea(
      AddressSpaceID    addressSpaceID)      /* DoDriverIO parameter */
{
   OSErr             status;
   ItemCount         mapEntriesNeeded;

   gSharedAreaPtr =
               MemAllocatePhysicallyContiguous(kSharedAreaSize, TRUE);
   if (gSharedAreaPtr == NULL)
      return (memFullErr);
   gSharedIOTable.options =
      ( kIOIsInput                 /* Device writes to memory. */
      | kIOIsOutput                /* Device reads from memory. */
      | kIOLogicalRanges           /* Input is logical addresses. */
      | kIOShareMappingTables );   /* Share tables with kernel. */
   gSharedIOTable.addressSpace = addressSpaceID;
   gSharedIOTable.firstPrepared = 0;
   gSharedIOTable.logicalMapping = NULL; /* We don't want this. */
   /* Describe the area we're preparing and allocate a mapping table. */
   gSharedIOTable.rangeInfo.range.base = gSharedAreaPtr;
   gSharedIOTable.rangeInfo.range.length = kSharedAreaSize;
   mapEntriesNeeded = GetMapEntryCount(gSharedArea, kSharedAreaSize);
   gSharedIOTable.physicalMapping = PoolAllocateResident(
         (mapEntriesNeeded * sizeof (PhysicalAddress)), TRUE);
   if (gSharedIOTable.physicalMapping == NULL)
      status = memFullErr;
   else
      status = PrepareMemoryForIO(&gSharedIOTable);
   if (status == noErr)
      status = CheckPhysicalMapping(&gSharedIOTable, kSharedAreaSize);
   return (status);
}
```

Listing 8. PrepareMemoryForIO utilities

```
/* Return the number of PhysicalMappingTable entries that will be
   needed to describe this memory area. */
ItemCount GetMapEntryCount(void          *areaAddress,
                           ByteCount     areaLength)
{
   ByteCount         normalizedLength;
   UInt32            theArea;

   theArea = (UInt32) areaAddress;
   normalizedLength = PageBaseAddress(theArea + areaLength - 1)
                    - PageBaseAddress(theArea);
   return (normalizedLength / GetLogicalPageSize());
}
```

(continued on next page)

Listing 8. PrepareMemoryForIO utilities *(continued)*

```
/* Check that the entire area was prepared and that all physical
   memory is contiguous. */
OSErr CheckPhysicalMapping(IOPreparationTable  *ioTable,
                           ByteCount           areaLength)
{
   ItemCount        i;
   OSErr            status;

   if (areaLength != ioTable->lengthPrepared)
      status = paramErr;        /* Didn't prepare the entire area. */
   else {
      status = noErr;
      for (i = 0; i < ioTable->mappingEntryCount - 1; i++) {
         if (NextPageBaseAddress(ioTable->physicalMapping[i])
          != ioTable->physicalMapping[i + 1]) {
            status = paramErr;   /* Area isn't physically contiguous. */
            break;
         }
      }
   }
   return (status);
}

/* Return the start of the physical page that follows the page
   containing this physical address. */
PhysicalAddress NextPageBaseAddress(PhysicalAddress theAddress)
{
   UInt32           result;

   result =
         PageBaseAddress(((UInt32) theAddress) + GetLogicalPageSize());
   return ((PhysicalAddress) result);
}

/* Return the start of the physical page containing this address. */
UInt32 PageBaseAddress(UInt32 theAddress)
{
   return (theAddress & ~(GetLogicalPageSize() - 1));
}
```

utility routines. Because PrepareSharedArea allocates memory for its physical mapping table, it must be called when your driver is initialized. Note that GetLogicalPageSize, used in several routines, returns a systemwide constant value; a production device driver would call it once, storing the value in a global variable.

To prepare a request-specific user area, your driver will initialize an IOPreparationTable with the procedure shown in Listing 9. Since your driver can be called from an I/O completion routine, it can't allocate a physical mapping table for each I/O request. Instead, your initialization procedure will allocate a maximum-length mapping table.

To process an I/O request, the driver initializes the options and I/O range and then calls PrepareMemoryForIO and, after I/O completion, CheckpointIO. How to

Listing 9. Initializing a request-specific IOPreparationTable

```
IOPreparationTable   gRequestIOTable;
ItemCount            gRequestMapEntries;

OSErr InitializeRequestIOTable(void)
{
   OSErr             status;
   ByteCount         mapTableSize;

   /* Compute the worst-case number of map entries. */
   gRequestMapEntries =
            GetMapEntryCount((void *) GetLogicalPageSize() - 1,
            kDriverMaxTransferLength);
   mapTableSize = (gRequestMapEntries * sizeof (PhysicalAddress));
   gRequestIOTable.physicalMapping =
            PoolAllocateResident(mapTableSize, TRUE);
   status = (gRequestIOTable.physicalMapping != NULL)
            ? noErr : memFullErr;
   return (status);
}
```

prepare a single request is shown in Listing 10. You call CheckpointIO to complete your use of the buffer in the interrupt service routine, as shown later in Listing 11.

A production device driver must extend the algorithm in Listing 10 to handle two more complex cases:

- Virtual memory is enabled. This being the normal case, the user area isn't necessarily physically contiguous. If your hardware can handle this, you can postprocess the physical mapping table into a scatter-gather table.
- The operating system has only a limited amount of permanently resident memory. Even if your hardware can perform a single 500 MB I/O transfer, you won't want to allocate that many physical mapping tables; you wouldn't get a significant performance gain and you would make your driver unusable on smaller configurations.

The solution to both of these problems is partial preparation. Your driver provides a physical mapping table of reasonable size. PrepareMemoryForIO prepares as much as possible and your driver uses the firstPrepared and lengthPrepared fields to navigate the physical mapping table. When your driver has performed all I/O in a partial preparation, it recalls PrepareMemoryForIO to prepare the next segment. So the overall, somewhat simplified, algorithm is as follows:

1. Prepare the first area.
2. Build scatter-gather tables and start up the device. When the device interrupts, continue with the next step.
3. When the device needs more data, have the interrupt service routine check the state field in the IOPreparationTable. If the I/O is incomplete, send a software interrupt to the driver's "restart I/O" task.

Listing 10. Using the request-specific IOPreparationTable

```
OSErr PrepareIORequest(AddressSpaceID    addressSpaceID,
                       LogicalAddress    userBufferPtr,
                       ByteCount         userCount)
{
   OSErr            status;
   ItemCount        mapEntriesNeeded;

   gRequestIOTable.options =
      ( kIOIsInput                 /* Device writes to memory. */
      | kIOLogicalRanges           /* Input is logical addresses. */
      | kIOShareMappingTables );   /* Share tables with kernel. */
   gRequestIOTable.addressSpace = addressSpaceID;
   gRequestIOTable.firstPrepared = 0;
   gRequestIOTable.logicalMapping = NULL;   /* We don't want this. */
   /* Store the user parameters in the IOPreparationTable. */
   gRequestIOTable.rangeInfo.range.base = userBufferPtr;
   gSharedIOTable.rangeInfo.range.length = userCount;
   mapEntriesNeeded = GetMapEntryCount(userBufferPtr, userCount);
   if (mapEntriesNeeded > gRequestMapEntries)
      status = paramErr;
   else {
      gRequestIOTable.mappingEntryCount = mapEntriesNeeded;
      status = PrepareMemoryForIO(&gRequestIOTable);
   }
   if (status == noErr)
      status = CheckPhysicalMapping(&gRequestIOTable, userCount);
   return (status);
}
```

4. Have the "restart I/O" task call PrepareMemoryForIO to prepare the next area (this can cause virtual memory paging). If successful, continue with step 2 to restart the device.
5. When I/O completes, call CheckpointIO to release the kernel resources reserved by PrepareMemoryForIO.

THE INTERRUPT SERVICE ROUTINE

When the hardware device completes a request, it interrupts the PowerPC processor. The operating system kernel fields the interrupt and searches an interrupt service tree to find a function that's been registered to handle that interrupt. A driver has established this function by calling InstallInterruptFunctions, as was shown in Listing 5.

A driver's interrupt service routine is generally broken into two parts: a primary routine that handles immediate operations and a secondary routine that completes the operation, releases any system resources held by PrepareMemoryForIO, and calls IOCommandIsComplete. (Note that some drivers will have no secondary routine.)

Secondary interrupt routines are serialized: they always run to completion before the system calls them again. However, they don't block other devices from interrupting the system. This greatly simplifies device driver design, as the secondary interrupt routine can manage the driver's internal queues without the significant overhead that blocking all processor interrupts would require.

Device drivers may need more complex processing than can be accomplished with primary and secondary interrupt routines. For example, a CD-ROM driver needs to check for disk insertion periodically. Also, all drivers need to handle virtual memory paging. To accomplish this, a driver can create a software task — an independent function that's scheduled at a time when all system services are available. Interrupt service and timer completion routines can schedule software tasks when necessary.

Listing 11 shows an extremely simplified interrupt service routine to familiarize you with this organization. DriverInterruptServiceRoutine, the primary routine, stores the hardware completion status and then queues a secondary interrupt routine to complete the operation. The secondary interrupt routine completes the I/O request by checkpointing the memory that was prepared before the transfer started. It then passes final completion status back to the operating system kernel.

This sample doesn't use the interrupt set member number, the refCon, or the interrupt count, which are needed for interrupt service routines that handle several devices (for example, in the case of a hardware device that controls several serial lines). Also, to simplify this sample, I'm presuming that all information is stored in driver globals. A better organization would make use of a "per-request" data structure that encapsulates all information needed for a single user I/O request (such as PBRead); this greatly simplifies the driver organization when you want to extend the driver to support multiple simultaneous requests (concurrent I/O).

Listing 11. A simplified interrupt service routine

```
InterruptSetMember DriverInterruptServiceRoutine(
      InterruptSetMember   interruptSetMember,     /* Unused here */
      void                 *refCon,                /* Unused here */
      UInt32               theInterruptCount)      /* Unused here */
{
   OSErr            status;
   UInt8            driverStatus;

   /* Retrieve the operation status from the device. This is fiction:
      a real device will be much more complex. */
   driverStatus = gDeviceBaseAddress[kDeviceStatusRegister];
   if (driverStatus == <device is not interrupting>
      return (kISRIsNotComplete);
   if (driverStatus == kDeviceStatusOK)
      status = noErr;
   else
      status = ioErr;
   /* The operation is (presumably) complete. Queue a secondary interrupt
      task that will release all memory and return the final status to
      the caller. We'll ignore an error from QueueSecondaryInterrupt. */
   (void) QueueSecondaryInterrupt(
            DriverSecondaryInterruptRoutine,
            NULL,                        /* No exception handler */
            (void *) status,             /* Operation ioResult */
            NULL);                       /* No p2 parameter */
   return (kISRIsComplete);
}
```

(continued on next page)

Listing 11. A simplified interrupt service routine *(continued)*

```
OSStatus DriverSecondaryInterruptRoutine(
      void     *p1,     /* Has ioResult value */
      void     *p2)     /* Unused */
{
   IOPreparationID   ioPreparationID;  /* Request I/O prep ID */

   /* Copy operation-specific values (such as the number of bytes
      transferred) into the caller's parameter block. */
   gCurrentParmBlkPtr->ioActCount = <device-specific value>;
   ioPreparationID = gRequestIOTable.preparationID;
   if (ioPreparationID != kInvalidID) {
      gRequestIOTable.preparationID = kInvalidID;
      (void) CheckpointIO(ioPreparationID, kNilOptions);
   }
   /* IOCommandIsComplete is the only function that should set the
      ioResult field. */
   IOCommandIsComplete(gIOCommandID, (OSErr) p1);
   return (noErr);
}
```

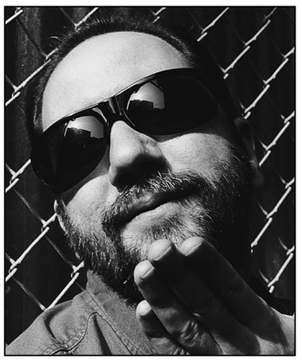
JUST THE TIP OF THE ICEBERG

There's a lot of material here — and a lot more that I haven't discussed. Still, this should give you a good overview of the new driver services and how they work together. While this may be overwhelming if you've never written a device driver before, those of you who have (for any operating system) will be happy to note how much isn't here: no assembly language, no dependencies on the strange quirks of the Mac OS, and all hardware dependencies either hidden from you or limited to your device's specific needs.

REFERENCES

- *Designing PCI Cards and Drivers for Power Macintosh Computers* will be available from APDA in mid-June.
- IEEE document *1275 — 1994 Standard for Boot (Initialization, Configuration) Firmware* (Part number DS02683, available from IEEE Standards Department, P.O. Box 1331, Piscataway, NJ 08855).
- *Inside Macintosh: Power PC System Software* (Addison-Wesley, 1994), Chapter 3, "Code Fragment Manager."

Thanks to our technical reviewers Jano Banks, Holly Knight, Wayne Meretsky, Tom Saulpaugh, and George Towner.•



MPW TIPS AND TRICKS

Building a Better (Development) Environment

TIM MARONEY

The days of the solitary hacker are long past. While this reclusive species is still spotted in the wildernesses of academia and shareware, today's commercial engineers roam the virtual plains in herds, overcoming the incessant problems of bloated software projects by sheer force of numbers.

Like all human groups, software teams are tied together by a shared language and environment. On the Macintosh, this common ground often contains a set of MPW scripts and tools. While most developers prefer the faster compilers offered by third-party vendors, the scripting and source control capabilities of MPW make it an indispensable workhorse in team software projects. It even serves as the cornerstone of many cross-platform efforts involving both the Mac OS and that other operating system.

Following a few simple principles will greatly reduce headaches resulting from maintaining a team's MPW configuration. These guidelines may seem obvious, but I have yet to see a project that followed all of them.

ENGINEERS ARE USERS, TOO

While we may be accustomed these days to thinking of a user interface as a sequence of pictures, a build environment in MPW is as much a user interface as any other software system. Like all such projects, designing it naively invites the wrath of your users — in this case, the engineers on your team. And unlike most users, they have your direct telephone number and know where you park your car!

The primary principle of user interface design is to stop thinking "I want to make the best X ever," whether X is a text engine, file system, image processor, build environment, or gorgonzola sandwich. That narrow form of thinking leads to excellent solutions to technical problems but systems that are difficult to use, because the model of the problem adopted by an engineer is likely to be different from that applied by an end user. For instance, to an image processing expert, rotating an image is a problem of accurate and rapid approximation across a grid, but to a scanner operator, the problem is one of deciding when to rotate, whether to do it automatically, whether to do it before or after other operations, and so forth. A technically superb rotation algorithm may completely fail to meet the requirements of the operator in a print shop if it wasn't originally designed with that environment in mind.

Balance technical problem solving by thinking through in detail how the system will be used to accomplish specific tasks. Spell out particular scenarios and make sure your solutions work in them. Otherwise, they probably won't.

So, to keep the needs of your various users in mind, you need to consider not only a normal build, but auxiliary tasks such as the following:

- installing and updating the system
- incorporating scripts from other sources
- giving MPW commands by hand
- personalizing the configuration
- maintaining a synchronized environment among all users
- archiving the environment for reproducing builds
- working from home over Apple Remote Access, and other ways of working remotely
- troubleshooting scripts and tools

Never assume that smart people make fewer errors. A rule of thumb is that the number of errors made is proportional to the number of possible errors, not to the skill of the user. Error prevention should be one of your guiding principles in any system design. For instance, don't require three commands in a particular order to complete a build; a single build should be a single command. If you have user interface design staff, get them involved with the development environment; their familiarity with principles such as error prevention and nonmodality could be very helpful.

TIM MARONEY has been tempered in the forge of computer networks, acquiring a rough, cast-iron finish that's often mistaken for obnoxiousness. His favorite animal name is "Kittens," his favorite food is anything dead, and his favorite new game involves building globe-spanning conspiracies out of overpriced trading cards. Tim supplements his seven-figure earnings from writing for magazines by developing software for Apple.•

Most of all, talk to your users. Ask them what they need and what their problems are. Sometimes their suggestions will be ones you can use directly; more often they won't hold up to scrutiny as actual designs, but they always indicate a legitimate area of concern that you'll need to address. Design your system on paper first, and have your users review your drafts. This time will pay off later in increased productivity.

Many of the principles of modern software design were originally developed for traditional command-line systems. See *The Elements of Friendly Software Design: The New Edition* by Paul Heckel (Alameda: Sybex, 1991).•

CHECK IN THE SYSTEM

An obvious, but flawed, approach to organizing a system of tools and scripts is to put them all on a server where everyone can reach them. Each engineer is responsible for synchronizing his or her local configuration with the latest files on the server, and anyone can improve the scripts in their copious free time. In addition, everyone can customize their own system as much as they like.

In practice this simple scheme wastes the time of everyone on the team, because no one ever has the same configuration as anyone else. A typical frustrating conversation under this system would be:

I can't build the SuperWidgets library. Does it build for you?

Sure! Maybe you didn't get the new SourceGrinder script?

No, I got that yesterday, after I couldn't build Pat's latest brilliant changes to WhizzySnork. Let's take a look at your copy of the MungePrefix tool.

Hmmm. It seems to match yours. Gee, I don't know what the problem could be. Let's both do a complete reinstall and try again.

(Repeat until hysteria ensues.)

The solution to the problem of synchronization is to keep the build system itself under source control. When people run into problems, they'll make sure that they've checked out the current scripts and tools as well as the current source files, *before* they bother you about it. If they don't, they'll look silly. Since that will probably bring back unpleasant memories from the playground, they'll try harder next time.

For complex projects, you'll probably want to institute a regular build process with versioning and source archiving. When you archive the sources for a build, don't forget to archive a matching revision of the development environment! You may need to reproduce that build in the future, for which you'll need the source code and the exact build system.

In some larger projects, the development environment may be maintained by a group separate from the programmers who use the system. In that case, it may not be practical to archive the environment as part of a project build. The environment group needs to archive the system with named versions, and the project team needs to always build with respect to a named version of the environment. The project team also needs to record in its release notes which version of the development environment was used for each archived build. This allows the build to be reproduced from the two archives.

HAVE AN INSTALLED COPY HANDY

Bootstrapping an MPW configuration for a new engineer can be painful. There is a chicken-and-egg problem with any script-based installation of an MPW build system. The scripts you want to use for installation are checked in, but how does the first-timer get to them? You can write out careful step-by-step instructions, but few engineers can resist the temptation to improvise. You'll wind up doing it for them after all when they fail.

Instead, keep an up-to-date copy of a preconfigured MPW on a convenient server. The new user simply copies the entire MPW folder from the server to the local disk (remember those licensing restrictions, though!), edits the configuration file, and is ready to run.

THE DREADED USERSTARTUP•PERSONAL FILE

It's perfectly clear to the development environment designer that the user needs to type his or her name where it says

```
Set MyName "Your Name Here"
```

but no one ever seems to fill in the blanks correctly without hand-holding.

It may be worth your while to write a mini-application that sets up the personal configuration file in the MPW folder. An hour or two creating a setup application with a nice, clear modeless dialog will pay for itself a few new hires down the road. More simply, you can use MPW's Request, GetFileName, and GetListItem commands in a setup script — but a single dialog is friendlier.

This application or script should also be stored on the server where you have the preconfigured MPW folder. With a little clever scripting, you can easily arrange for

the application to be run automagically if the personal configuration file hasn't yet been set up.

There are a few kinds of setup that can be done programmatically. For instance, if a script needs to know the monitor size, don't ask users to type it in themselves; an MPW tool can look at the graphics device list and figure it out by itself.

ESCHEW CLEVERNESS

One of the best programming tips I ever got was from an introductory LISP text I read a few centuries ago as an undergraduate. It warned against cleverness in coding. On the surface this would appear to be stupid advice. Isn't cleverness a requirement for programming? The problem is that when our own code strikes us as clever, it usually involves some trick or back door that's both fragile and hard to understand, not only for the next poor sap who inherits the code, but maybe for ourselves a month or three from now. Yet these clever tricks are rewarding. Not only does a trick resolve a sticky problem in one swell foop, but it reinforces our belief in our own intelligence and resourcefulness.

LISP, being inherently weird, lends itself to clever solutions. So does object-oriented programming. (I'll spare you the name of a program that buries its resolution of conflicting filenames — dialogs and all — deep in the bowels of a general-purpose string class.) Scripting languages such as those of MPW and **csh** also encourage cleverness.

Remember the scripts to accelerate launching in last issue's MPW Tips and Tricks? The form in which they were passed to me used a very clever method of signaling a cold boot: it aliased the built-in End command to Quit, bypassing the state-saving code in the Quit script. Needless to say, the potential for side effects was enormous, but no doubt someone enjoyed thinking of it! I changed the cold boot sequence to write an empty file called DontSaveState in the MPW folder, and the Quit script to detect and remove this file. It takes perhaps a tenth of a second longer, but it's comprehensible and free from harmful side effects.

KEEP IT SIMPLE, STUPID

Another common class of difficulties results from redesigning the basis of MPW. It can be tempting to make big changes to the system, such as by changing the default value of a built-in variable like Exit, or permanently blanking variables like CIncludes and RIncludes to prevent conflicts with local headers.

The problem is that this turns a multifunction system into a single-function system, making MPW useful solely for the build tasks you've planned. Scripts from other sources won't work anymore, and the existing techniques and skills of people on the team may become hard for them to apply in the oddly mutated environment. Getting rid of RIncludes might make some part of your build sequence easier to manage, but what happens when an engineer wants to DeRez something by hand?

The solution is to avoid changing the underpinnings of the MPW Shell. If you need to add variables, add them as new variables — don't mess with the old ones. Don't install patches that let you add whizzy graphical menus and floating windows if they interfere with the ordinary AddMenu and Open commands. When you do need to change something, change it only in the scope of the script where it's needed.

Among other reasons, you may someday need to have more than one build system installed. Suppose your company is acquired by the Gizmonics Institute and they have their own MPW configuration. Would you rather throw away yours and try to figure out how to shoehorn your source code into their system, or be able to run them both in the same MPW Shell? Or suppose (and I admit this is pretty unlikely) you start talking with the weirdos down the hall instead of just snickering about them behind their backs. Before you know it, you'll be drinking beer together and trying to integrate your build systems. Don't laugh; it happens.

THE JOY OF THEFT — SHARE AND ENJOY

There are various sources for useful MPW scripts. Instead of trying to do it all yourself, you can impress your manager by ripping off scripts from CDs, computer networks, friends, and so forth. Sometimes even magazines have good stuff.

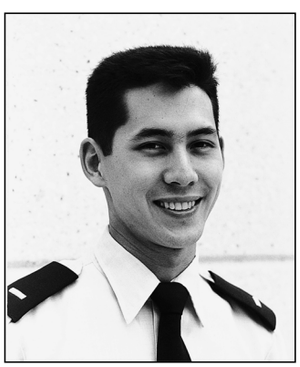
Apple already distributes quite a few useful scripts, such as those in the folder called DTS MPW Goodies on this issue's CD. Posting a note on a Usenet newsgroup may get you just the script you wanted in a matter of hours or days (even though you could have done it better yourself, of course).

Remember to share a little of your own work to balance the karmic load. This is the philosophy of UNIX®, and unfortunately it's better developed in that culture than in ours. Don't forget the others in the virtual herd!

Thanks to Shad Ahmad, Dave Evans, Arnaud Gourdol, and Eleanor the Wonder Gerbil for reviewing this column.•

Custom Color Search Procedures

Color QuickDraw can be customized for specific tasks in many ways, most commonly by replacing the "bottleneck" procedures at its heart. But another, often overlooked way of customizing Color QuickDraw is by writing and installing custom color search procedures. These procedures are very useful for color separation and other color processing tasks, and for modifying QuickDraw's default drawing behavior to solve particular problems. This article reviews some Color QuickDraw basics, explores how color search procedures work, and presents a sample search procedure.



JIM WINTERMYRE

It's 2 A.M., and you're finally ready to draw your carefully constructed offscreen GWorld to a window. The GWorld is 32 bits deep and has been set up to contain a color ramp using 100 shades of red. You've already created a palette containing the 100 shades of red you need and attached it to your window, so the exact colors will be available on your 256-color screen. You plunk in your call to CopyBits, recompile, and . . . Ack! Instead of the expected smooth red ramp, you get an image with 16 distinct bands of color (see Figure 1 on the inside back cover of this issue).

What happened? How can you get the results you want? This article attempts to answer both of these questions, and a few others along the way. What happened has to do with the way Color QuickDraw converts colors to pixel values, so we'll start with a brief review of how this works. As for getting the results you want, one way is to use a custom color search procedure, which is the main subject of this article.

A QUICK REVIEW OF COLOR IN QUICKDRAW

Before delving into custom color search procedures, let's pause for a quick review of how QuickDraw converts between colors and pixel values. If you're already familiar with this, feel free to skip ahead to the section "Drawbacks of Inverse Tables."

How QuickDraw converts colors to pixel values and vice versa is discussed in *Inside Macintosh: Imaging With QuickDraw*, and in the Color Manager chapter of *Inside Macintosh: Advanced Color Imaging* (available on this issue's CD in draft form). Only a brief overview of this complex topic is provided here.•

JIM WINTERMYRE (Internet winter@ai.rl.af.mil) is in the Air Force but doesn't get to fly a plane; instead, he gets to fly a Macintosh (he thinks he still deserves hazard pay, though). Officially, he's a Signals Intelligence Systems Engineer, but he always seems to find himself doing Macintosh programming in one form or another. When he's not busy solving the world's problems or coming up with another useless hack (the boundaries between the two have become fuzzy lately), he likes to engage in sports that let him pretend he really does have wheels on his feet. He was recently spotted playing jazz guitar in a smoky little bar in upstate New York.•

DIRECT AND INDEXED COLOR

When an application does any drawing with Color QuickDraw, the ultimate result is to change some pixel values in a pixel map somewhere. Color QuickDraw in System 7 (and 32-Bit QuickDraw in earlier systems) supports two distinct types of color pixel maps: *direct* and *indexed*.

In direct pixel maps (those with pixel depths of 16 or 32 bits) the pixel values in memory specify RGB color information for the pixel directly. For example, the 32-bit direct pixel value $00AABBCC specifies a red component of $AA, a green component of $BB, and a blue component of $CC — 8 bits of color information each for the red, green, and blue components. (A 16-bit pixel value contains 5 bits of color information for each component.)

In indexed pixel maps (those with pixel depths up to 8 bits) the pixel values in memory don't directly specify the colors at all; instead they specify positions in a table of the available colors, called the *color lookup table* or just *color table* (sometimes called a *CLUT*). Figure 2 shows an example; in this case, the 8-bit pixel value $1C in memory actually represents the RGB color $AAAA BBBB CCCC, found at position $1C in the color table.

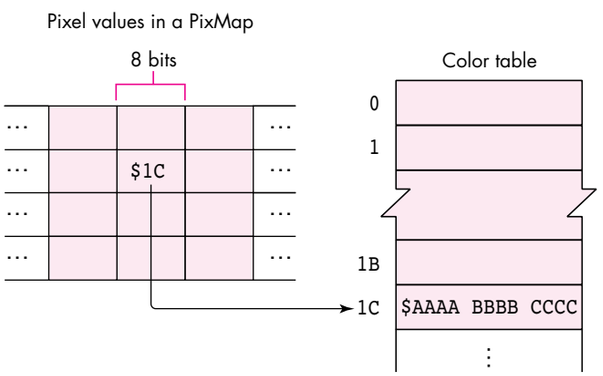


Figure 2. Indexed color

Typically, when an application wants to draw in a particular color, it specifies the desired color directly using an RGBColor record, and never deals with pixel values at all. Color QuickDraw and the Color Manager convert between RGBColors and pixel values as needed. If the application is drawing to a direct pixel map, the color information itself is used to build the pixel value, and no color table is involved. On the other hand, if the application is drawing to an indexed pixel map, Color QuickDraw uses the index of the closest-matching color in the color table as the pixel value (this process is called *color mapping*). But searching the entire color table for a match every time a pixel value is needed would be far too time-consuming, so the Color Manager uses something called an *inverse table* to speed up the lookup process.

INVERSE TABLES

An inverse table is something like a "reverse" color table: whereas a color table is used to convert an index to a color, an inverse table is used to convert a given color to an index into a color table. The conversion operation goes like this: You take some of the most significant bits of each color component and concatenate them, then use the resulting number as an index into the inverse table. The entry at that location in the inverse table holds, in turn, the index of the closest-matching available color in the corresponding color table. Figure 3 illustrates the process. Note that the closest-

matching color returned by this process need not match the original color exactly, since only a few of the most significant bits were used (the default is 4 bits).

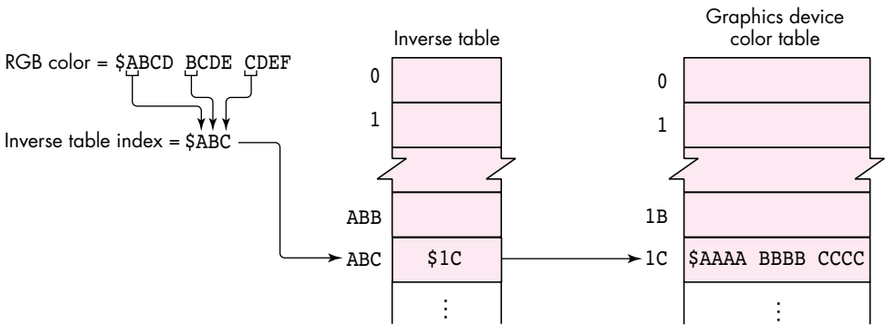


Figure 3. Inverse table with 4-bit resolution

Inverse tables are described in the Color Manager chapter of *Inside Macintosh: Advanced Color Imaging.*•

The number of bits each color component contributes to the inverse-table index is called the *resolution* of the inverse table. Higher resolutions would give you greater accuracy in color mapping, but also greatly increase the memory needed to hold the inverse table, so a maximum of 5-bit resolution is allowed. (Since there are three color components, each additional bit of resolution multiplies the size of the table eightfold.) You can use the Color Manager routine MakeITable to create inverse tables with resolutions of 3, 4, or 5 bits per component.

As an aside, Listing 1 shows how to temporarily change the resolution of the current graphics device's inverse table to 5 bits. (To permanently change the inverse table resolution, set the gdResPref field of the GDevice record, set the iTabSeed field of gdITable to the result of GetCTabSeed, and call GDeviceChanged.)

Listing 1. Temporarily changing the resolution of the inverse table

```
VAR
   gdh:        GDHandle;
   oldITabRes: INTEGER;

{ Get current graphics device. }
gdh := GetGDevice;

{ Get resolution of current inverse table. }
oldITabRes := gdh^^.gdITable^^.iTabRes;

{ Create a new inverse table at 5-bit resolution. }
MakeITable(NIL, NIL, 5);

{ Draw into a port on this device. }
...

{ Reconstruct inverse table at original resolution. }
MakeITable(NIL, NIL, oldITabRes);
```

THE IMPORTANCE OF THE CURRENT GRAPHICS DEVICE

An often misunderstood fact about Color QuickDraw is this: Color QuickDraw uses the *current graphics device's color table* when converting colors into indexed pixel values, ignoring the color table of the destination pixel map.

The inverse table is built from the color table in the graphics device's pixel map, not the one in the destination pixel map. When you're drawing to the screen, this is not a problem, since the destination pixel map and the current graphics device's pixel map match (the destination pixel map *is* the device's pixel map). However, it can be a problem when you're drawing offscreen (for example, when using CopyBits to copy one offscreen pixel map to another). If the color table of the destination pixel map doesn't match that of the current graphics device, you won't get the results you expect. The destination pixel map's color table is used only when converting the other way, from a pixel value to a color (for example, when the pixel map is actually displayed on the screen).

One of the nice things about using GWorlds for offscreen graphics is that you don't have to worry about this — GWorlds always have a graphics device associated with them, and routines such as SetGWorld ensure that the GWorld's pixel map and the graphics device's pixel map are synchronized for correct color mapping.

Note that inverse tables aren't found in pixel maps or color graphics ports. They're instead associated with *graphics devices* (astute readers may have noticed that the color table in Figure 3 was labeled "Graphics device color table" — this is why). So when converting RGBColors to indexed pixel values, the Color Manager uses the inverse table in the *current graphics device*. The implications of this are discussed in "The Importance of the Current Graphics Device."

DRAWBACKS OF INVERSE TABLES

The main problem with using inverse tables for color mapping is that because of their limited resolution, different colors can map to the same inverse table index. Inverse tables actually include some extra, undocumented information to allow the Color Manager to resolve such "hidden colors" — but examining this extra information is time-consuming, so some speed-sensitive QuickDraw routines don't always use it. One of these routines happens to be CopyBits, which is what accounts for our "100 shades of red" problem.

Let's look at the problem in more detail. The offscreen GWorld holding our image is 32 bits deep, allowing the pixel values to specify RGB colors directly, with a precision of 8 bits per component. When we copy the image to a window on an indexed graphics device, CopyBits uses an inverse table to convert these pixel values from direct to indexed. If our inverse table has a resolution of 4 bits (the default), it can only distinguish $2^4 = 16$ shades of red! (For example, all shades of red from RGB $0000 0000 0000 to $0FFF 0000 0000 will map to the same inverse-table index.) So *even if all 100 shades are available in the destination device's color table*, only 16 of them will actually be found and get drawn on the screen. This is why the actual result in Figure 1 has 16 bands of red instead of a continuum of shades.

The various depth conversion cases are discussed in the book *Programming QuickDraw* (see "Related Reading" at the end of this article) beginning on page 338.•

One way to deal with this problem would be to increase the resolution of the inverse table to 5 bits, which would give us 32 bands of red instead 16. Another approach would be to use the ditherCopy transfer mode in CopyBits. Both of these methods give better results but don't really solve the problem. After all, since we *do* have all the shades of red available, shouldn't there be some way to match the colors exactly?

INTRODUCING COLOR SEARCH PROCEDURES

Knowing that inverse tables might not be adequate for some applications, the QuickDraw engineers designed in a "hook" to allow developers to provide their own color-mapping code. Each GDevice record has its own linked list of custom *color search procedures;* there can be any number of such procedures installed for a given graphics device. As defined in the Color Manager chapter of *Inside Macintosh: Advanced Color Imaging*, a search procedure has the following interface:

```
FUNCTION SearchProc (VAR rgb: RGBColor; VAR position: LONGINT): BOOLEAN;
```

The rgb parameter is now always a VAR parameter. This was not true for direct-color destinations in 32-Bit QuickDraw prior to System 7. Also, note that *Inside Macintosh* Volume V incorrectly declared **rgb** as a value parameter. •

The Color Manager calls the search procedure with the RGB color it's trying to match, and expects the search procedure to do one of three things:

- Match the color — In this case, the search procedure returns the *pixel value* for the color in the **position** parameter, and a result of TRUE. On an indexed graphics device, the **position** parameter should contain the index of the appropriate color in the graphics device's color table. On a direct graphics device, this parameter should be set to the appropriate direct-color value.
- Modify the color — In this case, the search procedure modifies the **rgb** parameter and returns a result of FALSE. Color QuickDraw ignores the **position** parameter. See the next section for examples of using this technique.
- Do nothing — In this case, the search procedure simply returns a result of FALSE, leaving its parameters untouched.

The Color Manager runs through the list of search procedures for the current graphics device, calling each procedure in turn until one of them returns TRUE. If no search procedure returns TRUE, it uses the default color-mapping method on the original (or possibly modified) color. For indexed graphics devices, this means using the inverse table. For direct graphics devices, "color mapping" simply involves truncating the RGBColor components to the appropriate size.

When called with an arithmetic transfer mode, CopyBits calls custom color search procedures *before* the arithmetic operation is performed. You can get around this by doing the desired operation first and then installing the search procedure and using CopyBits with srcCopy mode to display the result. •

The search procedure mechanism provides a solution to our "100 shades of red" problem. If we know where all the shades are located in the current graphics device's color table, we can write a search procedure that returns the correct index for any shade of red we pass to it. This will avoid the bands shown in the actual result in Figure 1 and instead produce the expected result, with the exact colors intended. Of course, this technique can be applied to *any* image if we know where to find all the colors we need in the color table; we'll examine the technique in more detail later.

MODIFYING SEARCH COLORS

The fact that the desired color is passed to the search procedure through a variable parameter is significant: it means that the procedure can actually modify the color value it receives. In this case, the search procedure should return FALSE, telling

QuickDraw to perform the default color mapping on the *modified* color. This technique opens up several possible uses for search procedures.

One such application is color separation for three-color printing. The snippet called SearchProcs & Color Separation on this issue's CD shows how to do this. To separate all the greens from an image, for instance, you could install a search procedure that sets the red and blue RGB components to 0. Listing 2 shows a simple example.

Listing 2. Search procedure to separate green colors

```
FUNCTION GreenSepProc (VAR rgb: RGBColor; VAR position: LONGINT):
                          BOOLEAN;
BEGIN
   WITH rgb DO
      BEGIN
         red := 0;         { Set red and blue RGB components to 0, }
         blue := 0         { keeping only the green component. }
      END;
   GreenSepProc := FALSE
END;
```

A similar search procedure could be used to darken or lighten an image. For example, you could use the code in Listing 3 to darken the blue component of an image by a factor of 2.

Listing 3. Search procedure to darken the blue component

```
FUNCTION DarkenBluesProc (VAR rgb: RGBColor; VAR position: LONGINT):
                             BOOLEAN;
BEGIN
   rgb.blue := BSR(rgb.blue, 1);      { Shift right to divide by 2. }
   DarkenBluesProc := FALSE
END;
```

WHAT'S THE CATCH?

As usual, you do pay a price for all this functionality: search procedures definitely slow down the drawing process. Just how badly depends on several factors. In the case of CopyBits, the speed is most directly affected by the depth of the source and destination pixel maps. If the source pixel map uses indexed color, the search procedure needs to be called only once for each color in the source map's color table. For direct color, it must be called for *every pixel!*

Consider the very simplest search procedure — one that just returns FALSE without doing anything:

```
FUNCTION NothingSearchProc (VAR rgb: RGBColor; VAR position: LONGINT):
                             BOOLEAN;
BEGIN
   NothingSearchProc := FALSE
END;
```

(A search procedure that did nothing but return TRUE would actually be faster, but would be useless, since the value in the **position** parameter would be garbage; returning FALSE ensures that at least normal color mapping will take place.) Table 1 compares the speed of a CopyBits operation with and without this search procedure, along with the speed of using the ditherCopy transfer mode in place of srcCopy. The source image is the one shown in Figure 1.

Table 1. Influence of search procedure on CopyBits speed

Machine Type	srcCopy	ditherCopy	srcCopy With Search Procedure
Macintosh IIci, Apple 8•24 card	21	57	83
Macintosh Quadra 800, built-in video	8	21	23

Note: Speeds are given in ticks, and are for ten successive calls to CopyBits, copying a 100-by-100-pixel, 32-bit-deep image to an 8-bit screen.

As you can see, CopyBits with an installed search procedure runs just a little slower than a dithered CopyBits. Note that the figures in the table are very rough. Several other factors contribute significantly to the speed difference when a search procedure is installed, such as the size of the source image and the number of colors it contains. You'll also get different results depending on what drawing routines you call with the search procedure installed. But the "dithered CopyBits" rule of thumb seems to work quite well as a general guide.

It's up to you to decide whether the speed penalty for using a custom color search procedure is worth the improved display quality. For image-processing applications, where color accuracy is probably more important than speed, search procedures can be very useful; for applications such as arcade-style video games, which depend on real-time graphics, they're probably not the way to go.

SOLVING THE "SHADES OF RED" PROBLEM

It's very common these days for applications to prepare an image offscreen, using a 32-bit GWorld, before transferring it to the screen for display. Despite the decreasing cost of 24-bit graphics cards, indexed 8-bit color is still a very common configuration, and even users with direct color capability spend a lot of time in 8-bit mode, which can lead to anomalies like the "100 shades of red" problem. As mentioned earlier, we can use a custom color search procedure to draw direct pixel images into indexed graphics devices with exact color reproduction, provided that all of the colors are actually available in the destination device's color table.

The way to make the colors available on the device is of course to use the Palette Manager, attaching a palette of the needed colors to the window you're drawing in. (This works only if other applications aren't "hogging" too many colors.) Getting the right colors from a picture or pixel map won't be discussed in any detail here, but the sample code uses the octree method described in the article "In Search of the Optimal Palette" in *develop* Issue 10. It's probably easier to use the built-in popular and median color-sampling methods, but they truncate colors to 5 bits per component, meaning that they won't return separate palette entries for colors that differ only in the lower bits, as our shades of red do. The octree method doesn't truncate the colors, so it can be used to find *all* the colors in the image (assuming the image contains fewer than 256 colors). Another approach is demonstrated in the snippet CollectPictColors on the CD.

Once the colors are available, we can write a search procedure that simply searches the graphics device's color table and returns the index of the requested color. (If the color table doesn't contain all the needed colors, the search procedure may have to return FALSE; QuickDraw will then use the inverse table to map these colors, which can lead to unexpected results. See the section "Evaluating the Results," later in this article, for more on this.)

THE BRUTE-FORCE APPROACH

In true hacker fashion, let's try the brute-force approach first: we can simply scan straight through the current graphics device's color table and stop when we find a match. Listing 4 shows the code.

Listing 4. Brute-force search procedure

```
FUNCTION BruteSearchProc (VAR theRGB: RGBColor; VAR position: LONGINT):
                              BOOLEAN;
   VAR
      i:          INTEGER;
      gdh:        GDHandle;
      colorTab:   CTabHandle;
BEGIN
   { Get handle to current device. }
   gdh := GetGDevice;

   { Get color lookup table from current device. }
   colorTab := gdh^^.gdPMap^^.pmTable;

   { If the color table exists, loop through all its entries until we }
   { find a match. }
   IF colorTab <> NIL THEN
      WITH colorTab^^ DO
         FOR i := 0 TO ctSize DO
            WITH ctTable[i] DO
               IF (theRGB.red = rgb.red) & (theRGB.green = rgb.green) &
                     (theRGB.blue = rgb.blue) THEN
                  BEGIN
                     { We found the color, so pass back its index and }
                     { return TRUE. }
                     position := i;
                     BruteSearchProc := TRUE;
                     EXIT(BruteSearchProc)
                  END;

   { We didn't find the color in the table, so return FALSE to tell }
   { QuickDraw to use the default mapping method. }
   BruteSearchProc := FALSE
END;
```

If we install this search procedure and draw the "100 shades of red" image, it will find all 100 shades and produce the expected image. Unfortunately, it's *very* slow: a CopyBits with srcCopy mode using this search procedure takes 30 to 40 times as long as a dithered CopyBits.

HASH TABLES: A BETTER WAY

We can speed up our search procedure by using a hash table instead of a brute-force linear search. (Hash tables are familiar to most of you from basic computer science classes, and are described in any good book on algorithms, such as *Algorithms* by Robert Sedgewick.) In our case, we'll use the RGB color value as a hash key to find the corresponding color table index. For our hash function, we'll use the MOD operator to find the remainder of the hash key relative to some suitably chosen prime number. The bigger we make this prime number, the better the performance of the hash function will be. Assuming that the target device uses 8-bit indexed colors (for most images, any lower color depth will yield a color table too small to hold all the colors we need), we'll be working with a color table of 256 colors. We'll choose 251, a prime number near 256, as the divisor for our hash function. The MOD operator can't operate directly on 48-bit RGBColor records, so we'll use the high-order 8 bits of each color component to form a 32-bit integer of the form $00rrggbb (the same as a 32-bit pixel value) and use that for our key into the hash table.

Figure 4 shows the data structure containing our hash table. RGBHashArray is a zero-based array of records of type RGBHashNode. Each node holds a 32-bit color value (rgbComp), along with the index at which that color is stored in the color table. Nodes whose colors map to the same hash value are chained together in a linked list, with each node's **next** field holding the array index of the next node in the chain (this collision resolution method is called *separate chaining*). The first kPrimeRecords (251) entries in the hash array hold header nodes for all possible hash values; these point into the rest of the array, which holds the data nodes themselves.

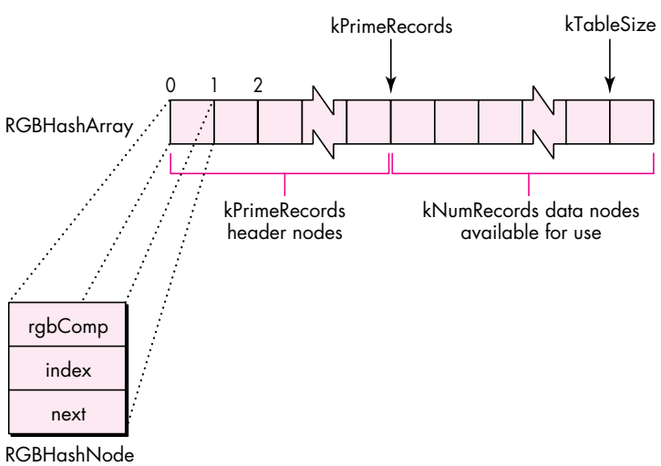


Figure 4. Hash table data structure

The data structure definitions for our hash table are shown in Listing 5. In addition to the array holding the table's contents, there's a short header containing the index of the next available data node along with the color table's *seed value* at the time the hash table was built. We can use the latter to keep our hash table synchronized with the color table. Any time QuickDraw changes the contents of the graphics device's color table, it also changes its seed value. Thus if the seed values in the hash table and color table don't match (as checked by the routine in Listing 6), we know the color table has been changed and we need to rebuild our hash table before using it.

There are two straightforward procedures, not shown here, for initializing the hash table and for clearing it out before building or rebuilding its contents (see the code

Listing 5. Hash table data structures

```
CONST
   kNumRecords = 256;         { Number of colors in color table }
   kPrimeRecords = 251;       { Number of hash entries }
   kTableSize = kPrimeRecords + kNumRecords - 1;
                              { Total size of (zero-based) hash array }

TYPE
   RGBCompressedColor = LONGINT;    { Color in 32-bit form ($00rrggbb) }

   { Data structure for hash table nodes }
   RGBHashNode = RECORD
      rgbComp: RGBCompressedColor;  { RGB color in compressed form }
      index:   INTEGER;             { Index of matching color in }
                                    { color table }
      next:    INTEGER              { Array index of next node in list }
   END;

   { Data structure for array to store hash table data }
   RGBHashArray = ARRAY[0..kTableSize] OF RGBHashNode;

   { Data structure for hash table itself }
   RGBHashTable = RECORD
      nextEntry:     INTEGER;    { Array index of next unused data node }
      curCTabSeed:   LONGINT;    { Value of color table seed when hash }
                                 { table was created (indicates when}
                                 { hash table must be updated) }
      table:         RGBHashArray      { Hash table contents }
   END;
   RGBHashTablePtr = ^RGBHashTable;

VAR
   gRGBHash: RGBHashTablePtr;    { Global hash table pointer }
```

Listing 6. Checking the validity of the hash table

```
FUNCTION HashTableNeedsUpdate (ctab: CTabHandle;
                               rgbHash: RGBHashTablePtr): BOOLEAN;
BEGIN
   HashTableNeedsUpdate := ctab^^.ctSeed <> rgbHash^.curCTabSeed
END;
```

on the CD for details). RGBHashInit zeroes out the entire hash table, while RGBHashClear clears only the list headers, making the table appear empty; there's no need to zero the data nodes themselves.

The procedure for inserting a color into the hash table is shown in Listing 7. It starts by doing some bit manipulation to convert the RGBColor to 32-bit form. It then uses the result to compute the hash-table index for the given color by finding its remainder modulo 251. Next, it fills in the fields of the next available hash node and inserts it at the head of the linked list starting at the computed index. Finally, it increments the hash table's nextEntry field to point to the next hash node in the array.

Listing 7. Inserting a color in the hash table

```
PROCEDURE RGBHashInsert (rgbHash: RGBHashTablePtr; rgb: RGBColor;
                         cTabIndex: INTEGER);
   VAR
      compressedRGB: RGBCompressedColor;
      hashIndex:     INTEGER;
BEGIN
   { Reduce 48-bit RGB value to 32-bit compressed form. }
   WITH rgb DO
      compressedRGB := BSL(BAND(red, $0000FF00), 8) +
               BAND(green, $0000FF00) + BSR(BAND(blue, $0000FF00), 8);

   { Compute hash-table index. }
   hashIndex := compressedRGB MOD kPrimeRecords;

   WITH rgbHash^ DO
      BEGIN
         { Store color data in next available node. }
         WITH table[nextEntry] DO
            BEGIN
               rgbComp := compressedRGB;       { Actual RGB color }
               index := cTabIndex;             { Index in color table }

               { Insert this node at front of linked list. }
               next := table[hashIndex].next;
               table[hashIndex].next := nextEntry
            END;

         { Update to next available node. }
         nextEntry := nextEntry + 1
      END
END;
```

Building a hash table from the current graphics device's color table is relatively straightforward (Listing 8). First we save the state of the color table handle and lock it in case we do something that moves memory while the handle is dereferenced. (Our code doesn't currently do anything to move memory, but if we should change it in the future so that it does, this precaution ensures that it will still work.) Next we call our RGBHashClear procedure to clear the hash table's list headers to empty, and save the color table's seed value so that we can tell when the hash table needs updating. Finally, we step through the contents of the color table, inserting each color into the hash table with RGBHashInsert (Listing 7). Then all that's left is to restore the color table handle to its original state, and the hash table is ready for use by our search procedure.

Finally, we get to the real heart of the hash-table search procedure, RGBHashSearch (Listing 9). First we pack the 48-bit RGBColor value into 32 bits. Next, we compute the hash-table index for the given color and retrieve the list header for that hash value. If the list is nonempty, we step through it, comparing the RGB color stored in each node with the color we're looking for. If the colors match, we get the index of the corresponding color table entry from the data node and return TRUE. If we don't find the desired color, we return FALSE to indicate that the color was not in the hash table. Note that this will happen only if the source image contains colors that didn't fit in the color table (an example of this is given in the next section).

Listing 8. Building the hash table

```
PROCEDURE CTab2Hash (ctab: CTabHandle; rgbHash: RGBHashTablePtr);
   VAR
      state:   SignedByte;
      i:       INTEGER;
BEGIN
   { Save state of color table handle and lock it. }
   state := HGetState(Handle(ctab));
   HLock(Handle(ctab));

   { Clear hash table to empty. }
   RGBHashClear(rgbHash);

   WITH ctab^^ DO
      BEGIN
         { Save current seed value. }
         rgbHash^.curCTabSeed := ctSeed;

         { Step through contents of color table. }
         FOR i := 0 TO ctSize DO
            { Insert each color into hash table with its index. }
            WITH ctTable[i] DO
               RGBHashInsert(rgbHash, rgb, i)
      END;

   { Restore original state of color table handle. }
   HSetState(Handle(ctab), state)
END;
```

Listing 9. Searching the hash table

```
FUNCTION RGBHashSearch (rgbHash: RGBHashTablePtr; rgb: RGBColor;
                           VAR index: LONGINT): BOOLEAN;
   VAR
      compressedRGB: RGBCompressedColor;
      hashIndex:     INTEGER;
      chainIndex:    INTEGER;
      nextIndex:     INTEGER;
BEGIN
   WITH rgb DO
      { Reduce 48-bit RGB value to compressed form. }
      compressedRGB := BSL(BAND(red, $0000FF00), 8) +
            BAND(green, $0000FF00) + BSR(BAND(blue, $0000FF00), 8);

   { Compute hash-table index. }
   hashIndex := compressedRGB MOD kPrimeRecords;

   WITH rgbHash^ DO
      BEGIN
      { Get array index of first node in list. }
      chainIndex := table[hashIndex].next;
```

(continued on next page)

Listing 9. Searching the hash table *(continued)*

```
      WHILE chainIndex <> 0 DO              { Loop till end of list. }
         { Is this the color we want? }
         IF table[chainIndex].rgbComp = compressedRGB THEN
            BEGIN    { If so, pass back its CLUT index and return TRUE }
               index := table[chainIndex].index;
               RGBHashSearch := TRUE;
               EXIT(RGBHashSearch);
            END
         ELSE        { Otherwise go to the next node. }
            chainIndex := table[chainIndex].next;
      { If we got here, either there were no links at this hash-table }
      { address, or we reached the end of the list. Both cases }
      { indicate that the color is not in the CLUT, so return FALSE. }
      RGBHashSearch := FALSE;
   END;
END;
```

Listing 10 shows how to install our search procedure for use in a drawing operation (gSearchProcUPP is a universal procedure pointer that points to our search procedure, which is simply a wrapper that calls RGBHashSearch). The Color Manager routines AddSearch and DelSearch, respectively, install and remove a search procedure for the current graphics device. Note that we install our search procedure just before the drawing operations that use it, and remove it immediately afterward. This is because the search procedure will be called for *any* drawing that occurs on the device it's attached to, and can significantly affect performance. Before installing and using our search procedure, we call our HashTableNeedsUpdate function (Listing 6) to compare the hash table's seed value with that in the current color table. The function returns TRUE if the seed values don't agree; this tells us to rebuild the hash table with CTab2Hash (Listing 8) before using our search procedure.

Astute readers may wonder what happens if the drawing area spans more than one screen in a multiple-monitor configuration, since search procedures "belong" to particular devices. Our sample code deals with multiple devices simply by calling DeviceLoop to do its drawing, installing the search procedure only on 8-bit color devices; on any other devices, CopyBits is called with ditherCopy mode.

EVALUATING THE RESULTS

Has all this optimization been worth it? Table 2 compares the speeds of the various search procedures, again using CopyBits in srcCopy mode to copy the image shown in Figure 1 from a 32-bit offscreen GWorld to an 8-bit device. For comparison, the speed of a "nothing" search procedure is also shown. Clearly, the work has paid off — the hash-table search procedure is over 15 times as fast as the brute-force approach, and is certainly comparable to a dithered CopyBits. In some cases (for example, when drawing an image in a zoomed-in state), our hash table technique is actually as fast as (or faster than) a dithered CopyBits.

Although our hash-table search procedure gives impressive results, there are certainly cases where its performance is less than optimal. The hash table method assumes that all of the colors in the source image can be loaded into the current graphics device's color table. If this condition doesn't hold, the search procedure will still work, but it won't be able to find colors that aren't in the color table, so QuickDraw will use the

Listing 10. Installing and removing a search procedure

```
{ Get color table from current graphics device. }
gdh := GetGDevice;
ctab := gdh^^.gdPMap^^.pmTable;

{ Update hash table if necessary. }
IF HashTableNeedsUpdate(ctab, gRGBHash) THEN
   CTab2Hash(ctab, gRGBHash);

{ Install search procedure right before drawing. }
AddSearch(gSearchProcUPP);

{ Example drawing code }
CopyBits(BitMapPtr(thePixMap^)^, myWindow^.portBits, srcRect, destRect,
         srcCopy, NIL);

{ Remove search procedure right after drawing. }
DelSearch(gSearchProcUPP);
```

Table 2. Comparison of search procedure speeds

Machine Type	Nothing Procedure	Brute-Force Procedure	Hash Procedure
Macintosh IIci, Apple 8•24 card	83	2234	175
Macintosh Quadra 800, built-in video	23	691	48

Note: Speeds are given in ticks, and are for ten successive calls to CopyBits, copying a 100-by-100-pixel, 32-bit-deep image to an 8-bit screen.

default inverse-table mapping method for those colors. This can give unexpected results. For example, Figure 5 (on the inside back cover of this issue) shows a version of the "Better Bull's eye" image from *develop* Issue 1 (from the article "Realistic Color for Real-World Applications"), drawn using the hash-table search procedure.

The image in Figure 5 has more than 256 distinct colors. The results may look all right at first glance, but if we zoom in on the top right corner of the image (Figure 6, also on the inside back cover), we can see unwanted bands of gray. Some of the actual grays that were supposed to appear at these locations were not available in the graphics device's color table. As a result, they were color-mapped to the closest available gray at a 4-bit resolution, resulting in banding.

A similar problem can result if you have several windows displaying different images at once. The frontmost window will display correctly, but the others may not have the correct colors available. Usually this isn't important, since the frontmost window is generally the one you're concerned with. Typically, you should install the search procedure only when drawing in the frontmost window.

Another, more subtle case where our search procedure can give unexpected results is when the destination rectangle passed to CopyBits is smaller than the source rectangle. If the source image uses direct color, CopyBits will average the color values of adjacent pixels to produce the reduced image. This usually gives more visually

appealing results than just dropping whole rows of pixels; but in this case, since averaging can produce colors that aren't in the color table, we run into the same kind of problem we've been discussing. (There's no problem when the destination rectangle is *bigger* than the source rectangle, since CopyBits will simply replicate existing pixels without introducing any new colors into the image.)

MAKING IT BETTER

Our hash-table search procedure is certainly much more efficient than the brute-force approach, but it can be improved still further. The most obvious idea would be to reimplement the code in assembly language for maximum efficiency, although this hampers portability and may not result in much of a speed improvement, depending on how good your compiler is. Another area for improvement might be the hashing algorithm itself: we could try a different hash function or another method of collision resolution. However, since the hash table in this application is so small, this may not be worth the effort.

A useful extension would be to find the closest match for colors that are *not* in the color table. This would alleviate the problems that occur when the image has too many colors to fit in the color table. Abandoning the hash table in favor of a tree-based algorithm might work, but it would be hard to make it as fast as the hash table method. Another approach might be to use some color-quantization algorithm to reduce the total number of colors in the image to 256 — but of course that would mean changing the actual image data.

NOW IT'S UP TO YOU

Custom color search procedures are one of the least-used methods for customizing Color QuickDraw. In this article, we've seen several practical uses for them — now it's up to your creativity to find others. (Let us know if you do!)

RELATED READING

- *Inside Macintosh: Advanced Color Imaging,* on this issue's CD in draft form and forthcoming in print from Addison-Wesley. See the Color Manager chapter. This is the most thorough documentation in *Inside Macintosh* for color search procedures and color mapping with inverse tables. You'll also find some useful information in the Palette Manager chapter.
- *Inside Macintosh: Imaging With QuickDraw* (Addison-Wesley, 1994), Chapter 1, "Introduction to QuickDraw," Chapter 4, "Color QuickDraw," Chapter 5, "Graphics Devices," and Chapter 6, "Offscreen Graphics Worlds."
- *Programming QuickDraw* by David Surovell, Frederick Hall, and Konstantin Othmer (Addison-Wesley, 1992). Everything you ever wanted to know about QuickDraw. In particular, see "Graphics Devices" in Chapter 3, "Drawing in Color," and see "Pixel Processing Traps" and "Depth Conversion and Dithering" in Chapter 7, "Image Processing with QuickDraw."
- "In Search of the Optimal Palette" by Dave Good and Konstantin Othmer, *develop* Issue 10. How to use the Picture Utilities Package to obtain a palette with the best colors for displaying an image on an indexed device.
- Macintosh Technical Notes "Principia Off-Screen Graphics Environments" (QD 13) and "Of Time and Space and _CopyBits" (QD 21).

Thanks to our technical reviewers Joseph Maurer, Don Moccia, Guillermo Ortiz, and Nick Thompson.•

Having trouble getting a smooth color image?

See the article “Custom Color Search Procedures” on page 66.

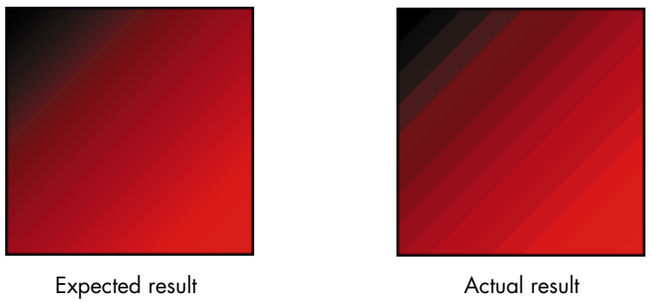


Figure 1. CopyBits of an offscreen color ramp to a window

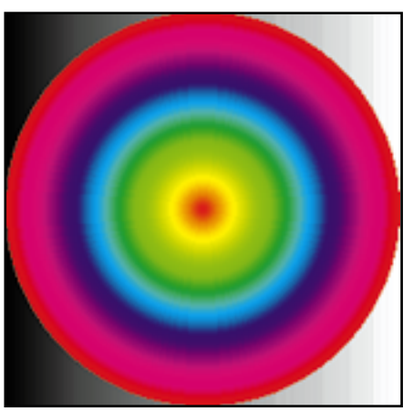


Figure 5. Bull’s eye

Figure 6. Enlargement of top right corner of bull’s eye

ACCORDING TO SCRIPT

Scripting Quandaries

CAL SIMONE

The world of scripting is heating up. More and more developers are getting hip to scriptability, but they're discovering it's not a simple matter — implementing scripting support in an application involves many complex issues. That's where this column comes in.

My article, "Designing a Scripting Implementation," in the last issue of *develop* (Issue 21) offered approaches you'll want to consider when designing your scripting implementation (your object model hierarchy and your human vocabulary), as well as some basic tips and guidelines for making your application scriptable. This column picks up where that article left off, elaborating on many of the same issues and providing further tips, tricks, and standards for scriptability. In this first installment of the column, I'll clarify a couple of points that some developers found confusing in the article; then I'll give some new guidelines.

STARTING WITH MENU COMMANDS

In my previous article, I suggested that one place to start your scripting implementation is to implement your menu commands. Permit me to clarify. Looking at menu commands is useful because they can suggest functionality that users should be able to script. But to maintain consistency with other object model–based applications, you should *not* implement scripting commands that simply mimic the menu commands.

Resist the temptation to fill up your dictionary with all your menu items, even though it might be easier to write your event handlers this way. Instead, implement the object model (discussed at length in "Apple Event Objects and You" in *develop* Issue 10 and "Better Apple Event Coding Through Objects" in Issue 12). Keep the number of verbs small, implement standard verbs wherever applicable, and let the script writer apply those verbs (especially **make**, **get**, and **set**) to a large number of objects.

LOWERCASING VERBS, TOO

I mentioned in the last issue that you should begin all the terms in your dictionary (except for proper names) with lowercase letters. This applies not only to object names but to verbs as well. There are two reasons for this rule. First, AppleScript allows commands to be embedded within commands (particularly when the embedded command is from a scripting addition such as **choose file**), and these complex command statements read better when all the verbs are in lowercase. For example:

```
set myFile to choose file with prompt "Pick it!"
```

Second, if you were to include an entire suite (such as the Required or Core suite) from the system dictionary and then add your own verbs starting with uppercase, you'd end up having a mixture of verbs beginning with uppercase and those beginning with lowercase displayed in your dictionary, not a pretty picture.

HANDLING REQUESTS TO GET AN OBJECT

Developers are sometimes confused about how to handle a request from an Apple event or a script to get an object. In the early days, especially when programs were communicating directly with other programs, developers thought that getting an object meant returning the internal data structure of the object, such as a WindowRecord or other C structure. In today's scripting world, you should *never* return raw internal data structures. What you should return depends on the object or property requested.

Applications, windows, documents, and interface elements. In most cases, when the object requested is an application, a window, a document, or an interface element (such as a button), you should return an error since you can't really bring these types of objects into your script. For example, **get window 1** should result in an error. One exception is that if your application is a script-controlled interface builder, you might want to return references to the windows, documents, and interface elements.

You should provide a **contents** property for objects such as windows and documents. When this property is requested (as in **get the contents of window 1**), you

CAL SIMONE (AppleLink MAIN.EVENT) spends a lot of time helping others make sense of AppleScript, escaping from Washington DC about once a month to promote or teach AppleScript. He lives in Adams-Morgan, the city's only real ethnic neighborhood; full of cultural diversity, it boasts 45 restaurants representing 18 different nationalities in just one block.•

should return the entire contents of the object specified, if appropriate.

Text elements. When the object requested is a text element, such as a word or a character (as in **get word 4 of paragraph 3 of document "Sales"**), you should return the contents of the object itself as a string, such as "Fred" (word) or "x" (character).

Graphics objects. When the object requested is a simple graphics object where a standard format is in widespread use, you should return the contents of the object itself, just as for text elements. For example, for a PICT you would return the picture's data; for a point or a rectangle you would return a list of integers. When the object requested is a compound graphics object, such as a grouped graphic, you should return a reference to the object, in the form of an object specifier.

Cells, fields, and form elements. You should provide a **value** property for objects such as cells in a spreadsheet, fields in a database, or elements of a form. (In essence, this property is the same thing as the **contents** property, but in natural language, people usually refer to the value of a cell or field and the contents of a window. Making a distinction between these two kinds of properties thus preserves a natural language style.) When this property is requested (for instance, **get the value of cell 3 of row 7**), you should provide the content data for the object specified. If you want script writers to be able to get the value in more than one form, provide the **as** parameter (for example, **get the value of . . . as styled text**).

Rows, records, and entire forms. When a row or column in a spreadsheet or table, a record in a database, or an entire form is requested, you can return a list of the data values in each field (for example, **get record 43 of database "Employees"** might return {"Fred", 45.00}). However, it might be more appropriate to return a reference to the object, since the list might be too large or might contain large data. By the same token, either a list or a reference should be returned when the **current record** property that some developers have implemented is requested.

An object's internal data. Rather than dealing with raw internal data structures, script writers should be able to get any piece of attribute data for an object through the object's properties, by using the **get <property>** construct. If you want to let them get all the attribute data at once with a single **get** command, provide a **properties** property (as, for example, QuarkXPress does), return a record with the values for all the properties, and provide a definition for the record as a new abstract class in the Type Definitions suite (discussed in my article in Issue 21).

As you can see, it's not always clear how to respond to an object request. While not a hard rule, the basic guideline is this: If an object is an elementary piece of data, such as a word or a rectangle, return its value directly; if it's a structure, such as a record or a row, return a list of its values or a reference to the structure; and if it's a complex or abstract object (especially part of the user interface like a window or a button), return either an error or a reference to the object. What you decide to return will often depend on the way you want script sentences to read.

THE APPLICATION AS CONTAINER

How to create and extract object specifiers from Apple events is explained in *Inside Macintosh: Interapplication Communication*. The outermost container for an object specifier is always the application itself, represented by a container of type typeNull. A null container is the only proper way to specify the application level of the containment; do not include a cApplication object specifier as a container. Figure 1 illustrates the right way to specify the application as a container. The wrong way in this case would be to have a third container level labeled cApplication between cDocument and cNull.

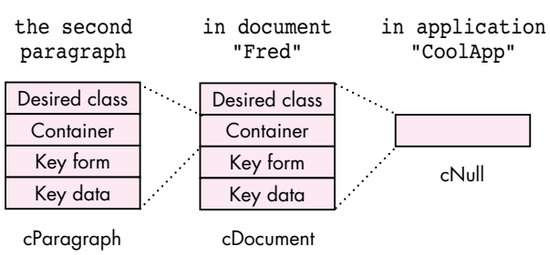


Figure 1. Specifying the application as a container

I'M OUTTA SPACE

Remember, everything you do in your event handlers, your object accessor functions, your error messages, and your dictionary will significantly affect the experience your users have in writing scripts. In future columns, I'll be discussing how to organize your dictionary; the increasingly crowded naming space for terms; how to organize and propose a new standard suite, event, or object definition; recording; and, as always, more tips, tricks, and conventions.

Thanks to Sue Dumont and Jon Pugh for reviewing this column.•

The OpenDoc User Experience

OpenDoc, Apple's compound-document architecture, offers a new experience to users. This article gives developers a guided tour of OpenDoc's human interface and describes its conceptual model. We provide the necessary background for helping you fit your application into the OpenDoc world, and present some of the decisions you'll have to make that represent a departure from today's applications.

DAVE CURBOW AND ELIZABETH DYKSTRA-ERICKSON

OpenDoc provides a new user paradigm: the user focuses on creating a document or performing a task, rather than on using a particular application. Understanding the OpenDoc user experience is a prerequisite to developing OpenDoc part editors that are consistent with and supportive of the OpenDoc design model. We've talked with developers at OpenDoc training classes who had written code without realizing what user features they had implemented; this article will provide a context for the OpenDoc code you write. The article describes the OpenDoc user experience on the Macintosh, but most of it also applies to Microsoft Windows and IBM OS/2.

Developer releases of OpenDoc are available through a number of different sources. The documentation provided in these releases, which includes the *OpenDoc Programmer's Guide*, *OpenDoc Human Interface Guidelines*, and the *Drag and Drop Human Interface Guidelines*, can give you much more detail on what's covered here (we concentrate on the basics, so a lot of exceptions aren't covered). Some of the technical basics of OpenDoc are also covered in the *develop* articles "Building an OpenDoc Part Handler" in Issue 19 and "Getting Started With OpenDoc Graphics" in Issue 21.

ALL ABOUT PARTS

OpenDoc provides an object-oriented user model, where documents are objects that contain other objects, and where each object may have distinct behaviors. However, *object* isn't a term that typical users understand in a document context, so we use *part* instead (for "part of the document").

DAVE CURBOW is the technical lead of the OpenDoc Human Interface team. Before that he worked on AppleScript and developer tools such as ResEdit. In an earlier life he was a software engineer on the Xerox Star and a now-forgotten mainframe operating system. When he escapes from the office, Dave can often be found working with his wife on their house or exploring cathedrals, castles, and other wonders in England (including the Kew Bridge Steam Museum). It's well known that Dave can be bribed with dark chocolate.•

ELIZABETH DYKSTRA-ERICKSON is a recent addition to the OpenDoc Human Interface team. She comes to Apple from research and product development in collaborative technology and interactive multimedia at US WEST Technologies, Pacific Bell, and the University of Amsterdam. In her copious free time, she teaches human-computer interaction at the University of San Francisco, conspires to resurrect her 1980's tech-punk band, and marvels with her husband at the havoc potential of their two-year-old daughter.•

Parts enable all kinds of content to be combined into a single document. The user sees each part as a self-contained entity with its own content, behavior, and set of properties. Each part contains one kind of data that's intrinsic to it, and may contain other parts as well.

PARTS AND DOCUMENTS

Every document consists of one or more parts: a single part at its top level, called the *root part*, and other parts that are embedded in the root part. Documents always reside on the desktop or in a folder — that is, they appear in the Finder. (Parts embedded inside other parts aren't considered to be documents.) Users assemble a document by embedding parts as needed, with drag and drop or with the Paste and Insert commands, as we'll see later. Parts can be dragged between documents or onto the desktop (where they become documents); documents can be dragged from the desktop into other documents (where they become embedded parts).

The root part of a document determines the document's overall characteristics such as its basic editing metaphor (for instance, text, drawing, or spreadsheet), the size of its work area (its "page"), its printing options, and whether saving is manual or automatic.

PART CONTENT: INTRINSIC AND EMBEDDED

Every part has some kind of *intrinsic content*, as defined by the part developer. This is the content that's natural to the part, such as characters and paragraphs in a text part, or lines, circles, rectangles, and so on in a graphics part. In addition to its intrinsic content, a part may contain embedded parts that have their own intrinsic content, as shown in Figure 1.

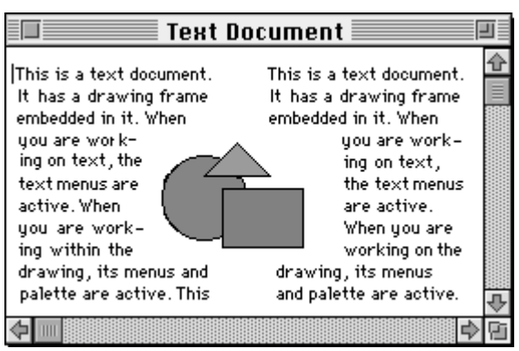


Figure 1. A text part with an embedded graphics part

There's no requirement that a part be able to contain embedded parts, although it's usually desirable. Some parts have content models in which embedding doesn't make sense — for example, sound parts or parts that display information, such as clocks or stock tickers. As a developer, you must decide whether it's desirable for parts you create to allow other parts to be embedded. But, a key characteristic of OpenDoc is that *if a part can contain one kind of part, it can contain all kinds of parts.* (Contrast this with traditional documents, which can contain only certain standard data types, such as text, PICT, and TIFF, in addition to their intrinsic content.) To a part, any parts embedded within it are "black boxes" — parts need know nothing about the internal structure or semantics of embedded parts.

PART BEHAVIOR: EDITORS AND VIEWERS

There's little difference between the appearance of the OpenDoc window in Figure 1 and that of a similar window in a page-layout application of today; manipulation of

the window contents, however, can be very different. When users interact with OpenDoc parts, the resulting behavior is determined by part editors and part viewers.

A *part editor* is a full-featured OpenDoc software component that allows the creation, editing, and viewing of parts of a particular kind, just as a conventional application allows manipulation of documents now. Like applications, part editors are sold or licensed and are legally protected from unauthorized copying and distribution. You supply users with the part editor (which the user installs in the Editors folder in the System Folder) and also a stationery pad (which the user will double-click or drag to create an OpenDoc document or part).

A *part viewer* is a special, limited type of part editor that can display and print a particular kind of part but can't be used to create or edit such a part. Often a part viewer will just be a part editor with its editing and part-creation capabilities removed. It's important that part viewers be widely available, to allow portability of OpenDoc compound documents across machines and platforms. We encourage you to create and freely distribute part viewers without restriction for all the kinds of parts that you support. Wide availability of a particular part viewer encourages purchase and use of its equivalent part editor, because users will know that other users will be able to view parts created with that editor.

Note that it may be possible to view a part even when neither its editor nor its viewer is present; translation may occur that substitutes a different, compatible editor. For example, suppose a user creates a document with a text editor named SurfWriter and sends it to someone who doesn't have the SurfWriter editor; the document is translated to a similar format supported by a text editor that the receiving user does have.

Users don't work with icons for part editors and part viewers the way they work with application icons today: editors and viewers aren't launched by double-clicking. So, as a distinct break from application icons, the icons for editors and viewers have a unique shape. This shape provides maximum customizable space for your identifying elements, with no required badges or identifiers such as hands and pencils (see Figure 2).

Figure 2 . The default part editor and part viewer icons

As a step toward becoming fully OpenDoc compliant (that is, becoming part editors themselves), some applications will be converted to *container applications* — applications that allow parts to be embedded in their documents, much as some documents today allow the embedding of QuickTime movies.

Many of today's applications have plug-in or extension APIs that may be used to add functionality to the application. These will continue to be important to extending the capabilities of part editors.•

PROPERTIES OF PARTS

All parts have a basic set of properties; these include the part kind, the part category, the view type (icon or frame, as we'll see in a moment), which editor to use, who last modified the contents of the part, and when the part was last modified. You may decide to support additional properties for parts that you develop — for example, whether to keep a paragraph of text with the next paragraph. Some part properties,

such as the view type, may be modified by users; other properties may be set only by developers or by the system.

Part kind and part category. Two critical part properties that you need to assign (in your part editor's 'nmap' resources) are part kind and part category.

- *Part kind* refers to the data format of a part's intrinsic content; it's analogous to a file's type. This property often has a name similar to the editor name. If the user changes a part's kind (with the Part Info command in the Edit menu), the part's content is translated to the new kind.
- *Part category* refers to a set of part kinds that are conceptually similar. OpenDoc uses categories to determine the set of part editors or viewers that are applicable to a given part, and to decide whether it's appropriate to translate data during inter-part editing (for example, when content is copied from one document into another). If a single part editor supports many kinds of data, these kinds are usually in the same category.

The list of categories is maintained by CI Labs, a consortium that coordinates cross-platform OpenDoc development; Styled Text and Video are two examples of part categories. If SurfWriter is a MacWrite-like text editor, its part kind might be SurfWriter Text and would be in the Styled Text category. The SurfWriter editor would most likely allow translation from other part kinds in the same category.

View type. Your part editor needs to assign the default view type for its embedded parts, which determines how each part is initially displayed: as an icon or in a frame.

- The icon for a part can be not only the standard 32-by-32- and 16-by-16-pixel sizes, but also a *thumbnail* icon (64-by-64 pixels). The thumbnail shows a miniature representation (a "poster page") of the part's contents to help users identify the part. Figure 3 shows the standard icons for a graphics part, and the thumbnail icon for a text part consisting of a one-page memo.

32-by-32 and 16-by-16 64-by-64 (thumbnail)

Figure 3. Icons for OpenDoc parts

- A part can be displayed with its contents in a bounded area called a *frame*, which allows editing in place (rather than requiring the part to be opened into a separate window). Frames are usually, but not necessarily, rectangular. A part's content may be displayed in more than one frame at a time and may have multiple representations; for example, a tabular part may be seen as a chart in one frame and as a text table in another.

In the Finder, documents are displayed only as 32-by-32- or 16-by-16-pixel icons in the initial OpenDoc release; eventually thumbnail icons and frames will also be supported at the Finder level.

Internally, all parts have frames, even when they're displayed as icons, but this implementation detail is hidden from the user and so is ignored in this discussion; here we use *frame* to mean only the view that displays the part's contents.•

Users can change the view type with the Part Info command in the Edit menu (and possibly with "accelerator" commands, such as View as Icon, that are provided by the part editor). In Figure 1 above, frame view is desirable because it allows the user to see the graphic laid out in the document and to edit it in place. An icon view might be preferable for, say, a spreadsheet part that gives supporting data on a subject covered broadly in the text. Any frame may be reduced to an icon at any time, or any icon opened into a frame, without affecting the view type of any other part; however, the containing part may reflow content when an embedded part's view type is changed.

Except that a part may be edited only when its content is viewed in a frame, icons and frames are functionally equivalent. Operations such as drag and drop that may be applied to one may be applied to the other. Whether viewed as icons or in frames, embedded parts can be opened into separate windows if desired (although they're still embedded parts and not documents).

WORKING WITH PARTS

Now that you know some of the basics about parts, let's look at what it's like to work with them. We'll start with what the desktop might look like after the user opens the document shown earlier in Figure 1 (see Figure 4). Document icons on the desktop look the same as today, even though some of them, like the Text Document icon, represent OpenDoc documents. From the user's point of view, there's no apparent distinction between an OpenDoc document and a "regular" application document. The menus are those of the text part editor (because the root part — a text part — is active). The menu names are those you'd expect when editing text, except that the File menu is named Document, and the Application menu icon (to the far right) is a document icon rather than an application icon. Finally, notice the Stationery folder; this contains the stationery pads that the user double-clicks or drags to create documents or parts.

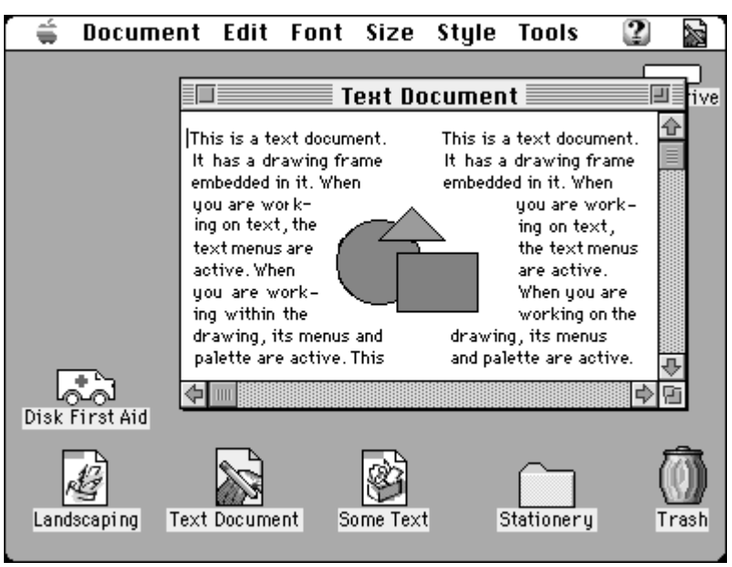


Figure 4. A Macintosh desktop in the world of OpenDoc

Now suppose the user wants to edit the content of this document's embedded graphics part. The first step would be to select the content to be edited, just as in applications today. To select the triangle, the user simply clicks it. As shown in Figure 5, a number of things happen: The graphics part editor highlights the selected graphics object by displaying handles. The graphics part becomes active (a border appears around its frame, its menus replace those of the text editor, and its tool palette appears). The text part is now inactive. Note that OpenDoc follows an "inside-out" model in determining which part to activate: it activates the smallest part that contains the mouse location.

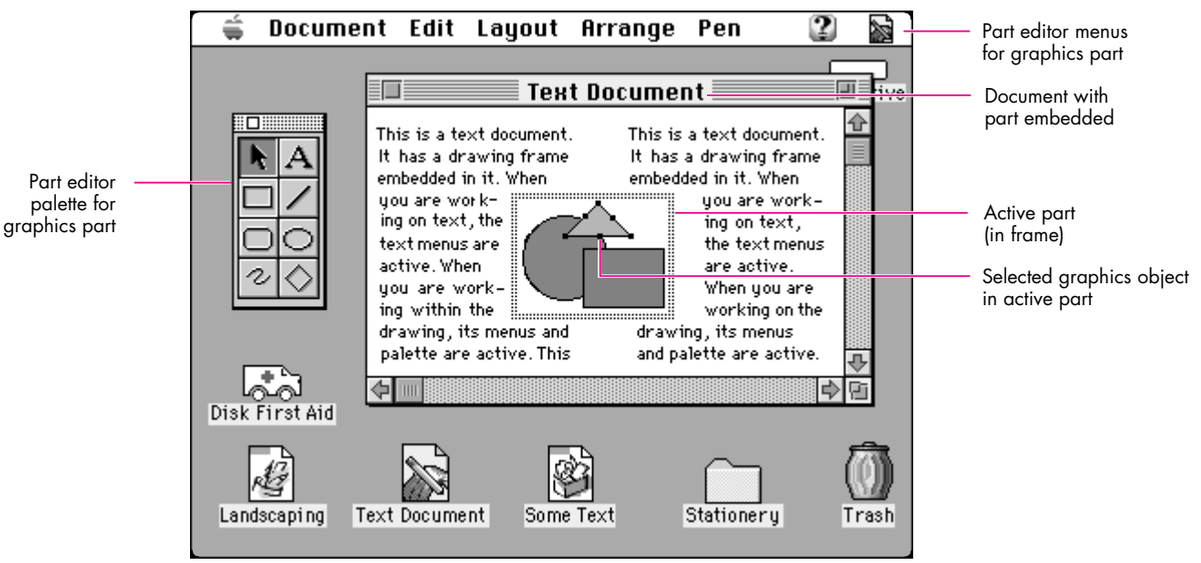


Figure 5. The same desktop after the user clicks the triangle

Just as the content of a part may be selected for editing, an embedded part (which is content of its containing part) may itself be selected for certain operations. To select an embedded part, the user drags across it or, if the part is active, clicks its border. Figure 6 shows what happens when the user clicks the active frame border in Figure 5: The graphics part is selected and its border changes to show handles; the part that contains it — the text part — becomes active again. The menus are replaced by the text menus and the graphics palette goes away. (The same thing happens to the menus and palette when the user selects text in the text part, or clicks there to get an insertion point; in all cases, content that resides in the text part has been selected.)

In summary, parts viewed in frames can be *active*, *inactive*, or *selected*. This state is indicated by the appearance (or absence) of a frame border. Parts viewed as icons can be only inactive or selected.

- A part is *active* when it contains the current selection or the insertion location (which could be a visible insertion point, as in text, or an unmarked default location, as when the background in a graphics part is clicked). The selection may be within the part's intrinsic content or it may be a part embedded in the active part. When an embedded part becomes active, OpenDoc displays the active frame border — a double dotted line — around the part.
- A part is *inactive* when the user is working in some other part. When viewed in a frame, an inactive part has no visible frame border around it. (Note that when a part is inactive, its part editor

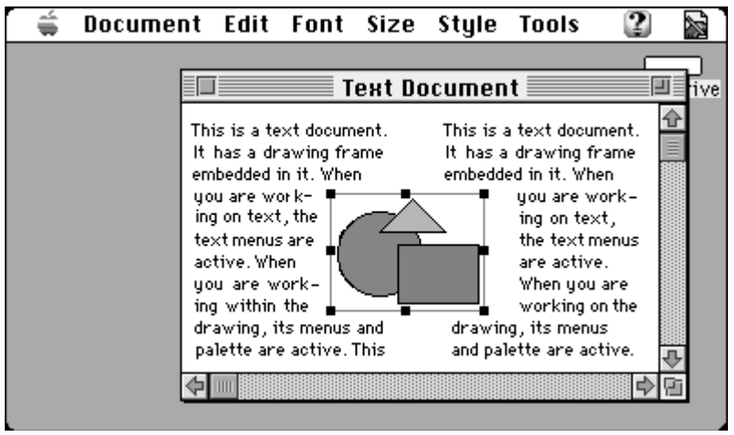


Figure 6. Selected embedded part

can still be running; for example, if there's a part that searches a database, the user can start the search and go off to do other work in another part in the same document while the search continues.)

- Finally, a part may be *selected*. The user selects a part in a frame view by dragging across it, or by clicking its border if it's the active part. The containing part is responsible for the visual appearance of a selected part's frame; typically, the frame shows handles, to allow resizing. To select a part viewed as an icon, the user simply clicks the icon.

In general, as soon as the user clicks inside a part's frame, that part becomes active. The editor for the previously active part removes its menus, palettes, and other user interface elements, and the new active part's editor displays its user interface elements. The active part receives commands and keyboard events. Only one part at a time may be active within a document because, as in today's documents, there can be only one selection at a time.

RESIZING FRAMES

From a user's point of view, resizing a frame is similar to resizing an on-screen object today. A difference in OpenDoc is that the same frame may show different numbers of resize handles on its border when it's in different containers, because the containing part's editor determines the appearance of a selected-frame border. Your part editor may display more or fewer resize handles than other editors — and perhaps none, if your editor doesn't allow the frame's size to be changed.

Your part editor also controls how much space an embedded frame occupies. When a user attempts to change the size of a frame embedded in your part, the embedded part negotiates with your part about the new size. Your editor may grant the requested size, reduce it, or refuse altogether, depending on its current contents and other part preferences such as snap-to-grid. The containing part also determines whether to adjust the layout of its own intrinsic content around the frame when a frame's size changes.

Some containing parts may require that embedded frames be rectangular, in which case their selection handles would resize only to rectangular areas. Others might allow embedded frames to be nonrectangular; for example, a containing part could provide selection handles that act independently, as shown in Figure 7.

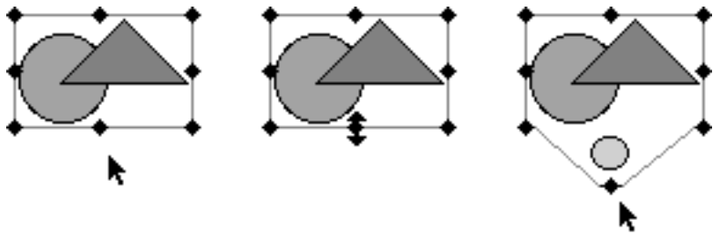


Figure 7. Independent resize handles

When a user changes the size of your part's frame, you should not stretch or scale the contents of the frame, but rather just change the viewing area. (Note how this differs from the resizing of content, such as a selected graphics object, in which case scaling may well occur.) Figure 8 shows a table part in its original state and after resizing to a smaller size; the viewing area has become smaller, but the content hasn't been scaled.

Although we recommend against scaling when a frame is resized, for some parts scaling may make sense.•

To see the entire table, the user can choose View in Window from the Edit menu. The table part then opens into a separate window (called a *part window*) allowing all its content to be seen, as shown in Figure 9. Although the part is viewed in a window, it's not a document — it's still an embedded part. Figures 8 and 9 show views of the same content, and any changes made in one are reflected in the other views.

COPYING AND MOVING CONTENT

The user can copy and move any content with the Cut, Copy, Paste, and Paste As commands as well as a variety of drag and drop operations. All these commands and operations work with embedded parts as well as intrinsic content, and they work

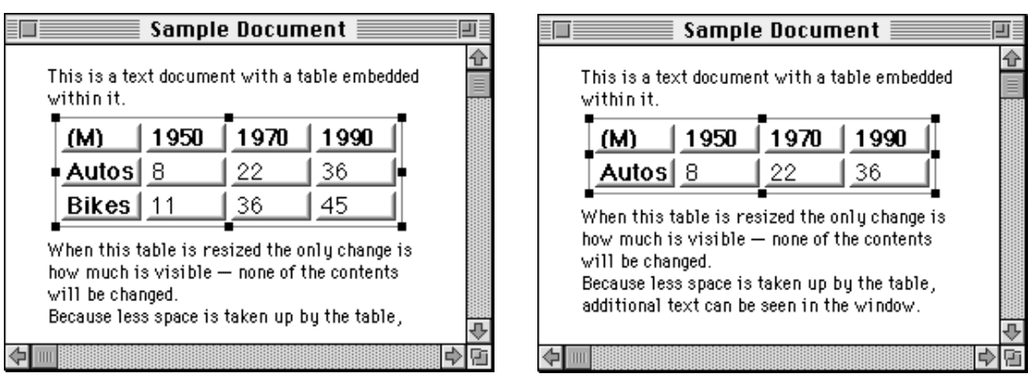


Figure 8. A table part before and after resizing

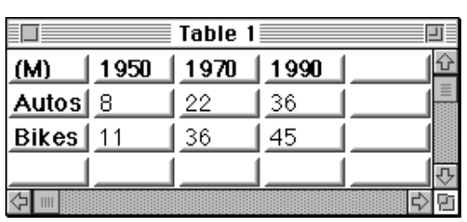


Figure 9. The table part opened into its own window

between parts in the same document as well as between different documents. Also, with drag and drop or the Insert command (in the Document menu), the source for a copy operation can be an entire document.

The Paste As command presents a dialog that allows the user to specify the data format to convert to when pasting. Holding down the Command key during drag and drop is the equivalent of Paste As; at the end of the drop, the Paste As dialog is presented.

Copied or moved content is inserted at the insertion location or replaces the current selection, with the exception that the Insert command inserts the contents of the document *after* the current selection.

The part editor makes "embed vs. merge" decisions in certain circumstances — that is, whether to insert the copied or moved content as an embedded part or to merge it with the destination part's intrinsic content. For example, a copied text part would be merged into another text part but would be inserted as an embedded part in a graphics part. Users will most often want the part editor to make these decisions, but they can always override them with the Paste As command.

Links. The Paste As command, or its drag and drop equivalent, also allows links to be created. Links are special cases of the copy operation: OpenDoc updates the copy when the original content changes (the user specifies in what situations updating should occur). Both intrinsic content and embedded parts can be linked, and links can exist within a single part, between two parts in a single document, or between multiple parts in different documents. The same content can be linked to multiple destinations, but each link is technically one-way; every link has a single source and a single destination. Typically, only edits to the source of the link are allowed. Some part editors may allow edits to the destination (such as a font change); however, these edits will not persist after the destination is updated from the source.

OPENDOC MENUS

OpenDoc provides these basic menus when a document is opened: Apple, Document (replacing the File menu), Edit, Help, and Application. The other menus vary depending on which part is currently active. As we saw earlier, when a part is activated, the associated part editor installs its menus and any tool palettes or in-window controls When a part (or the document window itself) is deactivated, the menus and palettes associated with the active part's editor are removed.

Note that when content is dragged into a frame, the part editor shouldn't install its menus unless the mouse button is released within the frame. For example, if the user selects some text in a text part and drags it into a graphics part, the initial text menus shouldn't be replaced until the user releases the mouse button within the graphics part.

Figure 10 shows what the basic menus might look like when a part whose editor is named SurfWriter is active. As you can see, the part editor has included its name in some of the commands. In the sections that follow, we'll look further at some of the commands in the Document and Edit menus.

DOCUMENT MENU COMMANDS

Most of the commands in the Document menu behave similarly to their File menu counterparts. Generally these commands refer to an entire document (the exception, Open Selection, is here because it's an open operation like Open Document). These commands should be augmented only if absolutely necessary, and only by the root part's editor.

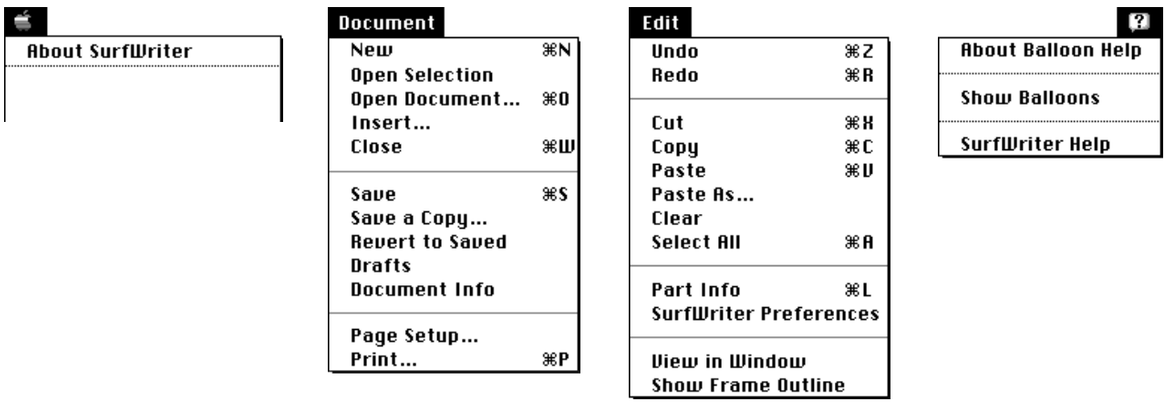


Figure 10. The basic OpenDoc menus

Notice that the Quit command isn't included, nor should it be added; OpenDoc editors are unloaded automatically when no longer needed. Also note that we're recommending support for Save a Copy rather than Save As; Save a Copy keeps the current document open and active, and the copy remains closed until the user opens it.

The Document menu includes these new commands: Open Selection, Open Document, Insert, and Drafts.

- Open Selection opens a selected part into a part window, allowing the user to, for example, look at the contents of a part viewed as an icon. (Double-clicking the icon would also work.) Note that this command applies to the currently selected part and not to the active part, which is the part *containing* the current selection.
- Open Document is analogous to Open File: it lets the user choose a document (through the Standard File dialog) and open it into a window, just as when the document is opened from the Finder.
- With the Insert command, the user can choose a document (again, through the Standard File dialog) to insert into the active part. It inserts the contents of the document at the insertion location or after the current selection. This command may be used with a stationery pad to embed a new "blank" part.
- The Drafts command allows the user to take a "snapshot" of the current state of the document at any time, creating a draft that can be accessed (or deleted) later through this command. Drafts are stored efficiently, as differences from the previous draft, so there's little penalty for using them. Previous drafts are "preserved" historical versions of the document; they aren't "live" and must be copied out to be edited.

EDIT MENU COMMANDS

The commands in the Edit menu are used to edit contents of the active part — for example, selected text or a selected embedded part — or to modify properties of a selected part. Because different part editors may require different editing commands, the active part editor may add additional commands to this menu.

Undo and Redo work as usual except that they can be invoked successively — that is, if the user chooses Undo three times in a row, the last three "undoable" actions are undone in order.

As described earlier, Cut, Copy, and Paste can be used to copy and move embedded parts as well as intrinsic content. The Paste As command lets the user specify the data format to convert to when pasting, and also includes an option for creating links.

The other commands of special interest in the Edit menu are Part Info, View in Window, and Show Frame Outline:

- Part Info displays a dialog for editing a selected part's properties — for example, changing the part kind to another (compatible) part kind or changing the view type from icon to frame or vice versa. This command is replaced by Link Info if the user selects a linked part. If intrinsic content is selected, the part editor may change this command — for example, to Paragraph Info or Circle Info.
- View in Window opens the active part into a part window. If the active part is already viewed in a window, this command brings that window to the front. (Remember that a *selected* part can be opened into a part window with the Open Selection command in the Document menu, or by double-clicking if it's an icon.)
- We recommend that you add Show Frame Outline to the Edit menu when a part in a frame view is opened into a part window. In the part window, this command puts an outline around the content that matches what's visible in the frame in the containing part, making it easier to correlate the two. The user can also drag the outline to change what's displayed in the frame in the containing part.

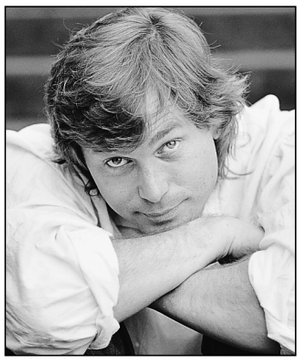
OPENING A WORLD OF POSSIBILITIES

OpenDoc revolutionizes the way developers deliver software, and does so without any dramatic upheavals to the user. OpenDoc part editors and container applications can coexist with applications and documents of today, providing OpenDoc's benefits without disrupting how users work with current applications. Your OpenDoc parts should behave much like current applications, so that users don't have to go through a substantial learning process. Our user tests show that users think OpenDoc simply fixes some "bugs" and lets them work the way they want. In other words, you and OpenDoc will provide business as usual — plus.

RELATED READING

- *OpenDoc Human Interface Guidelines, OpenDoc Programmer's Guide, OpenDoc Class Reference, OpenDoc Cookbook,* and *Drag and Drop Human Interface Guidelines.* These documents will eventually be available in printed form but meanwhile are provided electronically with early releases of OpenDoc.
- The OpenDoc World Wide Web pages, located at http://www.cilabs.org.
- *develop* articles by Kurt Piersol: "Building an OpenDoc Part Handler," Issue 19, and "Getting Started With OpenDoc Graphics," Issue 21. (The term *part handler* used in these articles has since changed to *part editor.*)
- *Apple Directions*: "OpenDoc Your Mind" in the December 1994 issue, and periodic articles (including Q&As) by the authors, starting with the January 1995 issue.

Thanks to our technical reviewers Dave Bice, Tantek Çelik, Ray Chiang, and Lori Kaplan, and to the OpenDoc Human Interface teams at Apple, Claris, IBM, and WordPerfect, especially Sue Bartalo, Kristin Bauersfeld, Dick Berry, Alex Bigney, Jennifer Chaffee, Pat Coleman, Dan Jordan, Jeff Kreeger, Per Nielsen, Kerry Ortega, David Roberts, David C. Smith, Mark Stern, Mike Thompson, and Ron Zeno. Special thanks to Dave Bice for providing source material for this article.



THE VETERAN NEOPHYTE

Paper Juggling

DAVE JOHNSON

I've been juggling seriously since the summer of 1979, when I saw a performance of the Pickle Family Circus in a park one gorgeous Saturday afternoon. I already knew how to juggle three balls — shakily — but that was the day I *really* discovered juggling. I had never seen clubs juggled up close and in person before (clubs are those bowling pin–like things that are thrown spinning end over end through the air), and in particular I had never seen jugglers throw things back and forth between each other (called *passing*). The Pickle Family did lots of both.

I was stunned. I was bowled over. I was frozen in my seat, gaping and incredulous. I couldn't believe that what I was seeing was possible. I *had* to learn how to do that.

Fortuitously, the circus offered workshops in various circus arts, including juggling, so I immediately signed up. The following morning, I learned the basics of passing balls, forced my roommate to learn to juggle three balls so that I'd have someone to try it out with, and embarked on a long and fruitful juggling binge. The fire that was lit that day burned white hot for over five years, and will remain fitfully smoldering as long as I can still lift my arms, close my fingers, and count to 3.

My favorite kind of juggling nowadays is getting together with other jugglers and passing clubs. We arrange ourselves in various formations about the floor, start juggling all together, and throw the juggling clubs back and forth in varied and complex — but mostly predetermined — ways.

Which brings me to the main topic of this column: how multiperson juggling patterns work, and one way to write them down on paper. I'm going dangerously far out on a limb here, assuming that it will be interesting to you, even though it has precious little to do with programming computers, and even though you're probably not a juggler. This particular limb is propped up a little by the very high proportion of computer people, mathematicians, engineers, and other scientists among jugglers. (There have been long-winded and unresolved discussions about why this should be so, but whatever the reason, it's a fact.) It's also been my observation (at Apple at our weekly juggle, and at the Worldwide Developers Conference) that computer people, in their endearing analytical way, often stand around for a long time trying to figure out the patterns.

Once you understand the rules of how the objects interleave and the jugglers interconnect, you can search for new patterns on paper, whether or not you know how to juggle. It's like a puzzle, or like a mathematical game. It's even conceivable (though just barely) that a knowledge of juggling patterns could be useful to you. I saw a citation on the rec.juggling newsgroup a while back for a paper called "Juggling Networks," published in the proceedings of a conference on parallel and distributed computing. From the abstract:

> *. . . these constructions are based on a metaphor involving teams of jugglers whose throwing, catching, and passing patterns result in intricate permutations of the balls. This metaphor affords a convenient visualization of time-division-multiplex activities that should be of value in devising networks for a variety of switching tasks.*

There have been several mathematical papers that deal with juggling in one way or another, and even so eminent a personage as Claude Shannon, the father of information theory, was an amateur juggler and was interested in the permutations and combinations in juggling patterns. He wrote a paper called "The Scientific Aspects of Juggling," and I heard that when he appeared at a juggling convention he drew thunderous applause from the assembled jugglers (another indication of how many jugglers are science types).

Club passing is by far my favorite kind of juggling. The jollies I get from it are all over the map; it's deeply satisfying for me on many, many levels. Part of it is social, of course. Like sex, it's just more fun with others. And a big part of it is the cooperation, being a part of this complicated group pattern that's built and

---

DAVE JOHNSON first met his wife, Lisa, in a stage combat class, learning to swashbuckle in dramatic fashion. Dave took fencing in college for a couple of years, always secretly wishing there were more yelling, ducking, slashing, and diving, instead of the tightly controlled, linear, minimalist motions of good foil fencing. Then he discovered the world of stage combat, and he's never gone back. He and Lisa are currently enrolled in a new class: Elizabethan Swordplay, using rapier and dagger. En garde!

maintained by everyone together. I suspect it's a lot like jamming with a band in that sense: we all agree on a framework — 12-bar blues in E or a seven-club four-count with triples, as the case may be — and then go for it, the members either struggling to keep up or embellishing wildly, according to their level of skill. Sometimes we'll hit a "groove," a day and a pattern and a distribution of people that just feels right, the beat solid, the hands sure of their grip.

Club passing can feel like being part of some giant, whirling, clockwork contraption, with everything ticking and clacking along. Talk about being a cog in the machine! The spinning clubs form this sort of living, writhing, flying tangle with its own weird existence, a kind of "energy net" connecting the jugglers involved. The old saw "what goes around comes around" has a particularly pointed truth in club passing: if one juggler throws a pass badly — say without quite enough spin, or a little off target — it causes the receiving juggler some, well, discomfort. That discomfort often manifests itself in another bad pass, causing the next receiver to struggle, and so on. It's often actually visible; you can see the disturbance making the rounds, until it either gets smoothed out by jugglers who manage to keep their cool, or amplifies itself so badly that the whole pattern comes crashing down around the jugglers' heads. (Interestingly, the disturbance often travels independently of the clubs themselves, in a different direction or at a different speed, like a wave passing through water.) And passing clubs fosters — *requires*, actually — a sort of heightened awareness of the other people involved. Often a quick, nearly imperceptible motion on the part of one juggler, a tiny hesitation, or the beginning of a wrong throw, corrected almost before it happens, causes another juggler to react reflexively. Typically both burst out laughing, mostly because it's unbelievable that such a tiny signal is transmitted at all.

And then there's the patterns game: a significant portion of the time spent "juggling" is really spent standing around, fiddling with the clubs, and trying to come up with new formations, new ways to arrange ourselves and the clubs in space and time so that everything fits together. The landscape of possible patterns is vast and complex, but also highly structured in mysterious ways. As in other iterative systems (computers and economies spring to mind), the underlying rules are relatively simple but the results can be very complex and widely variable. It's a kind of combinatorics and is, I think, actually covered by the mathematics of group theory.

I wrote a computer program that implements one particular kind of juggling notation, introduced to me by a juggler named Martin Frost and known as *causal diagramming*. This notation can be handy for doodling around trying to find new multiperson passing patterns. (Actually, I *started* writing the program. It's still rickety and unfinished, and will probably always remain so — it was more an experiment in QuickDraw GX programming than anything else. Nevertheless, it's included on this issue's CD, for your edification and/or derision.) The program implements a kind of active graph paper, allowing you to draw only "legal" throws, and constraining your diagrams in appropriate ways (such as preventing you from drawing throws that go back in time, for a start).

Figure 1 shows the diagram for a juggler doing a basic three-object pattern (called a *cascade*), and will serve to show both how the notation works and how juggling works. First the diagram: Time marches off inexorably to the right, divided into nice, even steps (called *counts*). A juggler is represented through time as a row of Ls and Rs, representing the juggler's left and right hands, alternately throwing things. A thrown object is represented by an arrow from the hand that throws it to the hand that catches it. The pattern wraps around at the dotted lines, and repeats endlessly — or until someone drops something. (The program always shows two repeating cycles like this, with the repeated parts "faded.") Note that the arrows (throws) form an unbroken line traveling through time from left to right, and that each hand has exactly one "input" and one "output."

Figure 1. A three-object juggle

Contrary to what you might think at first glance, the overall path the arrows make doesn't directly trace the path of an individual club. If it did, this would just be a diagram of throwing one club back and forth between two hands. (That's a necessary prerequisite to juggling, but is definitely *not* juggling.) Instead, each throw *displaces* a club that is always assumed to be held, waiting, in the receiving hand. Think of the juggler as holding a club in each hand, while the third is in the air. The incoming club displaces the club that's already there, forcing the juggler to throw it elsewhere. In a cascade, the displaced club is thrown back to the opposite hand, where it in turn displaces the club that's there, which goes back to the first hand, displacing the club that's there, and so on, ad infinitum. (Note that although I'm saying "club" here, all these principles apply equally well to balls or rings or rubber chickens.) So the chain

of throws is really a conceptual one, not a material one; it's a chain of cause and effect through time.

Figure 2 shows two jugglers passing with each other (the repeated cycle was cropped for space reasons). Note that they juggle in time with each other, like musicians keeping a beat. (When juggling with clubs, you actually *hear* the beat, when the clubs slap into the jugglers' hands.) Both jugglers throw a club to each other at the same time, both from the right hand (though it could just as well be the left). Throwing a club to another juggler "breaks" the juggler's continuous line of throws, but the other juggler's club arrives in the nick of time, knitting the pattern back together. This is a requirement: any club thrown to another juggler must be replaced by an incoming one. Otherwise, juggling can't continue; the juggler just stops, a club in each hand, waiting. (Actually, there are common situations that force a juggler to "stall" like that for a count or two, but we'll limit ourselves to the nonstalling patterns here.)

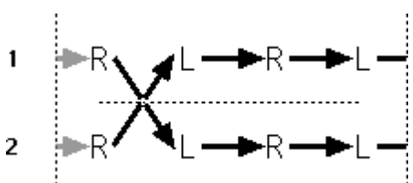


Figure 2. A four-count

Because of the close timing, both jugglers must agree on the pattern before starting. The pattern in Figure 2 is called a *four-count* because there's a pass every four counts. (Another name for this pattern is *every other*, referring to the fact that every other right-hand throw is a pass.) The four-count is a very common pattern, and for most club jugglers this is the default, "idling" pattern. Since there's so much time between passes, it's possible to do lots of fancy free-form tricks (affectionately known as "throwing trash") in the midst of the pattern. Of course, "so much time" isn't really much time at all: a club juggle is roughly 160 counts per minute, so there's just over a second between the passes in a four-count.

These diagrams show nothing about spatial relationships, by the way. The usual situation has the jugglers facing each other 6 or 8 feet apart, but the same patterns can be done standing side by side, back to back, or even with one juggler standing on the other's shoulders. These diagrams show only the "connectedness" of the pattern through time, and in fact you can draw patterns that work fine on paper but are difficult to actually do because of mid-air collisions.

Figure 3 shows another pattern that demonstrates some other important concepts. In this case, every right-hand throw is a pass (which makes this pattern a *two-count*). Although the jugglers are juggling to the same beat, note that they are out of sync; one juggler's right-hand throw is simultaneous with the other's left. Note also that each pass spends twice as long — two counts — in the air. In all the previous diagrams, the throws have been *singles*, meaning that the club spins around once during transit. The passes in Figure 3 are *doubles;* since they're in the air twice as long, they have time to spin around twice before being caught. (The left-hand "self" throws are still singles.)

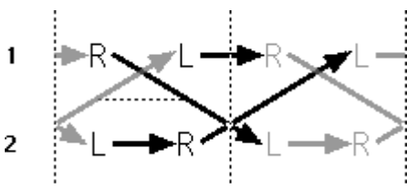


Figure 3. A two-count with right-handed doubles

A warning about these multiple-spin throws: It's tempting, on paper, to make heavy use of long arrows (throws that spend lots of time in the air between jugglers). A little physics tells you, though, that the time in the air is proportional to the height of the throw *squared*. So a double needs to be thrown four times the height of a single, and a triple must thrown nine times higher. A quadruple — a "quad" — must be *sixteen times* the height of a single, and that's about as far as you can reasonably go with any sort of accuracy (or safety!). I generally stop at triples.

Now take a look at Figure 4 (again, cropped for space). This shows a three-person pattern called a *feed*. In this case one person (juggler 2) acts as the *feeder* and the others are *feedees*. The feeder is passing twice as often as the feedees; the feeder is doing a two-count, while the feedees are each doing a four-count, interleaved with each other in time. The feeder switches back and forth

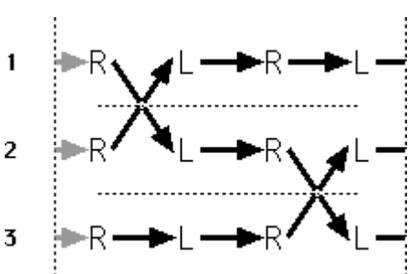


Figure 4. A feed

between the two feedees. This is another very common pattern, and can be added to indefinitely: Juggler 3 could pass with a new juggler, juggler 4, on the first count, at the same time jugglers 1 and 2 are exchanging clubs. That makes juggler 3 a feeder as well, feeding 2 and 4.

I think by now you can see how the patterns fit together. It's like building a network, where everything has to eventually connect up and balance out. Go ahead, give it a try. A favorite pattern of mine is a three-count, with a pass every third count; both left and right hands pass. How about a feed where the feedees do three-counts? How many three-count feedees can one feeder possibly handle? Try a ten-club feed (the feeder does two-count doubles, as in Figure 3, and the feedees each do four-count doubles). Admire the attractive and tidy braids that result. Go wild.

There are some interesting and nonobvious things about this notation that are probably worth pointing out. You can tell how many clubs there are in a pattern by taking a vertical slice through the diagram anywhere, counting the throws you intersect, and adding two clubs per juggler. (Note that Figure 3 is a seven-club pattern!) Also, if you start anywhere and follow the line of arrows around, wrapping back at the first dotted line, they always form closed paths, eventually arriving back where they began. Some patterns form one long continuous cycle; they're knit from a single strand, like a sweater. All the examples here are like that. Other patterns form distinct "orbits," where there are two or more strands making up the pattern; the three-count is an example. Each strand is an independent line of cause and effect, really an independent subpattern, that has no effect on the other parts of the pattern. You can actually decompose such patterns into their constituent parts, and juggle just one strand of the pattern at a time.

Also, the fate of any particular club isn't obvious at all in these diagrams. You can trace it, if you like — a club leaves a hand two counts after it arrives — but it's a bit of a pain (hmm, that might make a good addition to the program). Of course, tracing the paths of individual clubs isn't of primary interest to jugglers (though it's fun sometimes), in the same way that the path of an individual dollar is rarely of interest to economists and the trials and tribulations of an individual electron don't concern circuit designers. In contrast, I'd bet that the paths of the individual clubs are of *great* interest to the folks who wrote the network paper cited earlier. This notation would probably be a poor choice for them.

Finally, of course, the *experience* of juggling is nowhere to be found in these diagrams. In contrast to their clean, orderly lines, passing clubs is a very physical thing, full of grimacing effort, plagued with fumbling and mistakes, and occasionally bone-whackingly painful. It's more like chopping wood than like doing math; it's more like pounding nails than like tying macramé, despite the nice braided look of the diagrams. But when things get cooking, when everyone is warmed up and throwing well, when the pattern grows and takes shape between our hands and fills the air with intricate, swirling, impossible motion, there's nothing else quite like it in the world.

RECOMMENDED READING

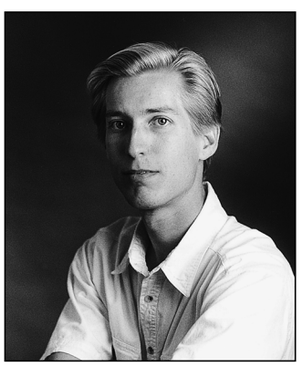
- "Scientific Aspects of Juggling," in *Claude Elwood Shannon Collected Papers* (IEEE Press, 1993).
- "The Academic Juggler," in *Juggler's World,* Vol. 45, No. 4 (Winter 1993–94). A discussion of the origins of juggling notations.
- The Juggling Information Service on the World Wide Web at http://www.hal.com/services/juggle/. You'll find juggling software, FAQs, archives of net discussions, movies, and lots more.
- *Operating Instructions, A Journal of My Son's First Year* by Anne Lamott (Ballantine Books, 1993).
- *June 29, 1999* by David Wiesner (Clarion Books, 1992).

Thanks to Lorraine Anderson, Jeff Barbose, Martin Frost, Bo3b Johnson, Lisa Jongewaard, and Ned van Alstyne for their enlightening review comments.•

Dave welcomes feedback on his musings. He can be reached at JOHNSON.DK on AppleLink, dkj@apple.com on the Internet, or 75300,715 on CompuServe.•

Futures: Don't Wait Forever

Futures provide a convenient way to implement asynchronous interapplication communication without having to manage unwieldy completion routines. This article presents an updated Futures Package that supports event timeouts, allows threaded execution of incoming Apple events, and has been revised to work with the Thread Manager in System 7.5.



GREG ANDERSON

Asynchronous Apple-event handling is difficult in Macintosh applications, and programmers who make the extra effort to implement it often find that detecting and recovering from event timeouts is an unmanageable task. Code that's written with the assumption that a completion routine will eventually be called will end up waiting forever if the event never completes. Futures provide a convenient way to support asynchronous interapplication communication and handle timeouts in a robust way, without sacrificing the simplicity or readability of the code.

Most applications attempt to manage multiple concurrent events through callbacks passed to AESend — but that leaves you, the application writer, with the burden of ensuring that the callbacks really do handle every event that's processed by the application's main event loop. For example, if you're writing an application that sends events to the Scriptable Finder, and you want to make that application scriptable itself, you'd have to be particularly careful not to lock up the user interface portion of your application every time an Apple event was received and processed. But by using threads, futures, and the asynchronous event-processing techniques described in this article, you can make the user-interface and event-processing modules of your application function independently — and almost without effort on your part.

If you're a long-time *develop* reader, you probably remember Michael Gough's article on futures that appeared in *develop* Issue 7. That article's information is still valid, and its code runs as well on today's Macintosh computers as it did when first published; however, it requires the Threads Package that came with Issue 6 in order to run. This article presents a revised version of the Futures Package, which works with the Thread Manager that's now part of System 7.5. We'll also delve a little deeper into the realm of asynchronous event processing and timeout event handling. And, for the curious, we'll open the black box and peer inside to examine the inner workings of futures.

For a review of threads and futures, see "Threads on the Macintosh" in *develop* Issue 6, "Threaded Communications With Futures" in Issue 7, and "Concurrent Programming With the Thread Manager" in Issue 17.•

GREG ANDERSON worked with Michael Gough on the original Futures Package that was described in Issue 7 of *develop*. One of Greg's favorite activities is ballroom dancing, which he does at every opportunity — particularly if he gets the chance to polka like a mad dog. Professionally, Greg is the technical lead of the Finder team at Apple.•

You can use the techniques described in this article with any application that uses Apple events, but they're particularly effective with scriptable applications that also send Apple events to other applications. You'll find the code for the new Futures Package on this issue's CD, along with the code for the FutureShock example, described later on, and preliminary documentation for the Thread Manager (eventually to be incorporated into *Inside Macintosh: Processes*).

For more on interactions with the Scriptable Finder, see "Scripting the Finder From Your Application," *develop* Issue 20.•

OVERVIEW OF FUTURES

For those of you who missed "Threaded Communications With Futures" in *develop* Issue 7, a *future* is a data object that looks and acts just like a real reply to some message, when in reality it's nothing more than a placeholder for a reply that the server application will deliver at some future time. (See "Client/Server Review" for a summary of how clients and servers interact.) Code written to use futures looks the same as code that waits for the reply to arrive (using a sendMode of kAEWaitReply) and then works with the actual data. The only difference is that the futures code uses a timeout value of 0. This causes AESend to return immediately to the caller with a timeout error — the normal and expected result — and execution of the client application is allowed to continue without delay.

The futures-savvy application then does as much processing as possible without accessing the reply, including sending other Apple events. When the data from the reply is absolutely needed, it's accessed as usual via AEGetKeyPtr or some other Apple Event Manager data-accessor function. It's at this point that the Futures Package steps in and suspends processing of the client application until the data for the reply arrives; other parts of the client keep running unhindered. Of course, it's not possible to stop one part of an application without stopping all of it, unless the application is multithreaded. Therefore, futures need to run with some sort of Thread Manager. Figure 1, which appeared originally in *develop* Issue 7, summarizes the roles of threads and futures and the interactions that take place when a client asks a question.

The primary benefit of the Thread Manager and Futures Package is that their use removes the burden of managing multiple concurrent events, whether they're Apple events or user actions. As mentioned earlier, most applications try to get around this problem by providing a callback procedure to AESend that can handle other incoming Apple events, update events, and user actions while the application is waiting for the reply. This technique works, but it's up to you to make sure the

CLIENT/SERVER REVIEW

In the vocabulary of interapplication communication, the *client* is the application that sends a message, and the *server* is the application that receives, processes, and responds to it. Since any application that processes events is a server, all scriptable applications are servers.

Some applications may take on the role of both client and server at different times. For instance, if an application needs to send an event to some other application in order to process the event that it just received, that application is the client of one application and the server of the other. It's also possible for an application to be a client of itself, if it sends itself messages; factored, recordable applications fall into this category.

Applications that act as both clients and servers should process events asynchronously — otherwise, the system can quickly become lost in a sea of woe and deadlock. But asynchronous event handling is complex and difficult; that's the problem that futures solve.

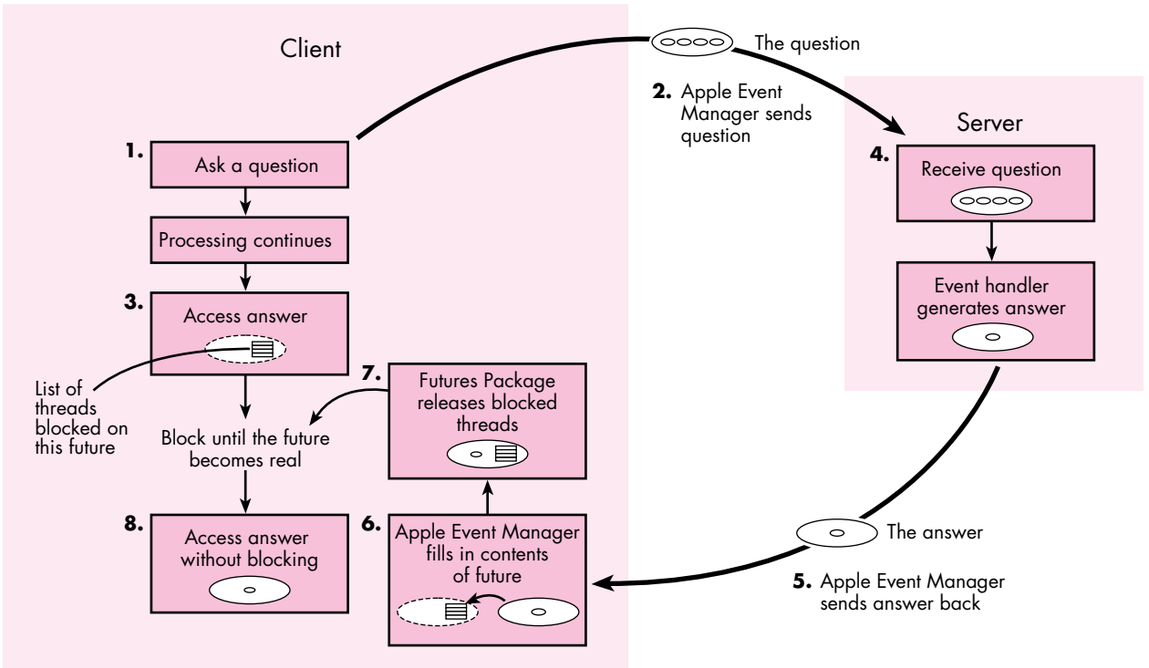


Figure 1. The transformation of a future into a real answer

callbacks handle everything. Listing 1 shows an example of how the callback approach works; notice that we need idle and filter procs to handle events that come in while the handler is waiting for a reply.

Responding to Apple events without using threads and futures is even more problematic, particularly if the application needs to send out another message in order to process the one that just came in (as in Listing 1). In that case, AESend is typically called again with the same callback procedure, and the whole process stacks up one level and repeats.

The problem with the stacked approach is threefold: First, the stack must unwind in the same order in which it was set up — an ill-timed incoming event, if it's a lengthy request, could interfere with the processing of the current outgoing request for quite a while. Second, every stack is finite in size; it's often difficult to prove that reentrant code will always have enough stack space to complete. Finally, writing callbacks and having multiple event loops in your application makes the source harder to follow, and what's more, it's a real drag. By contrast, futures allow the freedom of asynchronous operation without the drudgery of callbacks or completion routines — your code looks as simple as the normal synchronous version, but it runs asynchronously. The only difference from Listing 1 is that the code calls AskForFuture instead of AESend, as follows:

```
if (err == noErr)
   err = AskForFuture(&question, &answer, kAEDefaultTimeout, kNoMaximumWait,
      kNormalPriority);
```

One of the primary differences between the behavior of the code that calls AESend and the code that calls AskForFuture is that in the latter case, the event handler is already executing in its own thread when it's called. This is just one of the

Listing 1. An Apple event handler that sends an event

```
pascal OSErr AnAEHandler(AppleEvent* ae, AppleEvent* reply, long refCon)
{
   OSErr          err = noErr;
   AppleEvent     question, answer;
   AEAddressDesc  target;
   DescType       typeCode;
   long           actualSize, result;

   // Create an Apple event addressed to a previously determined target.
   // 'question' and 'answer' should be set to null descriptors.
   err = AECreateAppleEvent(kAnEventClass, kAnEventID, &gTarget,
      kAutoGenerateReturnID, kAnyTransactionID, &question);
   // Call AESend with the send mode kAEWaitReply. Note the idle and
   // filter procs.
   if (err == noErr)
      err = AESend(&question, &answer, kAEWaitReply, kNormalPriority,
         kAEDefaultTimeout, gAEIdleProcRD, gAEFilterProcRD);
   if (err == noErr)
      err = AEGetParamPtr(&answer, keyAEResult, typeLongInteger,
         &typeCode, (Ptr) &result, sizeof(long), &actualSize);
   if (err == noErr)
      err = AEPutParamPtr(reply, keyAEResult, typeLongInteger,
         (Ptr) &result);
   AEDisposeDesc(&question);
   AEDisposeDesc(&answer);

   return noErr;
}
```

conveniences offered by the new Futures Package, and it's a major enhancement; we'll describe how it works shortly.

OUR SAMPLE APPLICATION: FUTURESHOCK

This issue's CD contains a sample application called FutureShock that demonstrates the use of futures. You'll notice that there are two copies of this application on the CD, one sitting right next to the other. These copies are provided because FutureShock likes to talk to itself — well, not exactly to itself, but to other applications having the same process signature but a different process serial number. To use FutureShock, launch both copies of the application; you'll be presented with two instances of the same window. Clicking the button marked Send in one application window will send an Apple event to the other FutureShock application, which will acknowledge the receipt and begin "processing" the event.

Actually, no processing is being done — FutureShock is just whiling away the time looking at its watch (TickCount, that is), calling AEResetTimer every now and again, and calling YieldToAnyThread a lot. But don't tell the other FutureShock application that. It's busy keeping track of how long the message has been out for processing and how long it's been since its server last called AEResetTimer. If the server FutureShock is too slow, the client FutureShock will give up and cancel the message. (If you'd like to see this happen, use the set of radio buttons that allow you to inhibit the server from calling AEResetTimer.)

The sample source code included with the applications gives you a good example of how to use futures and keep track of message timeouts in a robust way. You'll also notice that FutureShock installs custom thread context–switching callbacks — a critical step for any application that uses threads (see "Custom Context Switching").

A LOOK AT THE SOURCE

The magic that makes Apple-event futures possible lies in the special blocking and unblocking callbacks supported by the Apple Event Manager. These callbacks aren't documented in *Inside Macintosh*, but they can be enabled with the function AEInstallSpecialHandler with the special keywords keyAEBlock ('blck') and keyAEUnblock ('unbk').

> **AEInstallSpecialHander is described** in *Inside Macintosh: Interapplication Communication,* page 4-100. •

If a blocking handler is installed, the Apple Event Manager calls it whenever an attempt is made to access data from an Apple-event reply that hasn't yet been received. Any Apple Event Manager function that extracts data, such as AEGetKeyPtr, causes the blocking routine to be called. The Apple Event Manager calls the unblocking routine as soon as the reply arrives. The blocking routine may be called many times for one reply (once for each call to AEGetKeyPtr or to another data accessor), but the unblocking routine will be called only once — whether it's needed or not.

The Futures Package makes use of the blocking and unblocking callbacks in a straightforward way. Whenever the blocking routine is called for a given Apple event reply, the reply's return ID is looked up via its keyReturnIDAttr attribute. The return ID is assigned by the Apple Event Manager whenever an event is sent. The Futures Package creates a semaphore and gives it an ID number that matches the return ID of the reply event so that the semaphore can be found again later. (For a review of semaphores, see "What's a Semaphore?")

CUSTOM CONTEXT SWITCHING

An application that uses threads must install custom thread context–switching callbacks if it has any global variables that need to have separate instances in every thread of execution. The most common reason for needing separate instances of a global variable is to maintain any global stacks in the application, such as the failure handler stack maintained by most exception handler packages.

A custom thread context–switching callback must be installed for every thread created by an application and also for the main thread (the thread created automatically by the Thread Manager). You can reference the main thread by using the constant kApplicationThreadID for its thread ID.

In the Metrowerks environment, an internally used global variable called _local_destructor_chain points to the top of a stack that keeps track of all the local variables that may need to have their destructor called (~TObject). If this variable isn't swapped out on a per-thread basis, one thread could cause the destructor for objects still active in another thread to be called out of context. The results, of course, would be disastrous (a crash). Compiler-specific global variables should be saved and restored within **#if** blocks, as is done in the following code (taken from FutureShock's swap-context-out callback):

```
#if _MWERKS_
   fLocalDestructorChain =
      _local_destructor_chain;
#endif
```

The same technique should also be used in the swap-context-in callback.

WHAT'S A SEMAPHORE?

A semaphore is an object that's used to arbitrate access to a limited resource or to somehow synchronize execution of independently operating processes. A semaphore controls the flow of execution in an application.

Threads of execution that "own" a semaphore are allowed to run, and threads that attempt to take ownership of a semaphore that isn't available are stopped and not allowed to run again until the semaphore becomes available. When used to arbitrate access to a limited resource, the semaphore also enforces strict sequencing of the threads that are blocked on it — ownership of the semaphore is provided to the threads that request it one at a time, in the order the requests are made. Typically, only one thread of execution is allowed to own a semaphore at a time.

With the Futures Package, when a thread attempts to access data from a future, a semaphore is used to synchronize its execution with the arrival of the reply. In this case, none of the threads owns the semaphore; conceptually, ownership lies with the future that the semaphore is associated with. When the future becomes a real reply, all of the threads blocked on the semaphore are allowed to run, and the semaphore is deleted.

The return ID is a long integer that's assigned sequentially when an event is created, and then copied into the reply event so that the Apple Event Manager can match the reply with the event that generated it.•

Once the semaphore has been created, the blocking routine gets a reference to the current thread, adds it to the semaphore, and puts the thread to sleep. The thread is now said to be *blocked on the semaphore*. If all goes well, the reply arrives shortly, and the Apple Event Manager calls the unblocking routine. Once again, the return ID is extracted from the reply event passed to the unblocking routine and is used to look up the semaphore created by the blocking routine. The unblocking routine then frees the semaphore, waking up all the threads that are blocked on it. Listing 2 shows the implementation of the blocking and unblocking routines in the Futures Package.

THE CLIENT SIDE — SENDING EVENTS

To use futures in your application, simply follow these guidelines:

- Use the Macintosh Thread Manager.
- Call InitFutures once when your application starts up to initialize the Futures Package. If your application has a custom thread scheduler, you'll probably want to provide a thread creation procedure. Alternatively, you can prevent the Futures Package from ever spawning threads, and keep track of housekeeping and asynchronicity issues on your own.
- Call AESend using the send mode kAEWaitReply, but specifying a timeout of zero ticks. Ignore the resulting error if it's errAETimeout. You may instead prefer to use AskForFuture, a convenient wrapper to AESend.
- Call AEGetKeyPtr and other standard Apple-event accessors to extract data from your replies. If the reply has not yet arrived, the current thread is blocked automatically. Make sure that the current function is running within a thread before accessing the data of the reply event; it wouldn't do any good at all to block the main thread.
- Call AEDisposeDesc to dispose of the event sent and the reply when done with them, just as with any other Apple event.

Listing 2. Blocking and unblocking routines

```
pascal OSErr AEBlock(AppleEvent* reply)
{
   TSemaphore*    semaphore = nil;
   OSErr          err = noErr;

   // It should always be possible to create and grab the semaphore.
   semaphore = GetFutureSemaphore(reply, kCreateSemaphoreIfNotFound);
   if (semaphore != nil)
      err = semaphore->Grab();
   else
      err = errAEReplyNotArrived;
   return err;
}

pascal OSErr AEUnblock(AppleEvent* reply)
{
   TSemaphore*    semaphore = nil;
   OSErr          err = noErr;

   semaphore = GetFutureSemaphore(reply, kDontCreateSemaphoreIfNotFound);
   if (semaphore != nil) {
      semaphore->ReleaseAllThreads();
      semaphore->Dispose();
   }
   return err;
}
```

As you can see, there's almost nothing special you need to do in order to use futures — your code will look almost exactly the same as similar code that doesn't use futures at all.

THE SERVER SIDE — RESPONDING TO EVENTS

Futures provide a convenient way to send messages and receive replies asynchronously, but it's just as important for the server application to *process* events asynchronously. There are a number of techniques for creating threads to process incoming events, but the most convenient thing to do would be to spawn a thread before calling AEProcessAppleEvent and allow the Apple Event Manager to dispatch the event from within the cozy, asynchronous environment of its own thread. Unfortunately, AEProcessAppleEvent is not reentrant; if you call it from a thread, your application will crash if another event is received before the current one finishes processing — which rather defeats the whole purpose of asynchronous processing, to put it mildly. Fortunately, there's a convenient workaround for this problem.

The solution is to install a predispatch handler that intercepts all events being dispatched by AEProcessAppleEvent and makes sure that the event is suspended and that the handler exits right away. The predispatch handler also forks a new thread that manually dispatches the event when the thread is next scheduled. When the event handler returns, this thread calls AEResumeTheCurrentEvent to force the Apple Event Manager to send the reply back to the client. Listing 3 shows how this is done.

Listing 3. Spawning a new thread before dispatching the event

```
#define kUseDefaultStackSize 0
pascal OSErr Predispatch(AppleEvent* ae, AppleEvent* reply, long refCon)
{
   OSErr               err = errAEEventNotHandled;
   PredispatchParms**  dispatchParams = nil;
   AEEventHandlerUPP   handler = nil;
   long                handlerRefCon = 0;

   if (GetAppleEventHandlerUPP(ae, &handler, &handlerRefCon) == noErr) {
      dispatchParams = (PredispatchParms**)NewHandle(
            sizeof(PredispatchParms));
      if (dispatchParams != nil) {
         ThreadID newThreadID;
         (*dispatchParams)->fAppleEvent = *ae;
         (*dispatchParams)->fReply = *reply;
         (*dispatchParams)->fEventHandler = handler;
         (*dispatchParams)->fHandlerRefCon = handlerRefCon;
         if (NewThread(kCooperativeThread,
               (ThreadEntryProcPtr)RedispatchEvent,
               (void*)dispatchParams, kUseDefaultStackSize,
               kCreateIfNeeded | kFPUNotNeeded, nil, &newThreadID)
               == noErr) {
            dispatchParams = nil;
            // Suspend the current event so that the Apple Event Manager
            // won't break. Set the error to noErr to tell the Apple
            // Event Manager we handled the event.
            AESuspendTheCurrentEvent(ae);
            err = noErr;
         }
      }
   }
   // Dispose of the dispatch parameters if created but not used.
   if (dispatchParams != nil)
      DisposeHandle((Handle)dispatchParams);
   return err;
}

void RedispatchEvent(void* threadParam)
{
   OSErr    err = noErr;

   PredispatchParms** dispatchParams = (PredispatchParms**)threadParam;
   AppleEvent ae = (*dispatchParams)->fAppleEvent;
   AppleEvent reply = (*dispatchParams)->fReply;
   AEEventHandlerUPP handler = (*dispatchParams)->fEventHandler;
   long handlerRefCon = (*dispatchParams)->fHandlerRefCon;
   DisposeHandle((Handle)dispatchParams);
   // Call the event handler directly.
   err = CallAEEventHandlerProc(handler, &ae, &reply, handlerRefCon);
   if (err != noErr) {
      DescType    actualType = typeNull;
```

(continued on next page)

Listing 3. Spawning a new thread before dispatching the event *(continued)*

```
      long        actualSize = 0;
      long        errorResult;

      // If the event handler returned an error, but the reply does not
      // contain the parameter keyErrorNumber, put the error result into
      // the reply.
      if (AEGetParamPtr(&reply, keyErrorNumber, typeLongInteger,
            &actualType, &errorResult, sizeof(long), &actualSize)
            != noErr) {
         errorResult = err;
         AEPutParamPtr(&reply, keyErrorNumber, typeLongInteger,
            &errorResult, sizeof(long));
      }
   }
   // Tell the Apple Event Manager to send the reply.
   AEResumeTheCurrentEvent(&ae, &reply,
      (AEEventHandlerUPP)kAENoDispatch, 0);
}
```

The beauty of the technique shown in Listing 3 is that it's nicely isolated from the rest of the code. The application's main event loop still calls AEProcessAppleEvent as usual, and event handlers are installed and dispatched as usual. The only difference is that now, event handlers are processed in their own thread of execution and may call YieldToAnyThread to allow other parts of the application to run. The Futures Package installs this predispatch handler when it's initialized; Listing 4 shows an example of an event handler similar to the one in the FutureShock application.

Listing 4. Simple threaded event handler

```
pascal OSErr TestEvent(AppleEvent* ae, AppleEvent* reply, long refCon)
{
   OSErr    err = noErr;

   while (WorkLeftToDo() && (err == noErr)) {
      YieldToAnyThread();
      if (gHasIdleUpdate)
         IdleUpdate();
      err = DoSomeWork();
   }
   return err;
}
```

Note the call to IdleUpdate; on PowerBooks, if the operating system thinks that the system isn't doing anything important, it will slow down the processor to conserve power. This happens after 15 seconds during which no user activity and no I/O occurs. In the realm of threads and interapplication communication, it's easy for 15 seconds to go by with no such activity, even if the machine is actually busy processing an event. Calling the Power Manager procedure IdleUpdate avoids the power-saving mode, and any application that performs lengthy operations should

do this. Be sure, though, to check the gestaltPMgrCPUIdle bit of the Gestalt selector gestaltPowerMgrAttr before calling IdleUpdate, because most desktop machines don't implement this trap.

IdleUpdate is described in *Inside Macintosh: Devices,* page 6-29.•

Another mechanism for spawning a thread besides using the predispatch handler is to use Steve Sisak's AEThreads library; see "The AEThreads Library" for more information.

CLIENT/SERVER TIMEOUT NEGOTIATIONS

The Apple Event Manager provides a function called AEResetTimer that lets servers inform their clients that work is being done on the event but that the reply is not yet available. AEResetTimer is of value only to clients that use the send mode kAEWaitReply — the intention was for clients to use a fairly short timeout value and for servers to periodically inform the clients of progress so that the call to AESend isn't aborted unless the server actually can't be reached (or crashes). The mechanism involves the Apple Event Manager sending a "wait longer" event back to the client, tagged with the return ID of the Apple-event reply. The "wait longer" event is intercepted by a filter inside AESend that's supposed to reset the event's timer; unfortunately, a bug in the Apple Event Manager prevents the "wait longer" event from working correctly, and the timer is not reset.

The existence of this bug shouldn't deter you from calling AEResetTimer in your server application, though. The bug exists in the code that runs on the client side of the communication, and some future version of the Apple Event Manager will fix it. Also, as you'll see shortly, the Futures Package hooks into the "wait longer" event and uses it to prevent blocked messages from timing out if the server application uses AEResetTimer to request more time, effectively bypassing the bug. Other applications that don't use the Futures Package could use a similar technique to detect server activity — thus, AEResetTimer is the correct protocol for the server, whether the client application uses AESend with kAEWaitReply or the Futures Package.

THE AETHREADS LIBRARY
BY STEVE SISAK

In my article entitled "Adding Threads to Sprocket" in the December 1994 issue of *MacTech Magazine*, I described an implementation of futures and a library called AEThreads that allows you to install asynchronous Apple event handlers. I stated that the futures code should really be supported by Apple and that you should go with their solution if they eventually provide one. This is the case here. The Futures Package addresses many of the issues that my library did not, and is also provided in source form. Therefore I recommend that you use Greg's futures implementation instead of mine in any new code.

You may, however, find AEThreads more useful for spawning threads than the predispatch handler in the Futures Package. Its main advantage is that it allows you to control, on an individual basis, which events are handled asynchronously and which are handled immediately. It also enables you to control all of the thread parameters (for example, stack size and needFPU) for your event-handling threads, and it doesn't interfere with installing a predispatch handler in your application (as described in *Inside Macintosh: Interapplication Communication* on pages 10-19 to 10-21).

The AEThreads library is provided on this issue's CD. To use it, don't install the predispatch handler when you initialize the Futures Package (as explained in the description of InitFutures later in this article), and call AEInstallThreadedEventHandler where you would have called AEInstallEventHandler. Everything else should work the same. If you have any questions, comments, or problems with AEThreads, please let me know at sgs@gnu.ai.mit.edu.

TIME AFTER TIME

How often should the server call AEResetTimer? Calling it too frequently is a bad idea, because an event is generated on every call to reset the timer. Some existing applications call AEResetTimer when half the message's timeout value has expired; the timeout value can be determined by examining the attribute keyTimeoutAttr in the Apple event that the server receives. The problem with this technique is that futures, as you may remember, are always sent with a timeout value of 0. Naive servers that always depend on keyTimeoutAttr to be a meaningful value will call AEResetTimer *much* too frequently.

At the very least, servers should define a threshold, perhaps 150 ticks, and never call AEResetTimer more frequently than that. The recommended solution, however, is first to check for the presence of the attribute keyAEResetTimerFrequency. If it exists, it indicates approximately how often, in ticks, the client would like the server to call AEResetTimer. If this attribute doesn't exist, the server should fall back on the default method of using the larger of half the value of keyTimeoutAttr or 150 ticks. This technique provides the greatest flexibility for clients, while allowing the server application to continue to perform reasonably well even with clients that don't provide specific timeout information in the events they send.

It's the responsibility of the client to pick timeout and reset frequency values that allow the server enough time to respond but still provide adequate response time to the user when the server actually isn't available. The client should take into account that the transit time for the event will vary, depending on whether the event is being sent to a local or a remote process and on the network conditions at the time the event is sent. Finally, when choosing timeout values, remember that almost no background processing is done on the Macintosh as long as the user is doing something with the mouse button down (such as browsing menus or dragging windows or Finder items). A client that picks too small a value for its timeout is in danger of having user actions interfere with the server's processing of its events, which could quite easily cause the client's events to time out unnecessarily.

HOW FUTURES DEAL WITH TIMEOUTS

The Futures Package keeps track of timeouts whenever a thread is blocked while accessing data from a reply that hasn't arrived yet. The client must specify the timeout value to use with the SetReplyTimeoutValue function, which must be called after the message is sent but before the reply is accessed. The AskForFuture function follows this protocol when it calls SetReplyTimeoutValue, so your application doesn't need to call SetReplyTimeoutValue if it calls AskForFuture. When this timeout value is set, the Futures Package creates a semaphore and stores the timeout values inside it. This same semaphore is used to block any thread that attempts to access data from the reply before it arrives. If an event times out, the semaphore wakes up all threads that are blocked on it and returns a timeout error to the future's blocking routine. The error is passed to the Apple Event Manager, which will return errAEReplyNotArrived to the accessor(s) that caused the thread to be blocked.

Both SetReplyTimeoutValue and AskForFuture take two parameters: a timeout value and a maximum wait value, both expressed in ticks. The timeout value indicates how many ticks the client is willing to wait before it hears anything from the server; if the server calls AEResetTimer, the client resets its timer and begins waiting again. But if the timeout value is the only control that a client has, a berserk-server-from-hell that does nothing but call AEResetTimer for days on end and never returns any results could keep the hapless client locked up forever. This is where the maximum wait value comes in: if the client specifies a maximum wait time, any event that remains unserviced for longer than this period of time immediately terminates, even if the server called AEResetTimer only a couple of ticks ago.

Usually, it's best for clients to assume that servers are well behaved, and that they will eventually return results as long as they're still working on the problem. Distributed computing applications, though, might find it better to reschedule some lagging events on a faster machine if the server initially selected doesn't respond quickly enough. The maximum wait value gives them the control they need to do so. If either the timeout value or the maximum wait time expires, the Futures Package automatically wakes up all threads blocked on that future. The error code returned by the Apple Event Manager is errAEReplyNotArrived, which is the same result that would be returned if a reply that had timed out from AESend was accessed without using the Futures Package.

Note that the Apple Event Manager doesn't assume that an application has given up on a reply until the reply is disposed of. Until that happens, the reply will be filled in as soon as it's received, even if the event has timed out. A distributed computing application that rescheduled an event on a faster machine could keep a reference to the old future around and use the result from the machine that finished first.

THE FUTURES PACKAGE API

Here's a description of the routines provided by the updated Futures Package.

```
void InitFutures(ThreadCreateUPP threadCreateProc, long initFuturesFlags)
```

The function InitFutures initializes and enables the Futures Package. The parameter initFuturesFlags should be set to the sum of the flags that the futures-savvy application wants to set. The Futures Package recognizes two flags: the first, kInstallHouseKeepingThread, causes the Futures Package to create a new thread that does nothing but call IdleFutures (described below); the other parameter, kInstallPredispatch, specifies that the Futures Package should install the predispatch handler shown earlier in Listing 4. This handler causes a new thread to be created for every Apple event dispatched by AEProcessAppleEvent. The threadCreateProc parameter to InitFutures is for applications that install custom context-switching routines or that maintain a custom thread scheduler. This thread creation procedure is called every time the Futures Package creates a new thread, allowing your application to hook the new thread into its scheduler and install custom context-switching routines.

The thread creation procedure is defined like this:

```
pascal OSErr MyThreadCreateHandler(ThreadEntryProcPtr threadEntry,
   void* threadParam, long handlerRefCon, ThreadID* threadMade)
```

The threadEntry, threadParam, and threadMade parameters should be passed on to NewThread. The handlerRefCon parameter is the refCon that was passed to AEInstallEventHandler when the event handler for the Apple event being dispatched was installed. InitFutures will also call the thread creation procedure to create the housekeeping thread; in that case, the refCon passed in will be 0. If a thread creation procedure isn't provided, the Futures Package will call NewThread directly.

```
void BlockUntilReal(AppleEvent* reply)
```

The function BlockUntilReal causes the current thread of execution to be blocked until the specified Apple event reply becomes a real message. Usually, this routine doesn't need to be called; the Futures Package automatically blocks the current thread whenever any Apple Event Manager accessor function is called to get data out of a future.

```
Boolean ReplyArrived(AppleEvent* reply)
```

The function ReplyArrived returns true if the given reply has been received, in which case it may be accessed without blocking. Usually, this routine won't need to be called. The whole idea of the Futures Package is to remove the burden of keeping track of whether a reply has arrived. ReplyArrived has a counterpart function named IsFuture, which is provided for compatibility with the Futures Package API presented in Issue 7 of *develop*.

```
void SetReplyTimeoutValue(AppleEvent* reply, long timeout,
   long maxWaitTime)
```

SetReplyTimeoutValue allows the client to specify a timeout value and an upper bound on the amount of time it's willing to wait before a thread that's blocked on a future should be awakened and informed that the event timed out. If used, SetReplyTimeoutValue must be called after the event is sent, but before the reply is accessed in any way. Usually, SetReplyTimeoutValue won't need to be called directly, because it's called by the function AskForFuture (described below).

```
void IdleFutures()
```

The IdleFutures function does the actual test to see whether any of the blocked messages have timed out. Usually, IdleFutures is called automatically by the Futures Package; if your application doesn't specify the flag kInstallHouseKeepingThread in InitFutures, however, it should call IdleFutures periodically. It's not necessary to call IdleFutures more frequently than every tick or so, but the function is smart enough not to do work superfluously, so there shouldn't be a negative performance hit to calling IdleFutures more frequently than once a tick. Don't go overboard, though — enough is enough.

```
OSErr AskForFuture(const AppleEvent* ae, AppleEvent* future, long timeout,
   long maxWaitTime, AESendMode sendMode, AEPriority priority)
```

The AskForFuture function calls AESend following the protocol defined by the Futures Package; keyAEResetTimerFrequency is set before the event is set, and SetReplyTimeoutValue is called with the specified timeout and maximum wait times. AskForFuture will always return immediately; the reply received will be a future, and timeout processing will be done correctly if the current thread of execution blocks on the future.

```
long GetResetTimerFrequency(const AppleEvent* ae)
```

The GetResetTimerFrequency function returns the frequency, in ticks, with which the Futures Package thinks that your application should call the Apple Event Manager function AEResetTimer, based on parameters in the provided Apple event. Note that GetResetTimerFrequency should be passed the Apple-event message; this is different from the Apple Event Manager routine AEResetTimer, which needs the Apple-event reply.

```
OSErr ResetTimerIfNecessary(AppleEvent* reply, unsigned long& lastReset,
   long resetFrequency)
```

ResetTimerIfNecessary calls AEResetTimer when enough time has elapsed since the last time it was called. The server is responsible for keeping track of the reset frequency and storing away the last reset tick, although the Futures Package will do the housekeeping of updating the last reset tick whenever AEResetTimer is actually called.

FUTURE DIRECTIONS

Apple events allow ordinary applications to become powerful tools for use both in scripting and by other applications; however, the power afforded by Apple events can be quickly negated if the server can't process multiple events asynchronously, or if the user can't work with the client process while it's waiting for a reply. As more applications become scriptable, and as component-oriented systems such as OpenDoc become more prevalent, the distinction between client and server becomes blurred, and more applications will take on both roles. In a world where asynchronous interapplication communication is the norm rather than the exception, the Futures Package allows you to harness the power of asynchronicity without becoming lost in a mire of completion routines.

RELATED READING

- *Inside Macintosh: Interapplication Communication* (Addison-Wesley, 1993).
- *develop* articles: "Threads on the Macintosh" by Michael Gough, Issue 6; "Threaded Communications With Futures" by Michael Gough, Issue 7; "Concurrent Programming With the Thread Manager" by Eric Anderson and Brad Post, Issue 17; "Scripting the Finder From Your Application" by Greg Anderson, Issue 20.
- "Adding Threads to Sprocket" by Steve Sisak, *MacTech Magazine,* December 1994.

Thanks to our technical reviewers Eric Anderson, Michael Gough, Ed Lai, and Steve Sisak. Special thanks to Ed Lai, who put futures support into the Apple Event Manager.

Macintosh Q & A

Q *I'm using the TPopup class in MacApp 3.0.1 in my window and I want to underline the title string of a pop-up menu programmatically. The title text style is stored in a field in the class but is used only when the pop-up menu is first created. How can I change the text style of a pop-up menu title after it has been created?*

A MacApp's TPopup class is basically just a wrapper around System 7's Popup CDEF (with its own CDEF for pre-System 7) and so is subject to the same limitations as normal System 7 pop-up menus. You're correct that the title style is stored in the TPopup class and referenced only once, when the pop-up control is created. What happens is that when a pop-up menu is created with NewControl, the Popup CDEF interprets the value parameter to NewControl to be the title style of the pop-up menu control. Thereafter, the value of the control is equal to the currently selected menu item. TPopup::CreateCMgrControl calls NewControl as follows:

```
ControlHandle aCMgrControl = NewControl(itsPort, qdArea, itsTitle,
      FALSE, (short) this->GetPopupTitleStyle(), fMenuID, fItemOffset,
      this->GetProcID(), fUseAddResMenuResType);
```

The important setting is the title style: notice the TPopup::GetPopupTitleStyle call, which returns a short integer corresponding to the text style settings. The problem is that there's no way of defining this title style after the control has been created, so you have to recreate the control when you want to change the title style. This may seem a bit much, but it takes only a few lines of code. The important thing to remember is that most of the information you need is already part of TPopup; all you're doing is recreating the control.

Dispose of the old control, set the fTitleStyle field to the title style you want, and then call CreateCMgrControl to create a new control with this title style, using all the characteristics already set in your TPopup object. Here's the code to do this:

```
CStr255  itsLabel;
short    itsVal;

/* First free the old control. */
DisposeControl(myPopup->fCMgrControl);
myPopup->fCMgrControl = NULL;
/* Now set the pop-up title style to underline. */
myPopup->fTitleStyle = myPopup->fTitleStyle + underline;
/* Get title and current value to send to CreateCMgrControl. */
myPopup->GetMenuLabel(itsLabel);
itsVal = myPopUp->GetCurrentItem();
/* Now create a new control with the desired text style. */
myPopup->CreateCMgrControl(itsLabel, itsVal, 0, 0, 0);
```

Q *I'm having a problem with Balloon Help, getting HMCompareItem to work properly. I've got several menu items that can change dynamically, and while HMCompareItem successfully finds the first item, all other items have no balloons. What's the problem here?*

A The problem is that the match string isn't exact. HMCompareItem only finds exact matches for the actual menu items. (A common case to look out for is ellipsis (…) versus three periods (...): always use an ellipsis in menus.)

If you can't determine the exact match ahead of time, we suggest that you use a different technique: modify the help string on the fly. A method that other developers have used is to store the current menu state in their preferences file along with the current menu help string; then, as the application changes menu items, they modify the 'STR ' resource that the help item refers to on the fly.

Q *I'm writing a QuickDraw GX printer driver and need to get the text size of a shape. I've tried GXGetStylePenSize and GXGetShapePenSize, but these continuously send back 12 no matter what the real size is. I've looked through the shapes in GraphicsBug, and 12 is there for the text size. What can I do to get the correct size?*

A QuickDraw GX has three different shapes to handle typography — text, glyph, and layout — and each one stores the typographic style objects (which are what you need) differently. (There's a good discussion of the three types of typographic shapes in *QuickDraw GX: Programmer's Overview* on pages 97 through 115.)

The important thing to remember is that simple text shapes can have only one type style (attached to the style attribute of the object), so they're fairly easy to work with. However, glyph and layout shapes can have one or more runs of type styles (attached to the style list attribute of the object's geometry), so they can be more complex to work with. Only if a glyph or layout shape doesn't have a style list attached to its geometry is the style attribute of the object itself used. For shapes with multiple style runs, there's no simple answer to the question "What is the text size of this object?"

For glyph and layout shapes, you'll need to write a "GetSizes" function that's capable of returning one or more sizes. This routine should get the style list by calling GXGetGlyphShapeParts or GXGetLayoutShapeParts. If the style list is nil, return the default size in the style attribute of the object itself; otherwise, return an array of each size in the list of styles, or whatever is appropriate for your application.

Q *I'm writing a QuickDraw GX application, and the glyphs that are drawn on the screen sometimes don't match the character codes in the shape. Any idea what's going on?*

A It's very important to pass the correct script system, language, and platform when your application creates a layout shape or a style used within a layout shape. The following code fragment will do the trick:

```
long  script;
long  language;

// Set myStyle's encoding correctly for this machine.
if ((script = GetEnvirons(smKeyScript)) != 0 &&
    (language = GetScript(script, smScriptLang)) != 0)
   GXSetStyleEncoding(myStyle, gxMacintoshPlatform,
      (gxFontScript) (script + 1), (gxFontLanguage) (language + 1));
else
   GXSetStyleEncoding(myLayoutStyle, gxMacintoshPlatform,
      gxRomanScript, gxNoLanguage);
```

In the case of a shape, the code is similar but calls GXSetShapeEncoding instead of GXSetStyleEncoding. Note the "(script + 1)" and "(language + 1)":

this synchronizes the information returned by the Script Manager with QuickDraw GX's representation of the same data.

Q *When displaying JPEG-compressed PICTs with DrawPicture, I call QDError and get the error code -8976, which isn't documented anywhere I've looked. What's going on?*

A This is the codecNothingToBlitErr error. It means that the picture was drawn into an entirely clipped-out bitmap. You can safely ignore this error. This is fixed in Apple's Multimedia Tuner (and will be fixed in future versions of QuickTime), so that the error won't be reported. Better, of course, would be to avoid drawing into clipped-out bitmaps at all.

Q *I'd like to add the capability to turn the Macintosh on and off automatically in my application. Is there an API for scheduled startup and shutdown?*

A Yes and no: there's an API for auto-startup, but not for timed shutdown. The auto-startup feature is built into the Time Manager. The Power Manager features flag has been updated so that it can easily be tested. Timed shutdown will have to be done manually: we recommend creating a background-only application that simply waits for the appropriate time and then issues the Finder event to shut the system down.

Auto-startup can be used if the PMFeatures routine returns a long word with bit 10 set. The name of the enum in the new headers is hasStartupTimer. If this flag is present, these routines are also supported:

```
void SetStartupTimer(StartupTime *theTime);
OSErr GetStartupTimer(StartupTime *theTime);
```

SetStartupTimer sets the time that the Macintosh will start up from a power-off state, and enables or disables the startup timer. On a Macintosh that doesn't support the startup timer, SetStartupTimer does nothing. The time and enable flags are passed in the following structure:

```
typedef struct WakeupTime WakeupTime, StartupTime;
struct WakeupTime {
  unsigned long wakeTime; /* startup time (same format as current time) */
  Boolean   wakeEnabled;  /* 1 = enable startup timer, 0 = disable */
  SInt8     filler;
};
```

GetStartupTimer returns the startup time and the state of the startup timer. If a particular Macintosh doesn't support the startup timer, GetStartupTimer returns 0.

Q *We'd like to open a (usually color) dialog with GetNewDialog such that, if indicated by the user, it ignores the 'dctb' resource and opens in black and white instead. This is a feature request from our users. The best solution we've got so far is to check to see whether we want a dialog to appear in color before calling GetNewDialog. If we want to suppress color, we patch GetResource with code that will look for any call to fetch a 'dctb' resource with the same ID as our dialog. If the patch detects such a call, it returns a nil handle; otherwise, it calls the original GetResource trap and returns its return value. When the GetNewDialog call is complete, we unpatch GetResource if we had patched it. Is this a good solution?*

A The method you describe — patching GetResource — will probably work, but is a needlessly complex solution to the problem. In general, patching should be considered a last resort solution, suitable only when there are no other options.

Why not just have two duplicate DLOG resources (both referencing the same DITL), one with a corresponding 'dctb' and one without? (DLOG resources are small, on the order of 30 bytes, so there shouldn't be a size problem.) You can then pass the appropriate ID to GetNewDialog to get either a color or a noncolor dialog. If you want to make this "automatic," make the DLOG resource without a corresponding 'dctb' have an ID that's, say, 1000 more than the one that has a 'dctb'; then keep a global variable that contains either 0 (if the user wants color) or 1000 (for black and white). When you call GetNewDialog, add the global to the (color) dialog ID and pass the result to GetNewDialog. Voilà! You get color if the global is 0, black and white if it's 1000. This is far cleaner and safer than patching could ever be — and easier, too.

Q *I have a question about the MailTime structure and setting the postIt.coreData.sendTime field based on the contents of GetDateTime. The messages I'm reading have the time and date in them, so I can retrieve the time the message was received. I convert this to a DateTimeRec and then call Date2Secs. What's the real way to handle this? I've written a routine that seems to be the right approach, but the time in the mailbox is always wrong, though my location* is *correct in the Date & Time control panel.*

A Here's a snippet that does what you need:

```
void MacToMailTime(unsigned long macTime, MailTime& mailTime)
{
   long             internalGmtDelta;
   long             dlsDelta = 0;
   MachineLocation  aLocation;

   ReadLocation(&aLocation);
   internalGmtDelta = aLocation.gmtFlags.gmtDelta & 0x00ffffff;
   if (BitTst(&internalGmtDelta, 23))
      internalGmtDelta = internalGmtDelta | 0xff000000;
   mailTime.time = macTime;
   mailTime.offset = internalGmtDelta;
}
```

Q *I'm trying to get unread mail messages from the PowerTalk mailbox, but the SMPGetNextLetter routine is returning an error of -903. That's a PPC Toolbox error noPortErr! What's going on?*

A The problem is that you haven't set the isHighLevelEventAware flag in the SIZE resource for your application. This is necessary because the AOCE Standard Mail Package routines require your application to accept high-level events.

Q *How can I count the number of unread messages in the PowerTalk mailbox?*

A Using the existing AOCE programming interfaces, this isn't possible. However, a new mailbox API will be available soon (if it isn't already) that will allow these types of operations.

Q *I've been using the Standard Mail Package to add mail capability to my application. In some circumstances we generate text or PICT files and add them as attachments to the document we're mailing. The documentation states that the enclosure isn't actually added until well after SMPAddAttachment has returned. I'd like to delete the files as soon as possible after I've generated them. How do I know when the file has been completely copied so that I can delete the original?*

A The problem with SMPAddAttachment is that the process is nondeterministic. The file is added by an asynchronous background process that's controlled by sending Apple events between your application and the Finder. Apple events are returned to your application during the file copy (they'll be handled by the normal Apple event mechanism).

Generally, these are the indications that the copy is complete:

- SMPMailerEvent returns with the kSMPDisposeCopyWindow bit set in the whatHappened field.
- The Finder sends a kAEReply Apple event to your application.
- The size of the enclosures field goes up by sizeof(FSSpec).

However, if you were to try to add another enclosure at this point, you might still end up getting the kSMPCopyInProgress error.

What you need to do is treat the kSMPCopyInProgress error as a "busy, try later" indication, and try the action again the next time through your event loop. This ensures that the Mailer (and the Finder) get a chance to move data between successive tries.

Q *I'm trying to write a patch that gets called when a floppy disk is inserted. I tried a GNE filter patch that looks for the diskEvt message in the event queue. It works, but it's not really what I want; the patch also gets called when any volume, not just a floppy disk, is mounted. Also, when a floppy disk is inserted, the patch gets called after the disk's icon shows up on the desktop, and I'd like to trigger the action before that. Any ideas?*

A The Finder always gets events before your GNE patch, which is why your patch gets called after the icon shows up on the desktop. Instead you should patch MountVol inside the Finder.

Q *I'm attempting to use the MenuHook routine called by MenuSelect to update a status bar with text as the user traverses menu items with the mouse. It seems that if I call TextEdit functions directly from within the function pointed to by MenuHook, problems occur: the mouse highlights the first item in a menu, but that item stays highlighted no matter where you move the mouse. In other words, it seems that MenuSelect stops working correctly or that the screen is no longer correctly updated. Can you tell me how to fix this?*

A You need to save and restore the graphics port in your MenuHook routine. Even if you aren't explicitly changing the port yourself, the TextEdit routines will probably leave it set to the edit record's owning port, not what it was when you started.

Q *I found the following declarations in Scripts.h:*

```
extern PASCAL Boolean IsCmdChar(const EventRecord *eventRecord,
                                short test)
   FOURWORDINLINE(0x2F3C, 0x8206, 0xFFD0, 0xA8B5);
```

But I can't seem to find any documentation for this call. Does such documentation exist? If this is what I think it is, it could be very useful.

A Whoops, thanks for pointing this out. That routine was introduced with System 7. We've updated the Macintosh Technical Note "International Canceling" (TE 23) accordingly. Here's a description of the routine:

```
FUNCTION IsCmdChar(keyEvent: EventRecord; testChar: CHAR): BOOLEAN;
```

This function tests whether the Command key is being pressed in conjunction with another key (or keys) that could generate testChar for some combination of Command up or down and Shift up or down. This accommodates European keyboards that may have testChar as a shifted character, and non-Roman keyboards that will *only* generate testChar if Command is down. It's most useful for testing for Command-period.

The caller passes in the event record, which is assumed by the function to be an event record for a key-down or auto-key event with the Command key down. The caller also passes in the character to be tested for (for example, '.'). The function returns TRUE if testChar is produced with the current modifier keys, or if it would be produced by changing the current modifier key bits in either or both of the following ways:

- turning the Command bit off
- toggling the Shift bit

Q *Someone has to ask this: just what are the "Miscellaneous Traps" toward the end of Traps.h, and in particular _HFSPinaforeDispatch?*

A Those few defines in Traps.h are leftover baggage:

```
/* Miscellaneous Traps */
   _InitDogCow             = 0xA89F,
   _EnableDogCow           = 0xA89F,
   _DisableDogCow          = 0xA89F,
   _Moof                   = 0xA89F,
   _HFSPinaforeDispatch    = 0xAA52,
```

0xA89F is really _Unimplemented and 0xAA52 is really _HighLevelFSDispatch. They were possibly left there to keep system builds working — or perhaps to keep the build engineers amused.

Q *To get started writing a raster printer driver for QuickDraw GX, I wrote a "skeleton" driver (only 5K!) that overrides only two messages: RenderPage and RasterDataIn. The RenderPage override merely posts debug notices before and after forwarding the message. The RasterDataIn override also posts a debug message, then returns immediately. So no data is sent to the printer; this is, after all, only a demonstration driver.*

A crash occurs after BuggyDriver forwards the RenderPage message but before control returns to the RenderPage override. RasterDataIn is called one or more times before the crash occurs. Since there's hardly any code, yet the driver still crashes, I'm guessing that

incorrect driver resources are to blame (I don't pretend that I've figured out the 'rdip' resource yet).

A You've stumbled onto a bug in QuickDraw GX: rendering into 32-bit-deep view devices at high resolutions causes the crash you're seeing. Since the problem is in several of the QuickDraw GX blitters, there's no workaround short of reducing the bit depth or the resolution, or both. This problem is fixed in version 1.1.

Q *I'm patching NewControl so that I can replace the standard controls on AV and Power Macintosh machines. This seems to work fine everywhere except in alerts. When an alert is posted, NewControl doesn't get called until after the original control is drawn once. Any ideas?*

A The "proper" way to do this is to patch NewControl and GetNewControl to change the procID to your CDEF's procID. This is pretty clean: the Control Manager thinks your CDEF was the one that was always asked for. The only drawback is that you'll have to make sure your resource file is always available and in the search path. Be sure to set the system bit in your CDEF to avoid constant reloading.

Q *I'm trying to call LMGetUnitTableEntryCount in my application, but when I compile for PowerPC I get a link error: the function isn't in the native libraries. Is this policy or an inadvertent omission? What can I do about it?*

A This is an oversight. You'll need to create an external function in a file (say, Extra.c) to access the low-memory global yourself (from native code only), as shown below. When an updated library is released, you'll only have to remove the Extra.c.o file from your link command and relink your application, not recompile it.

Your Extra.c file would be simple and look something like this:

```
// Compile with: PPCC -appleext on Extra.c -o Extra.c.o
// Add Extra.c.o to your PPCLink command line.
// Later, when a .xcoff file is provided by Apple, replace it with that,
// and delete your Extra.c.o file.
#if defined (powerc) || defined (__powerc)
   pascal short LMGetUnitTableEntryCount()
   {
      return *(short *)0x01D2;
   }
#endif
```

Of course, best of all would be to rewrite your application so that it doesn't depend on low memory at all.

Q *The cursor flickers when it's over a playing QuickTime movie. Is there any way to stop this? I don't want to hide it completely because the user needs to access controls elsewhere on the screen, and the movie is large, so hiding it only over the movie is still disorienting.*

A Unfortunately, there's no way to prevent the flickering cursor, short of hiding it completely. The Macintosh doesn't have a hardware cursor, so the cursor always

has to be hidden during blits to the screen. QuickTime 2.0 improved this situation by shielding the cursor less of the time, but it still happens.

Q *Is it OK for different nodes to run different versions of AppleTalk on a network?*

A You don't need to run the same version of AppleTalk for each node on the network. However, AppleTalk versions 53 and later support AppleTalk Phase 2, so if you're working on an application that depends on Phase 2 support (for instance, use of the NBP wildcard character "≈"), you'll need to use AppleTalk version 53 or later for all nodes running that application.

Q *When I use the LaunchApplication routine with the launchDontSwitch bit set in the launchControlFlags field for an application that doesn't have the canBackground size resource bit set, LaunchApplication returns 0 (noErr) but the application doesn't launch. What gives?*

A LaunchApplication doesn't return an error in this case because the application actually *is* launched. But since it doesn't have the canBackground bit set and it was launched into the background, it never gets any processor time, which means that it doesn't initialize anything and isn't added to the Application menu. If the user double-clicks an application that has been launched like this, it will bounce forward and get processor time, initialize itself, and be added to the Application menu as usual.

Q *A colleague of mine who is a Latin freak always calls me a "lens culinaris." What does it mean?*

A You are being called a "lentil."

These answers are supplied by the technical gurus in Apple's Developer Support Center. Special thanks to Pete "Luke" Alexander, Mark Baumwell, Mark "The Red" Harlan, David Hayward, Scott Kuechle, Larry Lai, Joseph Maurer, Jim Mensch, and Nick Thompson for the material in this Q & A column.•

Have more questions? Need more answers? Take a look at the Macintosh Q & A Technical Notes on this issue's CD.•

Newton Q & A: Ask the Llama

Q *When I try to print or fax something, the Newton usually runs out of memory. I'm sending a lot of data to my print format, but it seems that the **fields** frame passed to SetupRoutingSlip should have only a reference to the data. What am I doing wrong?*

A Unfortunately, you're missing one important step in the process of printing. The **fields** frame is eventually placed in the outbox soup. Thus all references are followed, which means that all the data you placed in the **fields** frame is duplicated. The duplication occurs in the soup:Add call.

In other words, you probably have a large data structure (or view, or proto) in your application. You put a reference to this structure in the **fields** frame in SetupRoutingSlip. When the user accepts the item to be printed (or faxed, beamed, or whatever), the **fields** frame is placed in the outbox soup, causing your structure to be duplicated.

Ideally, you should pass as little data as possible in the **fields** frame. As an example, assume that your data is all in a soup. You would pass information that allowed you to construct a query that returned the soup data of interest. This may be the index to search on plus the key to search for. You may even pass the query frame itself. Note that any changes made from the time the print request is made to the time the printing occurs will be reflected in the printed items. This may not be what you want. As in most cases, there are tradeoffs.

Q *I'd like to have something like a viewIdleScript in my communications endpoint. The endpoint proto doesn't contain a way to do this. What's a good way to do it?*

A One solution is to include your endpoint in a view object, such as a clView. You can then treat the whole clView as the communications object. You can use the view messages associated with opening a view to manage the control of your endpoint. Similar things can be done with viewQuitScript. And of course you can use a viewIdleScript.

Also note that using a view gives you a way to provide visual feedback (assuming it makes sense). Take a look at protoLlamaTalk in the LlamaTalk sample on this issue's CD for an example.

Q *How do I get a text view to redraw itself with a new font?*

A Simply use SetValue on the viewFont slot:

```
SetValue(theParaView,
   'viewFont,
   {family: 'espy, face: kFaceBold, size: 12});
```

Q *I'm trying to use a view that has vfFillGray as the background. I've found that it looks really bad. Why is this?*

A The pattern vfFillGray is an alternating on/off checkerboard pattern of pixels (that is, 50% gray). With the current state of the technology, all passive LCDs

---

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Personal Interactive Electronics (PIE) division. Send your Newton-related questions to NewtonMail DRLLAMA or AppleLink DR.LLAMA. The first time we use a question from you, we'll send you a T-shirt.•

have problems displaying large areas of 50% gray (or large areas of black). The problem (called *crosstalk*) is worse when you have alternating on and off pixels. Basically, the LCD freaks out. You should avoid using large areas of 50% gray where possible.

Q *When I try to add an index to my soup I sometimes get an exception -48019, but not always. What's going on?*

A That particular exception indicates that an entry in your soup has a value of NIL in the slot you're trying to create an index for (that is, the entry contains the slot with a value of NIL). You can easily recreate this error by trying to add an index on the **bday** slot in the names file. Here's some code that you can type in the inspector:

```
call func()
begin
   local nameSoup := GetStores()[0]:GetSoup(ROM_cardfilesoupname);
   nameSoup:AddIndex(
      {structure: 'slot, path: 'bday, type: 'Int});
end with ();
```

In this case the error occurs because the default cardfile entry has a value of NIL for the **bday** slot. The solution is to make sure that there are no soup entries with a value of NIL for the slot that you want to use for the new index. This is best done in the design of your soup data.

If this isn't possible, the only solution is to make sure that all entries in the soup either have a valid value for the new index slot or do not contain the new index slot. Unfortunately, you don't know in advance if the new index will fail. In this case you can wrap the code that adds the index in a **try/onexception** clause. If an exception occurs that has the -48019 error number, you know that you have to iterate through the soup and fix entries.

Also note that you may want to keep a list of those "fixed" entries around since you may have to unfix them after the index has been added. In other words, it's OK for an entry to have a NIL value in an indexed slot after the index has been added to the soup.

Q *I have a protoA2Z_TDS controlling a protoTextList. There are two things this combination doesn't do: (1) As the protoTextList contents are scrolled, the protoA2Z_TDS doesn't update the current letter, and (2) when the user clicks on a letter in the protoA2Z_TDS, I want to scroll the protoTextList to the appropriate place. How do I do these things?*

A For those who may not know, protoA2Z_TDS is sample code provided by PIE Developer Technical Support. In answer to the first question, all you need to do is set the curIndex to the correct value, where A is 0 and Z is 25. If you use SetValue, the display will update for you. So if your protoA2Z_TDS was declared as **indexer**, and you wanted to change it to the letter B, you would do this:

```
SetValue(indexer, 'curIndex, 1);
```

You could also write a method of the protoA2Z_TDS that would update the display based on a character:

```
SetIndex := func(newChar)
begin
   local newIndex := Ord(Upcase(newChar)) - 65;
   if newIndex < 0 then newIndex := 0;
   if newIndex >= numIndices then newIndex := numIndices - 1;
      SetValue(self, 'curIndex, newIndex);
end;
```

Note that this function will try to do the right thing with weird input. However, if you're expecting the full range of Unicode values, you'll have to change the function to accommodate multibyte characters.

Now let's tackle question number 2. You need to know about three things in the protoTextList that aren't yet documented in the *Newton Programmer's Guide*:

1. There's a slot named lineHeight that contains the height of each line in pixels.
2. The protoTextList uses SetOrigin to scroll. Therefore, the slot viewOriginY contains the number of pixels that the view is scrolled (and viewOriginY DIV lineHeight is the line number of the top displayed line).
3. There's a method DoScrollScript(offset) that scrolls from the current position by the specified offset (in pixels).

Given these three pieces of information, here's a method for a protoTextList that will highlight a particular row and make it visible:

```
protoTextList.HiliteRow := func(index)
begin
   // highlight this item
   SetValue(self, 'selection, index);
   // scroll as necessary
   local topItem := viewOriginY DIV lineHeight;
   if index < topItem or (index >= topItem + viewLines) then begin
      // we need to scroll so that the index is the first item
      :DoScrollScript(-(topItem - index) * lineHeight);
   end;
end
```

Of course you still have to calculate what index to pass to the function. But that should be fairly straightforward. The protoA2Z_TDS will give you the first letter, which you can then find in your listItems array in the protoTextList. Note that if the listItems array is sorted, you can use a binary search to find the correct index.

Q *What is the origin of the llama?*

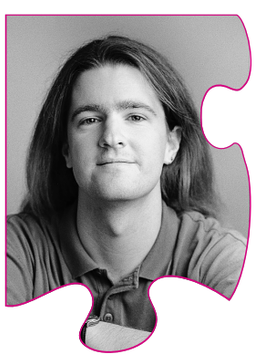
A The first evidence of llamas dates back to the llama raptor discovered by Dr. Leakey in the jungles of the Amazon. This find was dated back to the late "Jurassic Park" period. Early llamas are thought to have been both more violent and more intelligent than today's breeds. Cave paintings from the hills of Venezuela clearly depict early humans in use as pack animals for tribes of llamas.

Thanks to Erik York and our PIE Partners for the questions used in this column, and to Bob Ebert, J. Christopher Bell, Mike Engber, Neil Rhodes, Kent Sandvik, Jim Schram, Maurice Sharp, and Bruce Thompson for the answers.•

Have more questions? Need more answers? Take a look at PIE Developer Info on AppleLink.•

A Branch Too Far

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer and guest puzzler Chris Yerga. 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 please, make KON & BAL's day by submitting a puzzle of your own to AppleLink DEVELOP.



CHRIS YERGA

Chris I have a piece of code that runs fine on my Quadra, but when I run it on a plain old 68000 it crashes.

KON A 68000? So you're still trying to get GX to run on that Mac Portable in Cary's office, huh? How does it crash?

Chris With an address error.

KON What's hard about that? Your code is doing a 2- or 4-byte access to an odd address, which is OK on a 68040 but not on a 68000. It's a trivial problem.

Chris That's what I thought, but the address it's accessing appears to be uninitialized data. The code is simply allocating a block and then storing a pointer in the block, but the store never seems to occur, because afterward the block has random data in it. And of course the block itself is long-aligned, because it came from the Memory Manager, so there's no problem there.

KON Maybe on your Quadra, but on a Mac Plus the Memory Manager allocates blocks that are word-aligned.

Chris Thanks for the history lesson, chief, but this isn't a Mac Plus and the Memory Manager on this machine is much different — much simpler, actually. It just so happens that our Memory Manager always long-aligns blocks.

KON Since when are you writing new Memory Managers? I thought you wrote graphics code.

CHRIS YERGA During his four years working on QuickDraw GX, Chris learned a lot about graphics systems and large projects. He's currently employed by Catapult Entertainment, where he learned that a Sega is kinda like a Macintosh, a SNES is kinda like an Apple IIGS, and carbon dioxide can be explosive.•

Chris Yes. But the cornerstone of any decent graphics system is its Memory Manager — I wouldn't expect a "QuickDraw classic" guy like you to understand.

KON I understand memory management just fine, Jackson. Show me where you're setting up this data.

100 Chris This is the interesting section:

```
NewDMAQueueEntry
    ...
    +0030  0020C5D6  MOVE.L    D3,D0
    +0036  0020C5DC  _NewPtr
    +0038  0020C5DE  MOVE.L    A0,-$0008(A6)
    +003C  0020C5E2  MOVE.L    -$000C(A6),-(A7)
    +0040  0020C5E6  MOVEA.W   -$000E(A6),A1
    +0044  0020C5EA  MOVE.L    A1,-(A7)
    +0046  0020C5EC  JSR       *+$53A4             ; 00211990
    +004A  0020C5F0  MOVE.L    D0,-$000C(A6)
    +004E  0020C5F4  MOVE.L    A2,(A0,D0.L)
```

It makes the _NewPtr call, makes some other function call, and then stores the pointer in A2 into our newly allocated buffer.

KON What's the other function call doing?

95 Chris I'm not sure, actually. The C source doesn't indicate that a function call should be happening:

```
buffer = NewPtr(totalSize);
count = count * size;
*(long *) (buffer + count) = (long) handlerProc;
```

The only thing I see happening in the source code is a multiply. You don't need a function call for that.

KON Could this be some wacky C++ operator overloading nonsense? C++ is very good at generating extra function calls. I think we'd better bring a SmartFriend in on this one.

Chris Don't bring out the big guns just yet. The code is written completely in plain old C: no C++ and no CFront.

KON It's got to be code you've written, because it's a PC-relative JSR. Since it's not an A5-relative JSR, it's not going through the jump table, so it couldn't have been linked in from a library or something external to your program. It looks like a call to a static function.

Chris Actually, all the JSR instructions in this code are PC-relative. I wrote a tool that transforms all JSRs that go through the jump table to PC-relative JSRs. You may want to sit down for this next part — it's a little tricky.

KON That's why it ended up in the Puzzle Page. Let's hear it.

90 Chris All the code is being built into a ROM; however, our development system of choice only allows us to build our code as a Mac application. So I created a custom tool that postprocesses the application and turns it into something that will run out of ROM. Basically we use the CODE 0 jump table to link everything together.

KON I see. You know where the code will reside in ROM, so you fake out the jump table to make it look as if all the segments are loaded. Since nothing will ever move or unload, your ROM jump table entries never need to change. Pretty tricky.

Chris But not tricky enough. The scheme you describe will work, but it has two problems. First, our ROM can be mapped into different addresses, so the code must be completely relocatable. A Mac jump table contains JMP instructions with fixed long addresses (for example, JMP $4083143A), which are not relocatable. Second, as a cycle counter like you should know, the jump table is superfluous here, because no code will ever move or unload while it's running. You're doing a JSR to the JMP in the jump table — the extra JMP is unnecessary and wastes cycles.

KON I didn't know you GX guys counted cycles. How do you get away with removing the extra JMP?

85 Chris Our tool scans all the object code for all instructions that reference the jump table. They are JSR xx(A5) (function call), PEA xx(A5) (pass a function pointer on the stack), and LEA xx(A5),Ax (get a function pointer in a local variable). When it finds one of these instructions, it looks up the function in the jump table and changes the instruction to a PC-relative version that simply references the address of the target function directly. Everything is PC-relative, so it relocates correctly, and there are no extra instructions.

KON But the PC-relative addressing mode has only a 16-bit offset, so you can only reference functions within 32K of the PC. Is your ROM that small?

80 Chris Are you kidding? The pictures for the About box are bigger than that. When we need to reference a function that's beyond 32K, we create a "jump island" that's still PC-relative but allows us a greater reach.

```
0020122A: JSR     $00201FA8       ;go to jump island
0020122E: MOVE.W  D0,(A3)
...
00201FA8: LEA     *+0,A0          ;get pc
00201FAC: ADDA.L  #$00014B02,A0   ;add long offset
00201FB2: JMP     (A0)            ;jump to destination
```

KON That looks suspicious to me. You're messing with A0 in your jump island.

75 Chris But that should be OK because the whole thing is written in C, which never passes a parameter in A0 and never expects A0 to be preserved.

KON I can't find a specific problem, but I'm still a little suspicious of all this OS code you're writing. You said that this code runs fine on your Quadra. Does the Quadra version undergo the jump table transformation process?

Chris No. The Quadra version is run just like a Mac application out of RAM.

KON Tell me all the differences between the two environments.

70 Chris The 68000 version of the software is different in two ways. First, it's generated with the exact same compiler, but without 68020 code generation enabled. Then, I run it through my BuildROM tool, which

transforms all the jump table references to PC-relative references. The 68000 target hardware is a diskless system; the only way to get code into it is through ROM cartridges, so we can't try to run a version of the software that hasn't undergone the BuildROM step.

KON So the bug may have nothing to do with the differences between the 68040 and 68000 processors — it may be the BuildROM process. Let's sum up what we know: First we saw a piece of code that allocates a block of memory and stores a value into that block, but the store never seems to happen. Near that code is an unexplained JSR which we believe is a function call. Finally, we know that a critical difference in the environment is that you alter the code path of function calls. It's really starting to smell like your BuildROM tool.

Chris The evidence is all circumstantial, counselor. The only questionable part of the BuildROM process we've seen is my usage of A0 in the jump islands, but I can't see a case where a C function call takes a parameter in A0 or assumes A0 is preserved.

KON We should trace through the NewDMAQueueEntry routine and see what that mysterious function call is doing.

65 Chris I put in a breakpoint. But when I run the program again, I drop into the debugger with a debug message complaining that some kind of parameter is out of range.

KON You mean the bug isn't reproducible?

Chris After many tries, this is the first time it's failed this way.

KON Hmm. What parameter is out of range?

Chris The debug message doesn't say, exactly. The value it's complaining about is in D2. The value is $00004E56.

KON That's a funny-looking number. It looks like an opcode to me. **dh 4E56** tells me it's a LINK A6,#xx instruction. Where did the value in D2 come from?

60 Chris From a MOVE.W (A0),D2 instruction. The code seems to think A0 points to some data.

KON But it looks as if it points to code instead. Where is A0?

Chris It points to the start of a routine called VBLHandler.

KON VBLHandler sure sounds like an interrupt service routine to me, which would explain the various failure modes. Is the routine written in C? C routines don't bother to preserve A0, so your interrupt routine is trashing a register!

55 Chris There's some inline assembly code to save all the registers. Take a look:

```
void VBLHandler(void)
{
asm {
   MOVEM.L   A0-A6/D0-D7, -(SP)
};

FlushQueue();
HandleVBLTasks();
```

```
asm {
   MOVEM.L   (SP)+, A0-A6/D0-D7
};
}
```

KON It seems quite suspicious to me that A0 points to the beginning of this routine. Sounds like your jump islands are at work. Look at the interrupt vector for your VBL handler.

50 Chris On this particular machine the VBL is handled as a level-6 interrupt. The level-6 vector is at $0078. It points to a jump island entry like the one above.

KON And the first thing it does is trash A0 before your inline assembly gets a chance to save it! Maybe you should stick to drawing bitmaps and leave the OS work to someone else.

45 Chris The problem is that the code that installed the interrupt handler is more than 32K away from the handler itself. So when it did an LEA xx(A5),Ax to get the address of the interrupt handler, the ROM builder tool needed to stick a jump island in there. Nasty. But I still need a way to do PC-relative jump islands.

KON Use the stack, son. Try this:

```
PEA *               ;push the pc
ADD.L  #xxxxxx,(sp) ;add a long offset
RTS                 ;jump to the destination
```

Chris All this and you can draw bitmaps too! I'll fix the ROM builder tool to do this. But somehow I doubt that a VBL interrupt was hitting us at the same spot in my NewDMAQueueEntry routine every time. There must be another problem there.

KON We still have the breakpoint there; before we recompile and fix the bug, let's run it again and see if we can get the original failure mode to happen again.

40 Chris This time we hit the breakpoint. We trace over the NewPtr and see that it returns a valid pointer in A0. We step into the JSR at $20C5EC and it takes us to one of my jump islands.

KON Which alters A0, and then jumps to the function being called. What is the function doing?

35 Chris It's a very short routine; it just takes some parameters off the stack and does a few multiplies. It returns the result in D0 just like any normal C function would.

KON But is it using the trashed A0 for anything?

Chris No. In fact, it's not using any address registers at all. It only uses data registers for the multiplies. I repeat: no C function would ever take a parameter in A0.

KON So we return to the NewDMAQueueEntry routine, with a return value in D0. We save D0 in a local variable on the stack frame, and then hit the instruction at $20C5F4, which stores a value in the buffer pointed to by A0.

30 Chris But A0 still has the result of the jump island calculation in it. The code didn't set up A0 to point to anything!

KON Look at the listing of NewDMAQueueEntry again. The code gets the result of the NewPtr call in A0, makes the function call, and then assumes that A0 still has the valid pointer in it. But meanwhile, some wannabe OS programmer has gotten in there and hosed A0 on us!

20 Chris That function call seems to be doing the multiply. It must be some runtime math library that the compiler uses.

KON I'll bet one of your variables is a long word. With 68020 code generation turned on, the compiler was able to generate a long multiply instruction, but the 68000 doesn't have a long multiply instruction, so it calls the math library.

10 Chris I see. Since the math library was written by the same people who wrote the compiler, their code generator knows that A0 won't get trashed, so it doesn't bother to save and restore it around the call to the long multiply routine. Pretty sneaky.

KON But nice. You want the person writing your code generator to be the ultimate cycle counter. Since the Mac Segment Loader implementation doesn't trash A0, it's a worthwhile optimization for them to make.

Chris Except in this case, the code that performs a multiply is more than 32K away from where the math library resides in the ROM, so it hits a jump island and loses the value of A0.

KON Even a lowly register like A0 is sacred sometimes.

Chris So it appears.

KON Nasty.

Chris Yeah.

SCORING

80–100 Please fax your resumé to Catapult Entertainment, Inc., (408)366-2471.
60–75 We also have junior positions available.
40–55 Don't worry; just let CopyBits do the tricky stuff.
10–35 I see you've done your share of long multiplies.•

Thanks to Josh Horwich, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

D

E

F

G

H

I

K

Q

R