Dynamic Display Dilemmas • Attaching and Embedding Scripts • Scripted Text Editing

develop

The Apple Technical Journal

Issue 26 June 1996

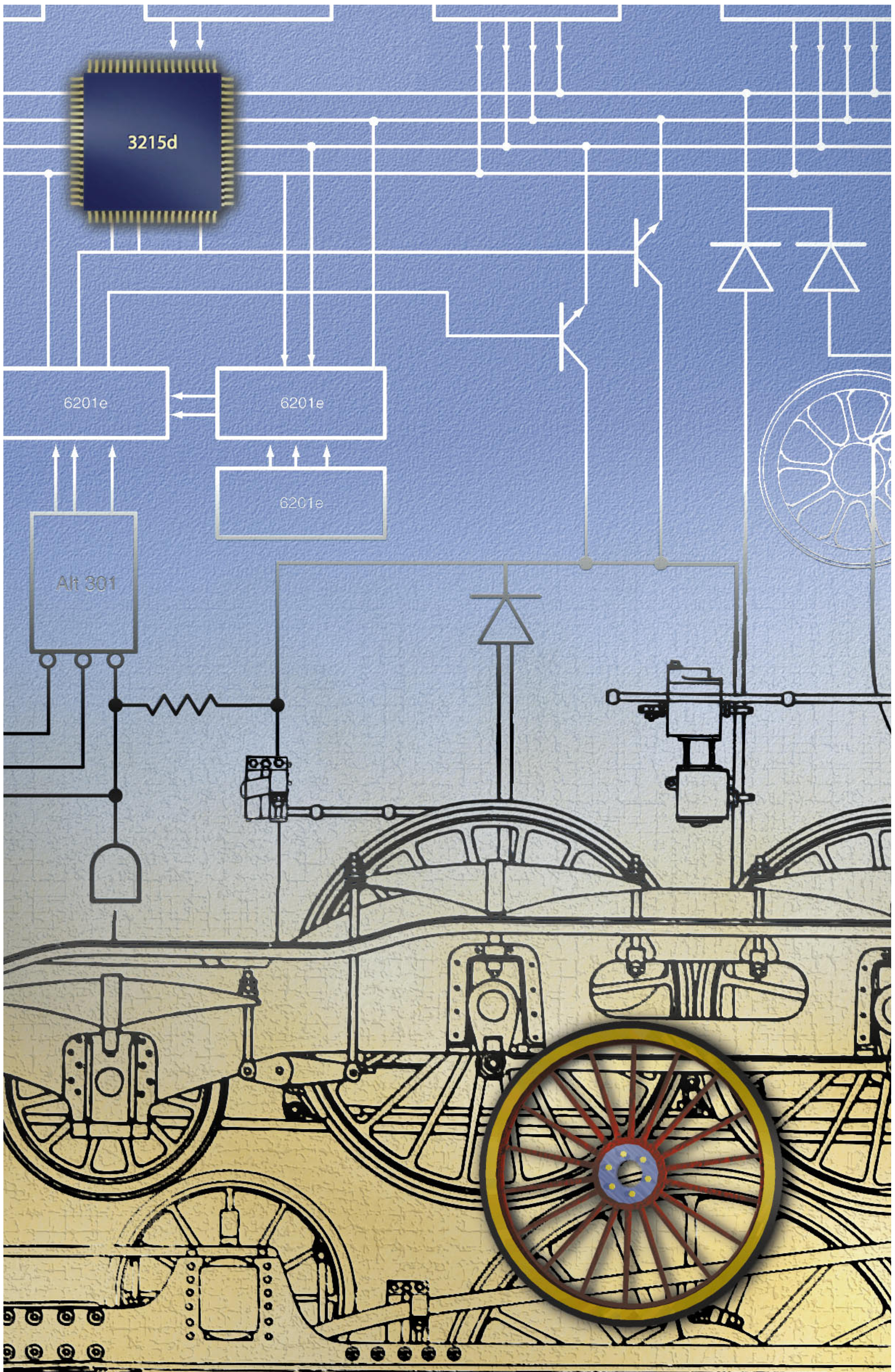
Planning for Mac OS 8 Compatibility

Connecting Users With QuickTime Conferencing

OpenDoc Parts and SOM Dynamic Inheritance

Adding Custom Data to QuickDraw 3D Objects

64-Bit Integer Math on 680x0 Machines



$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Managing Editor *Toni Moccia*

Technical Buckstopper *Dave Johnson*

Bookmark CD Leader *Alex Dosher*

Able Assistant *Meredith Best*

Our Boss *Mark Bloomquist*

His Boss *Dennis Matthews*

Review Board *Brian Bechtel, Dave Radcliffe, Quinn "The Eskimo!", Jim Reekes, Bryan K. "Beaker" Ressler, Larry Rosenstein, Nick Thompson, Gregg Williams*

Contributing Editors *Lorraine Anderson, Linda Fogel, Toni Haskell, Judy Helfand, Cheryl Potter*

Indexer *Marc Savage*

ART & PRODUCTION

Art Direction *Lisa Ferdinandsen*

Technical Illustration *John Ryan*

Formatting *Forbes Mill Press*

Production *Diane Wilcox*

Photography *Sharon Beals, Josephine Tsen, Socorro Anaya*

Cover Illustration *Graham Metcalfe of Metcalfe/Shuhert Design*

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. It provides developers of Apple-platform products with technical articles and code that 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 *Developer CD Series*, which is part of the Apple Developer Mailing available through the *Apple Developer Catalog*. 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 Technotes, 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*.

Much of the CD contents, including the *develop* issues and code, is also available at ftp://ftpdev.info.apple.com and in the Developer Services area on AppleLink. See also the Web site for Apple Developer Services and Products, at http://dev.info.apple.com.

Macintosh Technical Notes. A designation like "(CS 06)" after a reference to a Macintosh Technical Note or Macintosh Technical Q&A indicates its category and number on this issue's CD or on the Internet. (CS is the ColorSync category.) The new (uncategorized) Technotes are designated by number alone.

CONTACTING US

Feedback. Send editorial comments or suggestions to Caroline Rose at crose@apple.com or AppleLink CROSE. Technical questions about *develop* should be directed to Dave Johnson at dkj@apple.com, AppleLink JOHNSON.DK, or CompuServe 75300,715. You can send a fax to Caroline or Dave at (408)974-9423 or write to them at Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at develop@apple.com or AppleLink DEVELOP.

Subscriptions and back issues. You can subscribe to *develop* through the *Apple Developer Catalog* (see ordering information below) or use the subscription card in this issue. You can also order printed back issues through the catalog. The one-year U.S. subscription price is $30 (for four issues and four *develop Bookmark* CDs), or U.S. $50 in other countries. Back issues are $13 each. These prices include shipping and handling. For Canadian orders, the subscription price includes GST (R100236199).

Apple Developer Catalog. To order *develop* or other products through the catalog, or to make subscription-related queries, 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. You can send e-mail to order.adc@applelink.apple.com or AppleLink ORDER.ADC, or write *Apple Developer Catalog*, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319. For all subscription changes or queries, *please be sure to include your name, address, and account number as it appears on your mailing label.*

Printed on recycled paper by Stream International, USA

Issue 26 June 1996

ARTICLES

EDITOR'S NOTE

CAROLINE ROSE

A couple of issues ago, I told a story here about how my friend John had failed to find some song lyrics on the Web before I managed to locate them by old-fashioned, real-world means. A few of you wrote to say how easily you found those same lyrics on the Web. Maybe they were posted after the incident I related, or maybe John just didn't look in the right place, but my point was a general one: the Web is not the world.

At first I thought that editorial would be controversial, but before it was published I noticed similar remarks starting to appear elsewhere, along the lines of the Internet and the Web being overrated. Since then I've seen even more critical articles on the subject — the inevitable backlash, I suppose. Now that such kvetching has become socially acceptable, I should probably turn to another subject, but alas...

My main complaint is with the quality of a lot of what's posted online. I don't mind so much if someone's personal home page is a bit rough, but large corporations that should do better seem not to be doing even minimal copyediting and fact checking on what they post to the Web. It's gotten to the point where, to some people, being published on the Internet is becoming synonymous with being low quality. I even came across this sentiment in a review of a book (not related to computers) in the *New York Times*: after criticizing the book for sloppy editing, the reviewer wrote, "If this is the way books are going to be published, we may as well just shove the typing onto the Internet and forget about bound volumes altogether."

Why is it that the highest-quality online publications are those that are also published in print? It's as if "committing" something to print makes it seem more respectable, more enduring. As a provider of not only *develop* content but also a newsletter of my own on the Web, I find this ironic. Ever since my publications have been made so easily available online, reader feedback indicates that many more people have been referring to back issues; they treat all the content — past and present — as a single, timeless body of information. This timelessness argues for the same attention to quality online as in the print medium, or at least for more efforts in that direction.

I think one problem may be the confusion about where to put Web publishing in an organization. Most Web-related job descriptions I've seen ask for a content provider, formatter/designer, and HTML expert all rolled into one. That's like having authors of *develop* articles design the page layout and produce the printed product. Ask one person to do it all and what do you expect?

You may not care about minor errors, but inattention to quality will extend, web-like, beyond punctuation and grammar into the more critical realms of coherency and accuracy. So please, take a second look at your Web pages and other online content with this in mind. The world will thank you.

CRose

Caroline Rose
Editor

CAROLINE ROSE (crose@apple.com, AppleLink CROSE) finds her most difficult editing job to be rephrasing her work history for her bio in *develop*. There are only so many ways to say she's been working in the computer industry for a very long time, in various writing, editing, and programming capacities. The good news is that, having edited *develop* for five years, she finally qualifies for a sabbatical, which she'll be taking by the time you read this. She's sorry to miss Apple's Worldwide Developers Conference but decided that springtime in Tuscany was a fair tradeoff.•

LETTERS

TOOLFRONTEND FIXES

Thanks to Tim Maroney for his excellent column on ToolServer and CodeWarrior (*develop* Issue 25). But there's a bug in ToolFrontEnd that causes CodeWarrior IDE 1.5b2 to throw away the preferences every time I use it. In case you're interested, I have the solution to the problem. In addition, I've fixed a dialog for the CodeWarrior 8 API and cleaned up the code for the new API.

This bug aside, this excellent little utility has helped me a lot in my development.

— Andreas Magnusson

Thank you for the bug report. I've created a new version of ToolFrontEnd that contains your bug fixes and others; it can be found on this issue's CD. The plug-in API was in a state of flux when I wrote ToolFrontEnd — the new API documentation arrived on the day of my deadline for the CD, so there was no opportunity to adapt my first release for it.

I'm glad to have helped in your development; that's the real reason I write these things.

— Tim Maroney

FONTTOPICT SNAFU

Regarding this code in FontToPict in Issue 25's Print Hints column:

```
MakePSHandle(qdFont, qdStyle,
   myEncoding, &picCommentHdl);
PicComment(kPostScriptHandle,
   GetHandleSize(picCommentHdl),
   piccommentHdl);
```

I recall that the second argument to PicComment is a word, which means you can't have a picture comment bigger than 32767 bytes. I think Type 1 fonts are usually quite close to this size. Should people be worried about this?

— Lawrence D'Oliveiro

Color me stupid. You're right, people should be worried about this when they're sending the whole font. Hopefully you'll be sending only the portion of the font that you'll actually need, so the data requirements will be less. But if you're not, you need to break up the font data or use another mechanism to send it to the printer. Sorry about that.

— Dave Polaschek

SCRIPTABLE DATABASE UPDATE

When I try to use CodeWarrior 7 to compile Greg Anderson's Scriptable Database 1.0a11 (from his article on **whose** clause resolution in *develop* Issue 24), I get the following link error:

```
: mpwexit.c: '__cleanupandexit__'
referenced from '_exit' is undefined
```

How can I fix this?

— Jean Jourdain

You can't fix it; that version of the Scriptable Database won't compile with CodeWarrior 7, only with CodeWarrior 6 (I'll spare you the gory details). But the new version on this issue's CD (1.0a15) has been updated so that it works fine with CodeWarrior 7 and later. Sorry for any inconvenience.

— Dave Johnson

GOOD TIMING

Thanks to Martin Minow for his "Timing on the Macintosh" article on *develop*'s CD. It saved me from having to

ALL OPINIONS ARE INVITED

We welcome your letters to the editor, especially regarding articles published in *develop*. Write to Caroline Rose (crose@apple.com or AppleLink CROSE) or, if technical *develop*-related questions, to Dave Johnson (dkj@apple.com or AppleLink JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did). Address subscription-related queries to order.adc@applelink.apple.com or AppleLink ORDER.ADC.•

hunt down and strangle whoever wrote "you must write an application-defined routine that calculates the elapsed time" in *Inside Macintosh: OS Utilities* and then didn't supply one.

— Isidore Ducasse

Inside Macintosh can't supply code for everything; I'm glad develop *could help fill the gap. Note that the Timing article has been updated on the CD.*

— Caroline Rose

OODL(E)S OF SPEED IN LISP

Dave Johnson's excellent column on OODLs in Issue 24 is informative and straight to the point. When he's talking about the overhead associated with dynamic languages, however, he's not quite up to date. Dynamic languages need not be slower than static languages. They *can* be, if the programmer isn't interested in speed. But on the other hand, numerical code in a modern LISP is every bit as fast as FORTRAN or C code, if the programmer cares to add a few declarations. There's no need to add external modules for speed nowadays.

True, some dynamic languages have a lot of runtime overhead, but LISP isn't one of them. This fact needs to be emphasized to programmers, not the obsolete idea that LISP is a slow and memory-hungry dinosaur. Interpreted LISP might have been slow 15 years ago, but so was BASIC. Unfortunately, many programmers still think LISP is interpreted, and the comparison between a compiled language such as modern C or Pascal with an ancient interpreted LISP implementation is simply not fair, nor is it correct. With Common LISP, lexical scoping, and modern compiler technology, LISP can be just as fast as any static language. So, your example of a QuickDraw 3D renderer in LISP is in fact an excellent idea.

— Peter Bengtson

You're right, of course. Writing time-critical, number-crunching code in LISP is eminently practical now. Among dynamic languages, LISP in particular has matured in a big way and is now almost a hybrid language: full dynamism if you want it (with some accompanying overhead) or, with appropriate declarations and "explicitness of purpose" by the programmer, the speed (and brittleness!) of a static language. In a sense, it's the best of both worlds, letting the programmer *decide what best fits the situation. So yes, my example was flawed, though I hope the spirit of it came through despite this.*

— Dave Johnson

TOOTING OWN OUR HORN: *develop* WINS BIG IN COMPETITION

We're happy to announce that *develop* has won top honors in the STC's 1995 Northern California Technical Communications competition. STC is the Society for Technical Communication, an international organization of more than 18,000 writers, editors, and other technical communicators. In its category of Monthly or Quarterly Magazines, *develop* won not only the highest-level award, Distinguished Technical Communication, but also Best of Category. It then went on to win a Merit award in the STC's International Technical Publications Competition.

We're going to indulge ourselves here and list some of the judges' comments that we're particularly fond of.

The writing was very focused and stuck to the article's point. All articles seemed very informative.

Very well organized and well laid out.

The voice is very personable without being overly familiar.

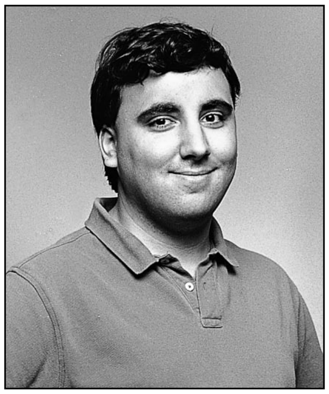
The articles use humor appropriately. The material is very readable.

develop *is a very polished, engaging publication from beginning to end.*

It's nice to get feedback like this from the competition judges, but you, our readers, are the judges who count the most. You're the ones we want to be sure are happy with *develop*. We'd like to take this opportunity to thank you for the valuable input you've given us over the six and a half years of *develop*'s existence, and to ask you to please keep it coming. Without your support and encouragement — and your critical feedback — we wouldn't be what we are today.

Planning for Mac OS 8 Compatibility

One of the most important goals for Mac OS 8 (formerly known by the development name "Copland") is the preservation of compatibility with existing applications. Customers consistently rank compatibility as a critical factor in their decision whether to upgrade to a new OS release, with good reason. This article sheds light on what will and won't be compatible, and gives developers a road map for ensuring compatibility with the Mac OS 8 release.


STEVE FALKENBURG

As one of the driving forces behind Mac OS 8, compatibility is at the forefront of the minds of Apple engineers hard at work on this system software release. Given the track record of nearly seamless compatibility with the Power Macintosh, customers will expect their existing applications to run under Mac OS 8 with few or no problems. Apple is working hard to deliver on this promise, and we're beginning to succeed. Most of the specific information for this article was learned the hard way — by getting many existing applications up and running.

Of course, tradeoffs must be made to move the platform forward. If Mac OS 8 were to remain compatible with all Macintosh software, the performance, reliability, and stability of the system would suffer. While some customers have been impressed by the stability of System 7, others would like to experience even fewer crashes and are willing to upgrade some of their software in the process. Apple's system software needs to be more stable, while still maintaining compatibility with most applications.

Luckily, there are quite a few techniques that you can use today and guidelines that you can follow to ensure compatibility with Mac OS 8. This way, you can impress your friends (and confuse your enemies) at compatibility labs by installing your software for the first time under Mac OS 8, and walking away 15 minutes later saying, "Gee, that was easy, everything works!" This is when following all of those *Inside Macintosh* chapters, *develop* articles, and Technotes will finally pay off.

Don't panic: Mac OS 8 isn't the compatibility "day of reckoning" that you've had nightmares about. I'm sure many of you have been told by Developer Technical Support, "Here's a really cool trick, but it may break in the future." In some cases, "the future" is in fact Mac OS 8, but on the other hand many techniques that are no longer being recommended (which we of course like to call "sick hacks") will continue to work.

STEVE FALKENBURG has worked on Mac OS 8 for several different groups at Apple, starting with being the Mac OS 8 liaison in Developer Technical Support, then moving into the Mac OS 8 High-Level Toolbox group, and finally finding a home in the Mac OS 8 Program Office Engineering Team. Steve is hoping to help ship Mac OS 8 without changing offices or groups again; then he plans to take a long vacation with his wife, Nancy, that doesn't involve computers or moving boxes.•

Remember that Mac OS 8 is just the first step in modernizing the Macintosh. Subsequent system releases will include features such as separate address spaces and full preemption for all applications. In the future, discouraged techniques will become areas of incompatibility; so even if your application runs under Mac OS 8, it's worth cleaning it up in preparation for future systems.

In this article, I'll go over a few things that will no longer work under Mac OS 8 as well as some of the techniques that will continue to work under Mac OS 8 but will break in future systems. For the more heinous examples of these techniques, I won't give code samples — I don't want people saying "Hey, I did it just like they did in *develop*" as an excuse. I'll also discuss some specific case histories of application compatibility problems, to further illustrate the need to be proactive when planning for compatibility.

For an overview of Mac OS 8, see http://www.macos.apple.com/macos8 on the World Wide Web, or the article "Copland: The Mac OS Moves Into the Future" in *develop* Issue 22. Other introductory documents can be found on this issue's CD. Please keep in mind when reading these materials, as well as this article, that the terminology has evolved over time and some of it may change again by the time you read this.•

WHAT WORKS AND WHAT DOESN'T

Before diving into guidelines, warnings, and examples, we'll start with an overview of exactly which types of software will be compatible with Mac OS 8, which will need to be updated, and which will need to be redesigned. This article focuses on applications, but an overview of compatibility in general is helpful to set the stage.

First, the good news: Well-written applications conforming to Macintosh development guidelines should run without any modification. This includes PowerPC™-native applications as well as emulated applications. Theoretically, you could have written a Macintosh Solitaire game in 1984 that would also run under Mac OS 8. There are, of course, caveats to application compatibility, which will be discussed later in this article.

Component software is becoming an important part of the Macintosh experience, and Mac OS 8 will support OpenDoc part editors as well as application-specific plug-ins — again, without any modification. Depending on the "parent" application, there may be issues with plug-in compatibility (as discussed later).

Now, the bad news: Existing extensions, control panels, desk accessories, ASLM libraries, and most drivers are unsupported for the Mac OS 8 release. Compatibility tradeoffs needed to be made in these areas to move the system forward and improve system reliability.

Macintosh power users often try to impress each other by comparing how many extensions load at startup, gracing their "Welcome to Macintosh" screen with several rows of happy little icons. The proliferation of INITs has made life as a Macintosh user exciting, if a little hazardous. With Mac OS 8, we're trying to attack the system stability problem head on by providing new, more reliable mechanisms for extensibility and patching. While the original Macintosh was presented as a complete solution, the Mac OS 8 team has realized that third-party extensibility is part of what makes the system great. This has led us to make ease of extensibility a key goal of the system.

As with extensions, for each existing code type that's not supported under Mac OS 8, a new and much improved mechanism will be provided. In other words, we'll still have MacHack entries for years to come, and they'll be just as cool but more reliable.

Other, less common software types are also unsupported. These include Text Services Manager input methods, FSM external file system modules, and debuggers. This article focuses on application-level issues, so I won't go into a lot of detail in these areas, but a section on migration paths is included in the article.

APPLICATION TAXONOMY

For the Mac OS 8 release, the types of applications that developers write can be split into several major categories. These categories have been defined by the Mac OS 8 project teams, and work is being done to ensure that each of the application types is fully supported by the Mac OS 8 design. This article doesn't cover each type in detail, but we'll use the classifications as guideposts for migration and compatibility plans.

USER INTERFACE APPLICATIONS

The first, and most familiar, application type is the *user interface application*. This type makes up the majority of applications on the Macintosh. These applications use windows, menus, and dialogs and are usually document-centric. Examples include ClarisWorks, Quicken, and Microsoft Excel. There are three variants of this application type: Mac OS 8–savvy, minimal-adoption, and Mac OS 8–compatible.

Mac OS 8–savvy applications. A *Mac OS 8–savvy application* is just what you'd think it would be: an application that takes significant advantage of Mac OS 8 features, such as preemptive tasking, improved event delivery, and new user interface features. Because these APIs aren't available under System 7, a Mac OS 8–savvy application will not run under pre–Mac OS 8 system releases. Note that Mac OS 8–savvy applications still must maintain a cooperatively scheduled task so that they can call the Macintosh Toolbox, and this cooperative task lives in the same address space with all other System 7 and Mac OS 8–savvy applications. It's possible to have other portions of the application run in separate address spaces, and servers (such as file sharing) are completely protected from applications.

Minimal-adoption applications. Not all developers may want to or be able to make the move to a Mac OS 8–only application immediately. The *minimal-adoption application* type is intended for these situations. The characteristics of this category include "fitting in" with the Mac OS 8 look and feel while still maintaining compatibility with System 7. Some subset of Mac OS 8 features (not APIs) will be available to these applications, such as the removal of the fixed-size Memory Manager heap limitation, and the new-look, theme-specific, user interface elements. This application type is distinguished from System 7 applications by its correct appearance under Mac OS 8 and its adoption of Mac OS 8 features on a runtime-check basis.

Mac OS 8–compatible applications. Well-written applications authored originally for pre–Mac OS 8 systems will continue to work under Mac OS 8. If you follow the guidelines in this article and adhere to documented Macintosh application programming practices, your application should be Mac OS 8 compatible. Of course, to take advantage of Mac OS 8 features in your application, you may need to release a new version.

REAL-TIME APPLICATIONS

Real-time applications are applications that have time constraints on aspects of their behaviors. If these constraints aren't met, the application either fails or needs to adapt gracefully to these operating conditions. Examples include data collection applications such as LabView, multimedia applications such as Premier, and games such as Marathon. Under Mac OS 8, real-time applications may choose to take advantage of enhanced timing services and preemptive scheduling to improve their performance.

In some ways, however, Mac OS 8 provides new challenges to these applications, since virtual memory is always enabled and preemptive scheduling may cause the application to lose control of the CPU in unforeseen situations. That said, performance will be vastly improved from the System 7 Virtual Memory Manager, so this shouldn't be a concern for most developers.

OPENDOC PART EDITORS

OpenDoc part editors are a relatively new application type. OpenDoc will continue to be an important direction for document-oriented applications under the Mac OS 8 release. As you'll see throughout this article, OpenDoc is also a suggested migration path for several types of existing applications.

SERVERS

Servers are a new concept introduced in Mac OS 8. They're preemptively scheduled and run in their own protected address spaces. These features provide independence from the cooperative Toolbox environment and mean that servers have greatly enhanced stability, surviving the crashes of applications or other, ill-behaved servers. New reentrant services are provided in Mac OS 8 to make servers possible, including tasking, messaging, memory allocation, file system access, and networking. Only a subset of the Mac OS 8 APIs are available to servers, and this subset does not include the Macintosh Toolbox calls (Window Manager, Menu Manager, QuickDraw, and so on). This means that servers cannot present a user interface. Candidates for servers include World Wide Web Internet servers, virus checkers, and high-end publishing print servers.

UTILITY APPLICATIONS

Utility applications manage a single window for user interface and have no menu bar. To the user, they aren't usually considered a separate application. Mac OS 8 will support utility applications as a generalized application type, unlike previous system releases. An additional use of utility applications is to present a user interface on behalf of a server. For example, a utility application could be used to configure an e-mail forwarding server with the proper e-mail addresses. Other examples of this application type include Apple Guide and configuration control panels.

OTHER NON-APPLICATION CLASSIFICATIONS

Three other classifications that are useful to our discussions but don't refer to applications are extension libraries, patch libraries, and drivers.

Extension libraries. *Extension libraries* allow additional code to be introduced into the Code Fragment Manager (CFM) closure for an application. An example use of an extension library would be to track software launches and quits through CFM initialization and termination routines to perform software auditing.

Patch libraries. *Patch libraries* allow patches to be installed into applications through data-driven means, simplifying the customization process. Patch libraries apply their patches in only one context, but can be combined with extension libraries to achieve global-effect patching. The Get/SetTrapAddress methods of the past have proven difficult or impossible to maintain and have resulted in greatly decreased system reliability. The patch library mechanism, with associated extension libraries, provide a more than capable replacement for the pre–Mac OS 8 patching methods.

Drivers. Under Mac OS 8, the device driver mechanism has been rearchitected to ensure a high-throughput and flexible I/O system. Pre–Mac OS 8 'DRVR'-style drivers are not supported and so need to be rewritten. As we'll discuss below, some software written as a driver today may be better written as another application type.

MIGRATION PATHS FOR EXISTING APPLICATIONS

The first step in preparing for Mac OS 8 is determining the migration path your application will follow. Depending on what type of application you have today, this could mean a complete rewrite, minor tweaks, or no changes at all.

The most important point along the Mac OS 8 migration path is the first one: compatibility. Before determining the best way of moving your application forward, you should ensure that it runs out of the box on Mac OS 8. We'll discuss compatibility specifics in a later section.

Each System 7 application type has a unique migration path; the sections below cover the available options.

USER INTERFACE APPLICATIONS

As shown in Figure 1, a System 7–based user interface application has several alternatives for migration. The simplest alternative is not to revise the application at all, or, if needed, do the minimum required to make the application Mac OS 8 compatible.

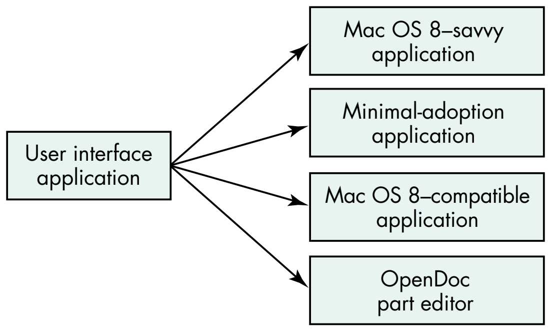

User interface application
Mac OS 8–savvy application
Minimal-adoption application
Mac OS 8–compatible application
OpenDoc part editor

Figure 1. The migration path for a user interface application

Another alternative is to convert the application into an OpenDoc part editor. I'm not going to go into OpenDoc in this article; if you choose this route, see the article "The OpenDoc User Experience" in *develop* Issue 22 for a good overview of how part editors work from the user's perspective.

If having a single binary for both System 7 and Mac OS 8 is important, the minimal-adoption option may be appropriate. By migrating to minimal adoption, you ensure user interface consistency and may be able to take advantage of a limited number of Mac OS 8 features. For example, more efficient memory management is possible if you restrict yourself to a subset of the Macintosh Memory Manager API and don't access Memory Manager heap structures directly.

The most ambitious path for migration is to make the application Mac OS 8 savvy. This will mean that the same binary won't run under both System 7 and Mac OS 8. If the application is made Mac OS 8 savvy, it can take advantage of the wide range of preemptive tasking services, efficient event handling, and an object-oriented version of the Macintosh Toolbox.

REAL-TIME APPLICATIONS

The migration path for a real-time application is more straightforward than for a user interface application (see Figure 2). Either the application can move to Mac OS 8 without changes or it can take advantage of additional real-time features provided by

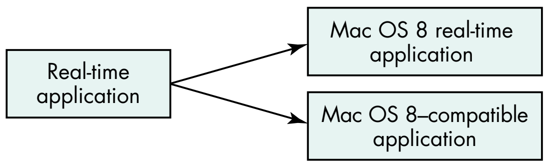

Real-time application
Mac OS 8 real-time application
Mac OS 8–compatible application

Figure 2. The migration path for real-time applications

Mac OS 8, becoming a Mac OS 8–specific real-time application. Which path is taken depends on whether the existing application performs properly under Mac OS 8 without changes. For example, the developer of an application that was using Time Manager tasks for accuracy-critical timing would want to consider migrating to take advantage of the improved timing services of Mac OS 8.

OPENDOC PART EDITORS

As shown in Figure 3, existing OpenDoc part editors will be compatible with Mac OS 8, but a part editor can be updated to take advantage of new Mac OS 8 APIs if the developer chooses. Both Mac OS 8–savvy and existing OpenDoc part editors can work on the same OpenDoc documents without problems.

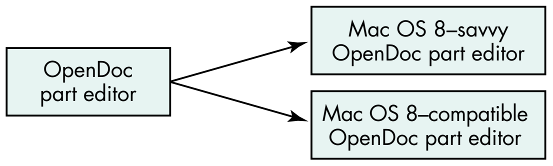

OpenDoc part editor
Mac OS 8–savvy OpenDoc part editor
Mac OS 8–compatible OpenDoc part editor

Figure 3. The migration path for OpenDoc part editors

FACELESS BACKGROUND APPLICATIONS

Faceless background applications (FBAs), also known as background-only applications, are supported under Mac OS 8, but the natural migration path for most FBAs leads to the server application type (see Figure 4).

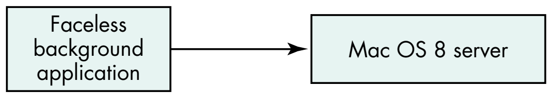


Figure 4. The migration path for faceless background applications

The preemptive nature of servers makes them easier to write than FBAs, since file system and networking calls can block until completion instead of being written with asynchronous calls and chained completion routines. More important, the enhanced reliability of servers makes this transition an easy decision. One drawback of servers is that they cannot access the cooperative Toolbox environment, since they're preemptively scheduled.

DESK ACCESSORIES AND CONTROL PANELS

We'll cover desk accessories and control panels together, since they share many of the same user interface characteristics. Most user interaction for these types takes place in a single window, and neither type maintains a full menu bar. All existing desk accessories and control panels are unsupported under Mac OS 8 and so need to be rewritten. As shown in Figure 5, the two suggested replacement application types are utility applications and OpenDoc part editors.

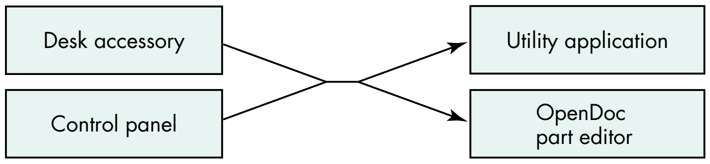


Figure 5. The migration path for desk accessories and control panels

DRIVERS

Figure 6 shows the migration path for drivers. On pre–Mac OS 8 systems, drivers were typically written for a variety of reasons. The most straightforward use of old-style drivers was to control hardware devices. With Mac OS 8, a newly designed driver architecture has been provided for these hardware control drivers.

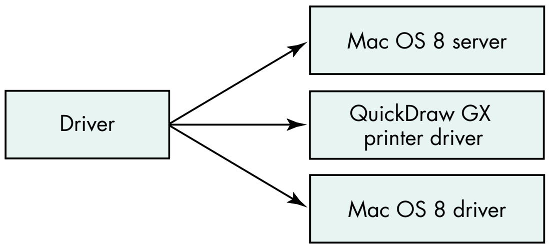


Figure 6. The migration path for drivers

Another common reason for writing a driver was to get periodic time from the system, or to present a common interface for other drivers or applications through the PBControl call. Mac OS 8's server mechanism is a much more capable solution for these types of products. Servers can get reliable periodic time through the Mac OS 8 kernel's timing services, and messaging or Apple events can be used to communicate with other servers or applications.

Certain existing drivers will be compatible with Mac OS 8. Display and networking drivers developed in strict accordance with the guidelines in *Designing PCI Cards and Drivers for Power Macintosh Computers* will work under Mac OS 8 without modification.

Finally, non–QuickDraw GX printer drivers are not supported under Mac OS 8. Since the printing mechanism for Mac OS 8 is an improved version of QuickDraw GX, existing non-GX printer drivers need to be updated to the QuickDraw GX printer driver model.

EXTENSIONS

Extensions, in the form of both individual INITs and INITs packaged within control panels, are not supported under Mac OS 8. These extensions, for the most part, can be divided into two categories: those that need periodic time from the system and those that operate through global-effect patching. The former will typically be replaced by servers, while the latter will migrate to either the built-in extensibility services or the extension library mechanism (as shown in Figure 7).

For extensions that patch traps such as SystemTask to get periodic time to run "on the dime" of other applications, the Mac OS 8 server model provides a much more straightforward mechanism. The only drawback to using servers in this way is that the cooperative Toolbox environment is not available to preemptive callers, so an old-style faceless background application may be another migration alternative.

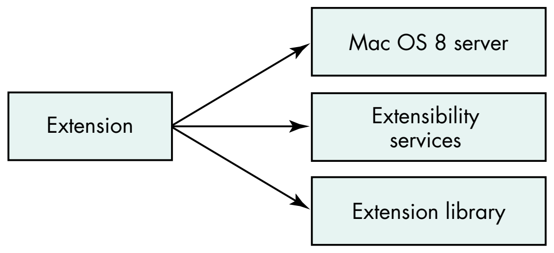


Figure 7. The migration path for extensions

As mentioned earlier, hooks for extensibility have been designed into the Mac OS 8 system. For this reason, patching is no longer necessary in most cases. These built-in extensibility hooks should accommodate most former clients of patching.

Since we can't forecast every possible way of extending the system, the extension library and patch library mechanisms have been provided. Through data-driven means, patches can be installed on a per-application basis to control behaviors. Super Boomerang, for example, could be rewritten in this way.

Some items located in the System 7 Extensions folder will continue to work under Mac OS 8. These include CFM shared libraries, 'appe' background applications, Communications Toolbox tools, Chooser extensions, and Apple Guide guide files. Note that no INITs associated with these extension types will be loaded by Mac OS 8, which could impact compatibility for these products.

Several types of non-INIT Apple extensions are not supported, however, and so need to be rewritten. These include sound sifters, inline input methods, and FSM modules.

APPLICATION PLUG-INS

By application plug-ins, we're referring to products like Kai's Power Tools for Photoshop. There shouldn't be any compatibility issues specific to plug-ins operating in existing applications. There are, however, compelling reasons for application developers to update the way they present plug-in interfaces. Figure 8 shows the migration path. For applications that present their plug-in model in an object-oriented fashion, the System Object Model (SOM) should be used. Its many advantages include strong type checking, object-oriented interfaces, and a solution to the fragile base-class problem. For developers who want to maintain a functional interface, the Code Fragment Manager is the best option.

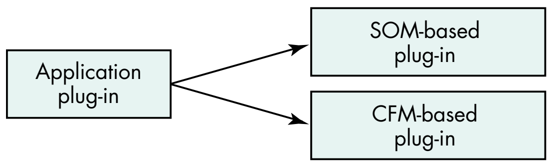


Figure 8. The migration path for application plug-ins

There are some potential pitfalls related to plug-in compatibility with Mac OS 8. Although mixing Mac OS 8 APIs with System 7 APIs is permitted, there may be problems running System 7 plug-ins with Mac OS 8–savvy versions of applications. For instance, Photoshop could choose to take advantage of new memory management services provided in Mac OS 8. For the payoff of not having a fixed-size Memory Manager heap, they promise not to make certain Memory Manager calls. If a System 7

Photoshop plug-in makes any of these disallowed Memory Manager calls, Photoshop may not work.

For this reason, we're suggesting that all applications that support a plug-in model provide isolation for these plug-ins from system calls. If the plug-in needs to allocate memory, for example, it should call the application to do so, instead of calling the Memory Manager directly. In this way, plug-ins can take advantage of new Mac OS 8 features without ever being updated.

Several existing plug-in mechanisms are either not available or not encouraged for use under Mac OS 8. The Apple Shared Library Manager, as we'll discuss later, is not available. The Component Manager is available but is not recommended for use, since the Code Fragment Manager and SOM are now the preferred mechanisms.

PREPARING FOR MAC OS 8 TODAY

There are quite a few steps you can take to ensure application compatibility with Mac OS 8. You've heard a lot of these suggestions before, while some are new. All are important techniques that can save you many hours of debugging.

You'll find that in the Mac OS 8 release, a debugging version of the system has been provided that will assert via the debugger whenever your application performs a questionable behavior. Hopefully, if you follow the guidelines in this article, you won't ever have to see these assertions.

For each suggestion, we'll show a snippet of code or an example of the technique where appropriate. Each one of these techniques can be put into action today.

USE THE LATEST UNIVERSAL INTERFACES

Within Apple, all Macintosh development is now done using the same set of universal interfaces. These interfaces are periodically released to developers on the MPW Pro disk and online, and are also available on this issue's CD. Some third-party tools, such as Metrowerks CodeWarrior, also ship with the latest interfaces. By using these interfaces, you'll be developing with the same C, Pascal, and assembly headers as Apple engineers, which ensures that you'll be up to date with the latest changes from Apple.

Over time, the universal interfaces will have Mac OS 8 features conditionally added. For example, several compile-time switches will be added to indicate which calls are available in certain situations.

PORT YOUR APPLICATION TO POWERPC CODE

Mac OS 8 is, at heart, a PowerPC processor–based system. Porting only performance-critical sections of your application to PowerPC code and leaving the rest in 680x0 code will begin to become more of a liability than an advantage under Mac OS 8. New API calls introduced in Mac OS 8 will not have traps associated with them. They'll be made available only through CFM and SOM-based interfaces. This means that 680x0 code will not be able to access these new services. Note, however, that 680x0 applications are still fully supported under Mac OS 8 for compatibility.

USE A SUPPORTED FRAMEWORK

Considering all the changes coming for developers in the Mac OS 8 release, you may choose to move your application to a well-supported Macintosh framework. Apple is already working closely with several framework providers, and you may decide to take advantage of their efforts. One word of caution is that using a framework doesn't guarantee compatibility. If that framework "breaks the rules," or if your own application code uses unsupported behaviors, you still have compatibility concerns.

SUPPORT ONLY SYSTEM 7 AND LATER

While getting a variety of applications running on the Mac OS 8 release, we've found that many developers have obsolete code buried in their applications. For instance, several developers used the code shown in Listing 1 to check for the availability of certain system traps. Note the function ToolboxTrapTableSize. This code checks to see if the application is running on a Macintosh with an expanded Toolbox trap table — but this has been the case ever since Color QuickDraw was introduced. Considering that the applications performing this check were PowerPC native, this check is overkill and so can be removed.

Listing 1. Obsolete code for checking trap availability

```
Boolean IsTrapAvailable(short theTrap)
{
   TrapType    trapType;
   Boolean     available;

   if ((theTrap & 0x0800) > 0)
      trapType = ToolTrap;
   else
      trapType = OSTrap;
   if (trapType == ToolTrap) {
      theTrap &= 0x07FF;
      if (theTrap >= ToolboxTrapTableSize())
         theTrap = _Unimplemented;
   }
   available = NGetTrapAddress(theTrap, trapType)
      != GetToolTrapAddress(_Unimplemented);
   return available;
}

short ToolboxTrapTableSize(void)
{
   if (GetToolTrapAddress(_InitGraf) == GetToolTrapAddress(0xAA6E))
      return 0x0200;
   else
      return 0x0400;
}
```

Other examples of long-obsolete behaviors include using old SFGetFile-style Standard File calls and relying on HFS working directories to make file system calls. In other words, you can safely assume that Apple will not introduce a PowerPC processor–based Macintosh that runs System 6.

MINIMIZE PATCHING

Many applications use patching to excess. A well-written PowerPC-native application should not have to patch any traps. Along the lines of removing old code, consider removing patches installed simply to work around a long-fixed bug, at least conditionally.

We found a particularly bad example of application trap patching when bringing up a major word processing application. The application called the Alert routine to display an alert to the user. The developers decided that they wanted a cool 3D button instead of the Macintosh-standard button, so they patched NewControl and watched for NewControl to be called from NewDialog (itself called from Alert) with the expected pushButProc procID. When this call was intercepted, they substituted their 3D button procID, and the alert was displayed with their button. Of course, they could have simply called the Dialog Manager and Control Manager themselves, thereby avoiding the trap patch entirely. There's no law that says you need to call Alert, after all.

We can't stress enough the importance of patch minimization. Relying on side effects that are undocumented, such as the fact that Alert will end up calling NewControl, may cause your application to break unexpectedly with any new system release. We found out about the above example because the Mac OS 8 Dialog Manager used its new modern mechanisms within Alert. To maintain compatibility with this application, we had to back off from this and revert to the existing mechanisms. In many key areas, Apple can't innovate as much as developers would like because of the behaviors of many existing applications.

FACTOR YOUR APPLICATION

As has been common for some time with cross-platform development, it's a useful exercise to separate your application into several distinct parts in preparation for Mac OS 8. At least two of these parts should be the user interaction component and the compute engine component. Separating all Macintosh-specific calls such as disk access and networking into modules may also be helpful.

Mac OS 8 provides several new facilities to make factoring your application easier. Tasking and synchronization services provided by the system allow you to divide computation into several tasks, resulting in greatly improved CPU usage. Unlike the Thread Manager, Mac OS 8's new tasking services provide preemptive multitasking. In addition, the Apple event mechanism has been significantly improved in both performance and functionality. The Apple Event Manager can be called by both preemptive and cooperative tasks, and is the preferred method of communication among an application's factored tasks.

The reasons for factoring your application are twofold. First, the separation will allow you to more easily bridge the gap between your Mac OS 8 source base and your System 7 source base. You could separate your core functionality (such as an image processing algorithm) from your user interface code, thereby allowing you to write a new Mac OS 8–savvy user interaction module. Or, under Mac OS 8, you could choose to run your image processing module preemptively in another task. This modularity could make it possible to substitute Mac OS 8–specific file system calls, for example, to achieve better throughput.

The second reason has a more immediate payoff. Factoring your application will help make it scriptable and recordable with AppleScript. With Mac OS 8, the event routing mechanism will change from a polling to a delivery mechanism, with high-level synthetic events being produced from lower-level user actions. This new model is just a short step from a System 7 AppleScript-recordable application.

USE STANDARD DEFINITION PROCEDURES WHEREVER POSSIBLE

With Mac OS 8, the Macintosh user experience will get a major facelift. Any number of user-selectable themes can be chosen to alter the appearance of windows, menus, and controls. Some examples of theme-specific windows are shown in Figure 9.

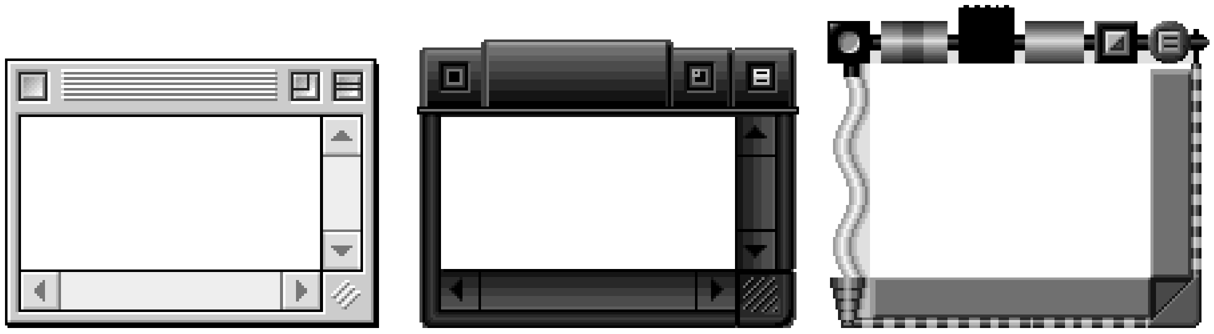


Figure 9. Theme-specific windows

As you can see, the appearance of themes can vary greatly. The big compatibility challenge here is that many developers have lost patience with Apple in the area of user interface look enhancements, and have implemented their own unique application appearances already. A user who switches the theme on his or her Macintosh will expect the appearance of all applications to change. The new Apple-supplied theme-specific appearances are tied to Apple-standard Toolbox definition procedures. If an application developer uses a custom 3D button or checkbox, it will look the same no matter which theme is selected. Needless to say, this can create combinations no graphic artist would ever approve of.

To prevent this situation, we're suggesting that you avoid using custom WDEFs, CDEFs, and MDEFs wherever possible, and if you must use them for competitive reasons, give the user the option of turning them off to revert to the system appearance. We realize that there are some cases where Apple doesn't provide a look that meets your needs, and in these cases custom definition procedures will still need to be used. Apple has provided some additional definition procedures recently, such as the floating palette window WDEF 124 with System 7.5 (note that this WDEF doesn't provide the floating behavior, only the correct appearance).

USE OPEN TRANSPORT WHEN AVAILABLE

Open Transport is the native networking stack for Mac OS 8. The traditional Device Manager AppleTalk and MacTCP calls are still supported for backward compatibility, but for maximum networking performance with Mac OS 8, Open Transport should be used directly. By adopting a factored approach to your application, you should be able to support both Open Transport and the traditional APIs in a single binary.

SUPPORT QUICKDRAW GX PRINTING

The native printing implementation for Mac OS 8 is based on QuickDraw GX, although the non-GX printing API is supported for compatibility. This means that your application is much more likely to print faster and more reliably if you support the QuickDraw GX printing API. Again, it's fairly easy to maintain support for both traditional and QuickDraw GX print loops — and again, traditional printing calls are supported for backward compatibility. (See the article "Adding QuickDraw GX Printing to QuickDraw Applications" in *develop* Issue 19.)

USE THE LOW-MEMORY ACCESSORS

With the advent of the Power Macintosh, new calls were added to provide access to supported low-memory locations. By migrating to the LMGet/LMSet accessor functions today, you can be assured that you aren't relying on any undocumented low-memory globals. The existing LMGet/LMSet calls will eventually be migrated to the individual owner components, where they'll be made into full-fledged API calls, and their connection with low memory will be broken. Note that in Mac OS 8, the LMGet/LMSet calls still do access the global low-memory area.

BE VIRTUAL MEMORY FRIENDLY

In the instruction manuals of many popular Macintosh software applications, you'll find the directive to turn virtual memory off for optimal performance. This is not an option for Mac OS 8, but fortunately the memory usage of the system and the virtual memory architecture have been substantially improved for much better performance. Given that virtual memory is always on, competing with it by designing your own virtual memory system would be unwise. If you have your own memory management system, count on doing some performance tuning once you get your application up and running.

LOCATE SPECIAL FOLDERS WITH FINDFOLDER

This tip is along the lines of the "System 7 and later" directive above. FindFolder, which has been available since System 7, can be used to locate the System Folder, Preferences folder, Extensions folder, and other system-created folders. A corollary of this guideline is always to store all user-specific preference information in the Preferences folder. By using FindFolder and correctly storing your preferences, you'll be compatible with the workspaces mechanism in Mac OS 8, which allows different system users to have their own sets of application settings.

USE DEBUGGING VERSIONS OF SYSTEM COMPONENTS

Several portions of the system are already available in debugging versions. These special versions will flag questionable behaviors, allowing you to correct problems that would otherwise go undetected. Currently, debugging versions of both the Modern Memory Manager and QuickDraw GX are available for use with System 7.5. These tools should save valuable time in the Mac OS 8 application debugging process.

SPECIFY YOUR STACK SIZE IN THE CODE FRAGMENT RESOURCE

If your application needs additional stack space above and beyond the default stack size, it should use the application stack size field provided in the code fragment ('cfrg') resource. Calls to GetApplLimit and SetApplLimit have no effect on PowerPC-native applications in Mac OS 8. Applications compiled for 680x0 systems should still use these calls to adjust their stack size.

PREPARE FOR MAC OS 8 MEMORY MANAGEMENT

A major problem for Macintosh applications, in terms of both performance and efficiency, is the way that they manage memory. The Macintosh Memory Manager, whose fundamental structure was designed for the original 128K Macintosh, is woefully out of date, especially when used in Mac OS 8's demand-paged virtual memory environment.

To address this problem, the Mac OS 8 designers have provided two new ways to manage memory. For developers who don't want to redesign their memory usage model, a transitional API, based on a subset of the old Memory Manager entry points, is provided. Developers who are fully adopting Mac OS 8 APIs can use a completely new, modern memory management service. You won't have to adopt any new memory management techniques to make your application compatible with Mac OS 8; not adopting them just means that your application won't benefit from memory enhancements such as unbounded application heaps.

To prepare your application for Mac OS 8 from a memory management standpoint, you should avoid certain uses of the Memory Manager. Some steps you can take today are listed below; this list is not exhaustive, and you may need to do additional work to take advantage of the transitional API.

- Don't dispose of pointers and handles that are allocated indirectly by the Toolbox. For example, don't call DisposeHandle on a control allocated with NewControl.
- Don't access handles, pointers, or heap zones outside the application heap or system heap.
- Don't allow application plug-ins to call the Memory Manager directly. Instead, have them call back into the application to manage memory. This way, you can be sure any plug-ins for your application also follow the new Memory Manager requirements.
- Avoid allocating memory in the system heap or in temporary memory. Temporary memory is still supported for compatibility, but the transitional memory API is based on the classic Memory Manager API, which doesn't encompass temporary memory.
- Avoid or abstract the use of the following Memory Manager calls, which will all be unsupported in the transitional memory API: InitApplZone, SetApplBase, InitZone, GetApplLimit, SetApplLimit, MaxApplZone, MoreMasters, NewHandleSys, NewHandleSysClear, NewEmptyHandleSys, HandleZone, RecoverHandle, NewPtrSys, NewPtrSysClear, PtrZone, FreeMem, MaxMem, CompactMem, ReservMem, PurgeMem, TopMem, GrowZoneProcs, and PurgeProcs.

UNSUPPORTED BEHAVIORS UNDER MAC OS 8

We're making every effort to maintain compatibility with existing applications for the Mac OS 8 release, but unfortunately we're not able to support some behaviors that may work under current system releases. We're trying to get the word out early on these unsupported behaviors so that developers have plenty of time to correct the problems.

DON'T USE ASLM

The Apple Shared Library Manager is not available under Mac OS 8. Applications that rely on ASLM as a shared library mechanism or to maintain plug-ins will need to be redesigned. If an object-oriented shared library mechanism is still required, SOM should be used; in cases where object-oriented interfaces aren't necessary, the Code Fragment Manager can be used.

DON'T ACCESS THE TRAP TABLE DIRECTLY

Some applications access the trap table directly, either to call traps without going through the trap dispatcher or to apply patches. This is no longer allowed, since under Mac OS 8 the trap table is no longer maintained by the same mechanism. Attempts to write to or read from the trap table directly will not produce the expected results. An example of code that won't work is shown in Listing 2.

DON'T USE A GNEFILTER TO INTERCEPT EVENTS GLOBALLY

A common programming technique under System 7 is to install a GetNextEvent filter procedure into the GNEFilter low-memory location (0x29A). This behavior is documented in the Macintosh Technical Note "GetNextEvent; Blinking Apple Menu" (TB 11). Mac OS 8 supports the GNEFilter mechanism on a per-context basis only. The code in Listing 3 shows a sample GNEFilter faceless background application that beeps whenever a key is pressed. Under Mac OS 8, this code would not work, since background applications don't receive key-down events, and a GNEFilter

Listing 2. Unsupported trap table access

```
// UNSUPPORTED!
void PatchSysBeep(void)
{
   long  trapTableBase, sysBeepOffset, *sysBeepAddress;

   (void) Gestalt(gestaltToolboxTable, &trapTableBase);
   sysBeepOffset = (_SysBeep & 0x03ff) * sizeof(long);
   sysBeepAddress = (long *) (trapTableBase + sysBeepOffset);
   *sysBeepAddress = (long) MySysBeep;
}
```

Listing 3. An unsupported GetNextEvent filter procedure

```
// UNSUPPORTED!
GNEFilterUPP   gOldGNEFilter;
Boolean        gContinue;

void main(void)
{
   EventRecord ev;

   gContinue = true;
   gOldGNEFilter = LMGetGNEFilter();
   LMSetGNEFilter(NewGetNextEventFilterProc(MyEventFilter));
   while (gContinue)
      WaitNextEvent(everyEvent, &ev, 60, NULL);
   LMSetGNEFilter(gOldGNEFilter);
   ExitToShell();
}

void MyEventFilter(EventRecord *theEvent, Boolean *result)
{
   if (theEvent->what == keyDown) {
      SysBeep(1);
      if (theEvent->modifiers & controlKey) {
         gContinue = false;
      }
   }
   CallGetNextEventFilterProc(gOldGNEFilter, theEvent, result);
}
```

installed by a particular application is called only for events received by that application.

DON'T CALL PPOSTEVENT

Some pre–Mac OS 8 applications post fake keyboard or mouse events into the event queue with PPostEvent. The call works by posting an empty event into the application's event queue and returning a pointer to that queue element. PPostEvent is not supported under Mac OS 8, since the Event Manager no longer maintains the event queue in this way. The code in Listing 4 shows a typical use of PPostEvent.

Listing 4. An unsupported PPostEvent call

```
// UNSUPPORTED!
void PostMouseDown(Point *mouseDownPoint)
{
   EvQElPtr eventQueueElement;

   (void) PPostEvent(mouseDown, 0, &eventQueueElement);
   eventQueueElement->evtQWhere = *mouseDownPoint;
}
```

DON'T ACCESS PRIVATE TRAPS OR PRIVATE LOW MEMORY

As we've been warning developers for years, relying on private traps and private low-memory globals can cause compatibility problems for applications. All private traps have been removed from the Mac OS 8 trap table, since the system software no longer calls other services via the trap mechanism. This means that any emulated application that calls a private trap, or any native application that calls GetTrapAddress on a private trap and then CallUniversalProc on that address, will break under Mac OS 8. (See Listing 5.)

Listing 5. Calling private traps in an unsupported way

```
// UNSUPPORTED!
#if GENERATING680x0
   extern pascal OSErr InitDogCow(short moofCount)
      ONEWORDINLINE(0xA89F);

   void main(void)
   {
      (void) InitDogCow(2);
   }

#else
   enum {
      uppInitDogCowProcInfo = kPascalStackBased
         | RESULT_SIZE(SIZE_CODE(sizeof(OSErr)))
         | STACK_ROUTINE_PARAMETER(1, SIZE_CODE(sizeof(short)))
   };

   void main(void)
   {
      UniversalProcPtr  initDogCowUPP;

      initDogCowUPP = GetToolboxTrapAddress(_InitDogCow);
      (void) CallUniversalProc(initDogCowUPP, uppInitDogCowProcInfo, 2);
   }
#endif
```

DON'T LINK AGAINST PRIVATEINTERFACELIB

Enterprising Macintosh hackers have realized that there's a CFM library on the PowerPC processor–based Macintosh systems named PrivateInterfaceLib. This library contains CFM entry points for private system calls. Under Mac OS 8, PrivateInterfaceLib no longer exists. Applications that link against PrivateInterfaceLib will not work under Mac OS 8.

DON'T ACCESS CERTAIN PUBLIC LOW-MEMORY GLOBALS
Certain public low-memory globals are no longer supported under Mac OS 8. If a low-memory global is no longer supported, accessing it should be benign. For example, the low-memory global AuxWinHead is no longer used by the Window Manager, so calling LMGetAuxWinHead will return NULL.

DON'T RELY ON FILE SYSTEM HOOKS OR PATCHES
The existing file system hooks ExtFSHook and FSQueueHook are not supported under Mac OS 8. In addition, patching the existing API as documented in *Inside Macintosh: Files* is allowed from within an application, but the patch will only affect file system calls made from within the cooperative task of the application.

DON'T USE COMPRESSED RESOURCES
Compressed resources may not be supported by the Mac OS 8 Resource Manager, so your application and its files should not contain any compressed resources. The resource compression mechanism has never been public and should be avoided.

DON'T ALLOCATE ALL OF TEMPORARY MEMORY
Since Mac OS 8 is a demand-paged system, it grows the virtual memory space as necessary to accommodate additional applications or memory allocations. For this reason, asking how much temporary memory is available and then allocating all of it, or all but a certain amount, is a bad idea. This would just consume all your available disk space, giving you far more memory than you wanted.

DON'T ALLOCATE ALL HARD DRIVE SPACE
Similarly, allocating all hard drive free space is a bad idea, since you may be consuming all possible storage for virtual memory swap space, and this will immediately lead to major system problems.

DON'T SHARE TOOLBOX STRUCTURES BETWEEN APPLICATIONS
With the Mac OS 8 release, Apple is beginning to partition applications from one another. Although applications still run in the same address space, the individual Mac OS 8 managers maintain structures on a per-application basis. For this reason, creating a menu, for example, in one application and attaching it to the menu bar in another application will not work correctly under Mac OS 8. The same is true for other system structures, such as windows, dialogs, controls, resource refNums, file refNums, and working directory IDs, among others. Memory Manager structures such as handles and pointers can still be shared across applications.

DON'T HARD-CODE FONT USAGE
Mac OS 8 allows users to customize the system's appearance. Among the settings they control is which font to use for the system font and for the application font. For this reason, applications should use the calls GetSysFont and GetAppFont instead of hard-coding font selections (to, for example, Chicago 12).

DON'T WRITE TO YOUR APPLICATION'S DATA FORK
Many existing applications prompt users for their name and serial number the first time the application is launched. Several PowerPC-native applications store this information directly into the application's data fork, while the application is running.

Under pre–Mac OS 8 systems, this is allowed because of a loophole in the system software. Under Mac OS 8, PowerPC-native applications are always file-mapped read only with exclusive access. This means that an application will get an error from the file system when it tries to open its data fork with write access. Instead, applications should store their personalization information in either a resource or a preferences file. If this information must be stored in the data fork, it will need to be written by another application, such as the Installer.

DON'T ACCESS HARDWARE DIRECTLY

Some applications access hardware directly under pre–Mac OS 8 systems. With Mac OS 8, only applications executing in supervisor mode, such as drivers, may access hardware. This limitation greatly improves system stability. An example of unsupported hardware access is an application that accesses the GPI serial pin directly to detect phone rings.

DON'T ASSUME THE SYSTEM STATE IN NOTIFICATION MANAGER COMPLETION ROUTINES

Some developers have noticed that when a Notification Manager completion routine is called, they can make certain assumptions about the state of the system. For example, they assume that the routine will always be called in the context of the frontmost application, and they either access that application's window list or create a window or dialog of their own (as shown in Listing 6). Under Mac OS 8, these assumptions are no longer true.

DON'T CHANGE THE WINDOW LIST DIRECTLY

Before Mac OS 8, the Window Manager didn't support floating windows. Some developers implemented floating windows by manipulating the window list directly rather than calling the Window Manager API. Under Mac OS 8, the Window Manager maintains the window list separately from the nextWindow field, so changing this field directly will cause problems. Instead, BringToFront and SendBehind should be used to maintain the window list. See the article "Floating Windows: Keeping Afloat in the Window Manager" in *develop* Issue 15 if you need to maintain floating windows on pre–Mac OS 8 systems.

DON'T SET THE GLOBAL SHARE BIT IN A CFM LIBRARY THAT CONTAINS CODE

Linking against a per-context library from within a global library can create problems for that per-context library. The per-context library may end up with a different copy of its per-context globals than it expects. To avoid this problem, you should never set the global share bit in any CFM library that contains code. If you need to maintain systemwide global data, you should do so in a separate library that only contains data and doesn't link against any other libraries.

With the initial Power Macintosh system software, the InterfaceLib CFM shared library had its data section globally shared across all applications. Under Mac OS 8, however, the notion of CFM per-context data is used widely by all system components. For example, the File Manager tracks which files each application has opened via per-context globals in the Files shared library.

The global share bit is accessible in Metrowerks CodeWarrior from the "share data section" checkbox in the PPC Pef preferences pane, and from the **-s** option in the MakePef MPW tool.

DON'T RELY ON THE STRUCTURE OF SYSTEM MEMORY

Relying on information such as where certain code is loaded or where memory is allocated may cause compatibility problems under Mac OS 8. The relative locations

Listing 6. Unsupported Notification Manager hacking

```
// UNSUPPORTED!
NMRec    gNMRec;

void PostWindowInFrontApplication(void)
{
   gNMRec.qType = nmType;
   gNMRec.nmMark = 0;
   gNMRec.nmIcon = NULL;
   gNMRec.nmSound = NULL;
   gNMRec.nmStr = NULL;
   gNMRec.nmResp = NewNMProc(HackResponseRoutine);
   gNMRec.nmRefCon = GetCurrentA5();
   (void) NMInstall(&gNMRec);
}

pascal void HackResponseRoutine(NMRecPtr nmReqPtr)
{
   long        savedA5;
   WindowPtr   aWindow;
   Rect        bounds;
   Str32       windowTitle;

   (void) NMRemove(nmReqPtr);

   savedA5 = SetA5(nmReqPtr->nmRefCon);
   BlockMoveData("\pSurprise!", windowTitle, 10);
   SetA5(savedA5);
   bounds.top = bounds.left = 40;
   bounds.bottom = bounds.right = 350;
   aWindow = NewWindow(NULL, &bounds, windowTitle, true, documentProc,
      (WindowRef)-1L, true, 0);
}
```

of the system heap, application heap, and application stacks may change. For example, if you assume that loaded CFM data will appear in your application heap, and check a pointer to that data against the heap's boundaries, your application will not work properly.

DON'T RELY ON AOCE INTERFACES

Not all currently available AOCE interfaces may be available under Mac OS 8. Because of this, if you use AOCE in your application, you should be sure to weak-link against the AOCE library, and check Gestalt for the availability of any AOCE features.

SUPPORTED BUT DISCOURAGED BEHAVIORS

Some questionable techniques are still supported under Mac OS 8 to maintain compatibility with existing applications. Although supported, they're discouraged from use and may not continue to be compatible with future system software releases.

There's also a new concept in Macintosh programming called *deprecation*, whereby certain API calls are discouraged from use in preparation for their future removal; see "Deprecation" for more about this.

DEPRECATION

In many areas of the Mac OS 8 system, new API calls have been provided to replace existing mechanisms. For example, the files API described in *Inside Macintosh: Files* was designed with the old pre–Mac OS 8 interrupt model in mind and does not translate well to usage in a preemptive environment. For this reason, a new File Manager API has been introduced for Mac OS 8. The old files API is still available, but it's *deprecated*, meaning that it's no longer the preferred way of performing file system operations. In fact, new technical documentation that replaces the old *Inside Macintosh* will not include references to deprecated calls.

In the universal interfaces, several preprocessor definitions can be used to control API availability. By setting either of these first two, your application can still access functions that have been deprecated:

- BUILDING_FOR_SYSTEM7 — Code is intended to run on a System 7 machine or earlier.
- BUILDING_FOR_SYSTEM7_AND_SYSTEM8 — Code is intended to run on System 7 or Mac OS 8.

Setting either of the following means that deprecated calls will not be available to your application:

- BUILDING_FOR_SYSTEM8 — Code is intended to run on Mac OS 8 only.
- BUILDING_PREEMPTIVE_CODE — Code is intended to run as a Mac OS 8 server or driver.

DRAWING TO THE SCREEN DIRECTLY

Applications that write directly to the base address of the screen will continue to work properly under Mac OS 8. As before, ShieldCursor and ShowCursor should be used to ensure that the mouse pointer isn't overwritten; for more on this, see the Graphical Truffles column in *develop* Issue 11. Also see the Graphical Truffles column in this issue (Issue 26) of *develop* for some additional warnings about writing directly to the screen.

PATCHING WITHIN AN APPLICATION

Although existing INITs aren't supported, patching within an application is still allowed, via both SetTrapAddress and the new patch library mechanism. As mentioned above in the "Preparing for Mac OS 8 Today" section, you should try to minimize your use of patching, since it lowers overall system reliability and introduces compatibility risks.

ACCESSING MEMORY IN OTHER APPLICATIONS

Some applications may pass pointers to data to each other through Apple events, Gestalt routines, or other means. This is definitely not the recommended way of exchanging data between applications, because in future system software releases each application will run in its own protected address space. However, since under Mac OS 8 all traditional applications (not servers or drivers) run in the same address space, sharing data across applications is supported. If it's absolutely necessary for two applications to share memory, allocating the memory in the system heap will enhance future compatibility.

READING OR WRITING LOW MEMORY DIRECTLY

Some developers aren't using the LMSet/LMGet accessor functions from within their PowerPC-native applications. Since under Mac OS 8 the LMSet/LMGet calls still change low-memory locations, writing and reading directly from low memory is still supported; however, it will stop working in future releases.

USING A CUSTOM MBDF

As stated in *Inside Macintosh: Macintosh Toolbox Essentials* on page 3-87, Apple recommends that you always use the standard menu bar definition procedure. Under Mac OS 8, custom MBDFs are supported to a limited extent only. Your custom MBDF will be called only to process requests not related to drawing. The current theme, as selected by the user, maintains control of the menu bar and menu border appearances.

PATCHING TOOLBOX DEFINITION PROCEDURES

Some developers have discovered that it's possible to customize the behavior of windows, menus, or controls without writing an entire custom WDEF, MDEF, or CDEF. Instead, they write handlers for only the definition procedure messages they want to customize (wDraw for WDEFs, for example) and then call back into the standard definition procedure for all other messages.

Although not in the original spirit of the Toolbox design, this behavior is supported under Mac OS 8. "Stub" definition procedures have been provided for all standard WDEFs, CDEFs, and MDEFs (and the standard MBDF), which call back into the appropriate Toolbox managers to finish the definition procedure processing.

CASE STUDIES IN (IN)COMPATIBILITY

In the process of getting existing applications running on Mac OS 8, we've come across some interesting bugs in shipping applications that may provide insight to other developers. Not all of these bugs prevent the applications from running, but some do.

Of course, Apple cares about its developers, so the names have been omitted to protect the guilty. When we find a bug in an application, we're sure to let the application developer know so that it can be corrected in the next version. In any case, it's interesting what you find when you rewrite the system from scratch.

FAILING TO CALL INITDIALOGS

This application was crashing when it displayed a Standard File dialog. After much debugging, we realized that when the Standard File dialog was displayed, it contained several ParamText strings, even though ParamText had never been called. Further analysis led us to discover that the Dialog Manager had garbage values for the ParamText strings, and accessing these strings caused a crash. We finally determined that the Mac OS 8 Dialog Manager initialized these ParamText strings in InitDialogs, which the application did not call.

FAILING TO CALL INITMENUS

This application always crashed when it tried to draw the menu bar. After some poking around, we figured out that it was assuming that InitWindows called InitMenus as a side effect. The completely new Mac OS 8 implementations of windows and menus no longer have this behavior.

ACCESSING LOCATION 0

This application accessed location 0 by accident, treating it as a pointer to a C string. At startup, due to an obscure bug in their C++ code structure, the developers passed NULL as the parameter to a routine expecting a string. They then tried to copy this string to another buffer, which was 240 bytes in length. Under System 7.5, it just so happens that there's usually a 0 byte in the first 240 bytes of memory (some of the ROM vectors contain 0 bytes). Under Mac OS 8, the low-memory globals area contains, by default, -1 (0xFF) in every location, so there were no 0 bytes to be found. The string copy overwrote the buffer and crashed the machine.

CALLING SETITEMMARK WITH NULL

During launch, this application was accidentally calling SetItemMark on a NULL MenuHandle. It turns out that the developers iterated one item too far in a list of menus. Under System 7.5, there's an undocumented check in the Menu Manager that exits if the menu is NULL. The Mac OS 8 Menu Manager, written to the *Inside Macintosh* spec, didn't have this check. This led to a crash, until the check was added back into the Mac OS 8 code.

CREATING MENUS WITHOUT THE MENU MANAGER

This application cached the contents of the Font menu across launches. It stored this information in a resource, but not one of type 'MENU'. To read the cached information back, it called GetResource on this resource and then called CountMenuItems on the handle. The Menu Manager had no idea that this was a menu, since it wasn't created with either NewMenu or GetMenu.

TREATING ROM85 AS A POINTER

Within a WDEF, this application checked to see which version of the ROM was installed by checking the low-memory global ROM85. Unfortunately, the developers accessed ROM85 as if it were a pointer to a ROM version number, but in reality it's a value, not a pointer. This caused them to dereference 0x3FFF, which on Mac OS 8 was not a mapped memory address.

CALLING OPENSLOTSYNC BY ACCIDENT

This application, which was converted to PowerPC code from 680x0 assembly, called OpenSlotSync when it instead meant to call HOpenSync. The reason for the confusion was that both calls use exactly the same trap word (0xA200) but don't share the same CFM entry point. The 680x0-to-PowerPC converter converted A200 to OpenSlotSync, while the application wanted HOpenSync.

MAC OS 8 COMPATIBILITY STARTS TODAY

Using the techniques and guidelines outlined in this article, application developers should be able to begin working toward 100% Mac OS 8 compatibility immediately. By laying the groundwork for compatibility early, you'll find the preliminary releases of Mac OS 8 much more valuable, and you'll be able to concentrate on adding great new functionality to your software.

Compatibility is one of the critical factors for Mac OS 8's success in the marketplace. We're counting on developers to help us ensure that Mac OS 8 compatibility is not simply a promise, but also a reality.

RELATED READING

- "Copland: The Mac OS Moves Into the Future" by Tim Dierks, *develop Issue 22.*
- *Designing PCI Cards and Drivers for Power Macintosh Computers* (Apple Computer, 1995). This book is available through the *Apple Developer Catalog.*

Thanks to our technical reviewers Ken Bereskin, Paul Black, Jeff Cobb, Sharon Everson, Winston Hendrickson, Matt Mora, and Mike Neil.•

GRAPHICAL TRUFFLES

Dynamic Display Dilemmas

KENT MILLER AND CAMERON ESFAHANI

In the Dark Ages, an application could examine the graphics environment once and gather all the information it needed to know. After the System 7.1.2 Renaissance, the Display Manager made the graphics environment dynamic, which provided many new features (and introduced a few implementation issues). In Issue 24 of *develop*, the Graphical Truffles column described some important features of the Display Manager. Here we'll discuss some common pitfalls that can cause an application to fail in a dynamic display environment — and ways you can overcome them.

If you use QuickDraw routines in an existing application, your application may already support some aspects of the Display Manager without requiring any extra work on your part. An example we touch on in this column is the *graphics mirroring* feature, which allows users to make two graphics devices display the same image. QuickDraw, whose routines have already been updated to support the Display Manager, accomplishes graphics mirroring by overlapping the gdRects (global bounds) of the graphics devices. QuickDraw's internal version of DeviceLoop behaves correctly by detecting when devices overlap, then rendering the image properly for each device. This allows overlapping devices to have different color tables or bit depths and still be imaged correctly.

On this issue's CD, we've included a sample application, SuperFly, which illustrates several techniques you can use to support a dynamic environment in your application. Some of the sample code in this column is excerpted from SuperFly.

COMMON ERRORS

When we were integrating the Display Manager into new system software releases, we encountered some common problems that existing applications had when running in a dynamic display environment. Here's a list of suggestions regarding things that might have worked in the past but won't work now; we'll examine each error in turn and suggest a solution.

- Don't forget to account for mirrored graphics devices when walking the device list.
- Don't assume that just because your application uses only one logical display, it's drawing on only one physical device.
- Don't cache the graphics devices and their state on application startup.
- Don't assume that the menu bar will never move.
- Don't assume that menus will be drawn on only one display.
- Don't draw directly to the screen and bypass QuickDraw without checking for the mirrored case.
- Don't assume that certain 680x0 registers will contain the same values inside a DeviceLoop drawing procedure as when DeviceLoop was called.

Don't forget to account for mirrored graphics devices when walking the device list.

When writing applications in the past, some programmers assumed that graphics devices would never overlap. For example, you might assume that if a certain rectangle is fully contained within a gdRect, it isn't on any other device. To implement highlighting, your application might walk the device list and invert the selection if the global rectangle of what you want to highlight intersects the gdRect of that device. However, when there are two displays with the same gdRect in the device list, the first inversion accomplishes the highlighting but the second inversion restores the highlighted area to the original — unhighlighted — state.

KENT "LIDBOY" MILLER, in a recent attempt to reshape the course of history, has renounced the use of caffeine on a by-minute basis. Lidboy hails from the halls of Apple, where he can be seen pacing in his fuzzy bear slippers. He considers the pinnacle of western culture to have been achieved by the rock group known as Rancid, although he occasionally reads from literary quarterlies on the sly. Were Lidboy to be granted one wish, a side of rice from Taco Bell would no doubt be involved. The single most used word in his vocabulary is "Salsa!"•

CAMERON ESFAHANI (cameron_esfahani@powertalk.apple.com, AppleLink DIRTY) began his career in the spaghetti westerns of Sergio Leone. Industry analysts felt that by going off into that area he would cut himself off from the mainstream and ruin his career, but Cameron felt it was more important to follow his dreams. Now at Apple Computer and looking back, he feels that his spaghetti western period was one of the most exciting and rewarding of his life. He therefore dedicates this column to the memory of Sergio Leone.•

Solution: Use DeviceLoop for your drawing. If you want to write your own version of DeviceLoop for some reason, make sure that it handles overlapping displays. You could solve the inverting problem by designing an algorithm to guarantee that each rectangle in global space is highlighted only once. The MyHiliteRect routine in Listing 1 is an example of a suitable algorithm.

The code in Listing 1 solves the highlighting problem by keeping track of the global area that has been highlighted. When DMGetNextScreenDevice returns a mirrored device (which will already have been highlighted by the first QuickDraw call), the SectRgn will fail and that device will not be highlighted again.

Another solution is given in the sample code in the GDeviceUtilities.cp file on this issue's CD. The function BuildAListOfUniqueDevices builds a list of all graphics devices but eliminates mirrored devices. An application could cache this list and use it for highlighting. However, the list could be invalidated if the user changes the device configuration. The application

Listing 1. Highlighting a global rectangle only once

```
OSErr MyHiliteRect(Rect* hiliteRect)
{
   RgnHandle   hiliteRgn, gdRectRgn, tempRgn;
   GDHandle    theGDevice;

   hiliteRgn = NewRgn();
   if (hiliteRgn == nil)
      return (QDError());
   gdRectRgn = NewRgn();
   if (gdRectRgn == nil)
      return (QDError());
   tempRgn = NewRgn()
   if (tempRgn == nil)
      return (QDError());

   // Make hiliteRect into a region.
   RectRgn(hiliteRgn, hiliteRect);

   // Get the first screen device from the Display Manager.
   // Tell it to return only active screen devices so that we won't have to check here.
   theGDevice = DMGetFirstScreenDevice(true);

   // Loop until we run out of hiliteRgn or GDevices.
   while ((theGDevice) && (!EmptyRgn(hiliteRgn)) {
      // Does this device's rect intersect hiliteRgn?
      RectRgn(&(**theGDevice).gdRect, gdRectRgn);
      SectRgn(hiliteRgn, gdRectRgn, &tempRgn);
      // If it does, highlight it.
      if (!EmptyRgn(tempRgn)) {
         // Highlight the area described by tempRgn.
         ...
         // Take the area we just highlighted out of the region to highlight.
         DiffRgn(hiliteRgn, tempRgn, hiliteRgn);
      }
      theGDevice = DMGetNextScreenDevice(theGDevice, true);
   }
   DisposeRgn(hiliteRgn);
   DisposeRgn(gdRectRgn);
   DisposeRgn(tempRgn);
}
```

should register with the Display Manager so that it will be notified (through a notification callback or an Apple event) if the graphics world has changed.

Don't assume that just because your application uses only one logical display, it's drawing on only one physical device.
Some applications assume that they're using only one piece of graphics hardware when they're actually using multiple physical devices. An example of this is a multimedia player that searches through graphics devices and uses the first device it finds that meets its criteria for bit depth or size. This technique causes a problem when the application uses Toolbox calls specific to one physical graphics device, such as using SetEntries to animate the color table. If mirroring is turned on, this changes the color table of only one device; the second physical device still has the old color table.

Solution: If you use Toolbox calls specific to one physical graphics device, make sure you do it for all devices that overlap your application's windows and not just the first one you find. As shown in Listing 2, you could use DeviceLoop to accomplish this by calling SetEntries in

Listing 2. Calling SetEntries for overlapping devices

```
OSErr MySavvySetEntries(WindowRef aWindow, CTabHandle newColorTable)
{
   RgnHandle            tempWindowStructRgn;
   DeviceLoopDrawingUPP setEntriesDeviceLoopRD;
   OSErr                theErr = noErr;

   tempWindowStructRgn = NewRgn();
   // Was there a problem making the region?
   if ((theErr = QDError()) != noErr)
      return (theErr);
   GetWindowStructureRgn(aWindow, tempWindowStructRgn);

   // We want to get called for every display that intersects our window.
   setEntriesDeviceLoopRD = NewDeviceLoopDrawingProc(SetEntriesDeviceLoop);
   DeviceLoop(tempWindowStructRgn, setEntriesDeviceLoopRD, (long) newColorTable, singleDevices);
   DisposeRoutineDescriptor(setEntriesDeviceLoopRD);
   DisposeRgn(tempWindowStructRgn);
   return (theErr);
}

static pascal void SetEntriesDeviceLoop(short depth, short deviceFlags, GDHandle targetDevice,
      long userData)
{
#pragma unused(depth, deviceFlags)
   CTabHandle  newColorTable = (CTabHandle) userData;
   GDHandle    savedCurrentGDevice;

   // Since we'll be changing the current GDevice, we need to save and restore it.
   savedCurrentGDevice = GetGDevice();

   // SetEntries applies to the current GDevice, so make targetDevice the current GDevice.
   SetGDevice(targetDevice);

   // Insert the entire table into targetDevice. Do it in indexed mode.
   SetEntries(-1, 255, &(**newColorTable).ctTable[0]);

   // Restore the old current GDevice.
   SetGDevice(savedCurrentGDevice);
}
```

RECEIVING AND RESPONDING TO DISPLAY MANAGER EVENTS

The Display Manager has been available since System 7 and is built into system software version 7.1.2 and later. If the Display Manager isn't around, the application can count on devices not moving around and changing sizes.

An application must set the isDisplayManagerAware flag in its 'SIZE' resource to receive Display Manager events in its main event loop. If your application sets this flag, it's responsible for moving its windows if the graphics world changes, making sure that the windows all remain visible. If it doesn't set this flag, the Display Manager will automatically move windows for your application. Even if your application lets the Display Manager handle its windows automatically, it can still register for Display Manager events by using a callback procedure. The callback procedure is passed an Apple event that the application has to parse.

The Graphical Truffles column in *develop* Issue 25 describes this in more detail.

your DeviceLoop drawing procedure. Or you could use the Palette Manager instead of SetEntries.

Don't cache the graphics devices and their state on application startup.
With the Display Manager, many things about the graphics world can change, such as the following:

- Users can change resolutions on multiple-scan displays.
- Users can deactivate displays with the Monitors & Sound control panel (system software version 7.5.2 and later).
- Graphics mirroring could be off when the application is launched and turned on while it's running.
- Users can put their PowerBooks to sleep and attach or remove an external display.

If your application doesn't respond to these changes, it might present an inconsistent interface to the user. For example, the MPW Shell used to cache the gdRect of the main display and pass it to SizeWindow and MoveWindow. So if users changed the resolution of a display, they couldn't grow or move windows beyond the previous size of the display.

Solution: If the Display Manager is present, you should watch for Display Manager notifications to detect changes in the graphics world. (See "Receiving and Responding to Display Manager Events.") Specialized pieces of code, such as extensions or components, can register a callback procedure with the Display Manager. To avoid problems, you should use this method instead of patching to determine when the display environment changes. (Display Manager 2.0 and later even notifies you when the bit depth changes.) Better yet, don't cache device information at all unless your code absolutely needs the extra ounce of performance.

The Display Manager passes all the information about the old and new display configurations to the application when the world changes. The Apple event–handling code on this issue's CD shows some ways to handle Display Manager events. For instance, to detect whether any of the graphics devices have moved, we parse the Apple event and compare the old and new gdRects:

```
AEGetNthDesc(&DisplayID, 1, typeWildCard,
   &tempWord, &OldConfig);
AEGetKeyPtr(&OldConfig, keyDeviceRect,
   typeWildCard, (UInt32 *) &returnType,
   &oldRect, sizeof(Rect), nil);
AEGetNthDesc(&DisplayID, 2, typeWildCard,
   &tempWord, &NewConfig);
AEGetKeyPtr(&NewConfig, keyDeviceRect,
   typeWildCard, (UInt32 *) &returnType,
   &newRect, sizeof(Rect), nil);
if (!EqualRect(&oldRect, &newRect))
   HandleGDevicesMoved();
```

Don't assume that the menu bar will never move.
This is especially a problem with games or multimedia applications that would like to hide the menu bar. Most applications hide the menu bar by adding the menu bar area to the gray region. (The pointer to the gray region is stored in the GrayRgn global variable and can be retrieved with the Window Manager function GetGrayRgn.) When the user moves the menu bar from one display to another (using the Monitors or Monitors & Sound control panel), the Display Manager reconstructs the gray region. If the application wants to show the menu bar again, it removes the old menu bar area from the rebuilt gray region, but this will not accurately reflect the available screen real estate. This top strip, where the menu bar was, would be lost.

As shown in Figure 1, the menu bar was on the display on the left before the application was launched. The

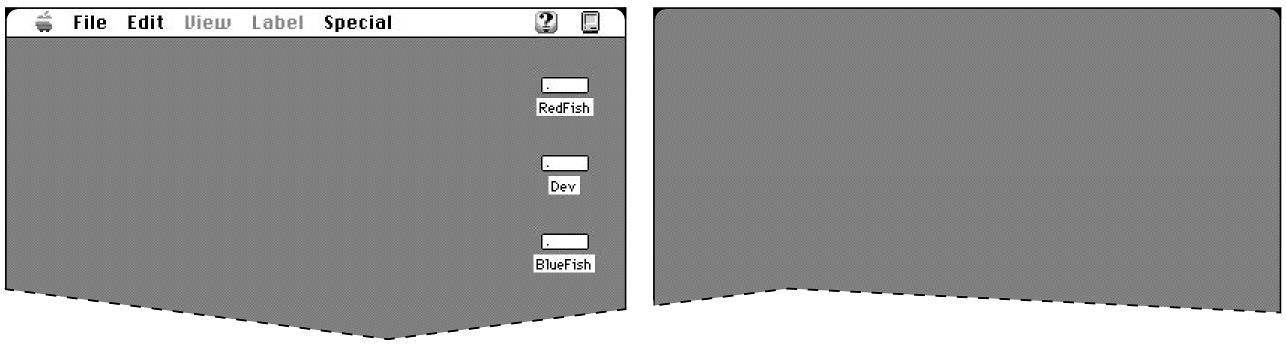


Figure 1. Original location of menu bar

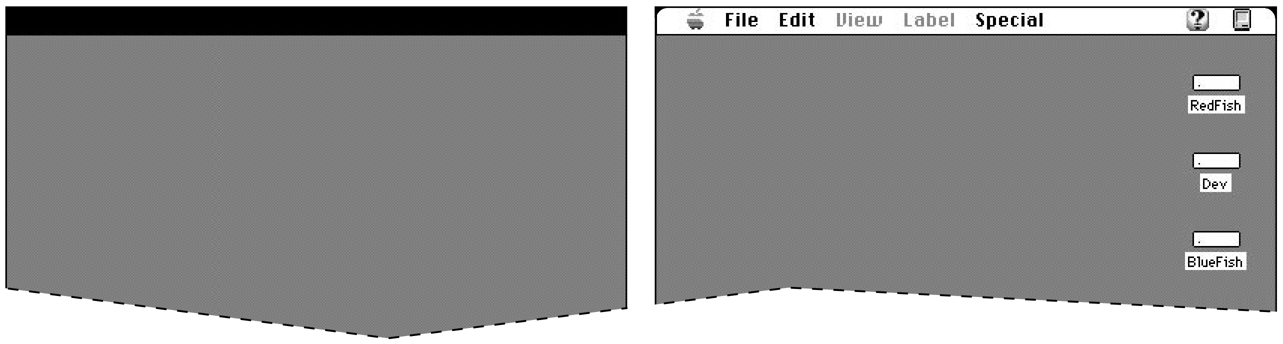


Figure 2. Gray region lost because menu bar moved

application saved the menu bar location and added it to the gray region so that it could draw there. As shown in Figure 2, when the application quit, it subtracted the old menu bar area back out of the gray region. Since the menu bar was moved, this part of the gray region is now lost. No desktop is drawn where the menu bar used to be (the black rectangle).

Solution: Become Display Manager–aware and get notified by the Display Manager when the menu bar moves.

Don't assume that menus will be drawn on only one display.
In the past, it was a safe assumption that the menu bar would be drawn on only one device at a time, so anything that wanted to draw in the menu bar (such as an MDEF) needed to know about only one display and a single bit depth. Since developers typically don't use DeviceLoop to draw menus or draw in the menu bar, sometimes images are drawn incorrectly when there are overlapped displays, especially on displays with different bit depths. An example would be a menu that contains color, like the Label menu in the Finder.

Solution: If you draw directly to the menu bar or to your menus and bypass QuickDraw, use DeviceLoop. Note that drawing in the menu bar with standard QuickDraw calls works fine in the mirrored case because QuickDraw takes care of the overlapping case for you.

Don't draw directly to the screen and bypass QuickDraw without checking for the mirrored case.
Obviously, if an application draws directly to the screen and mirroring is on, the other displays will not reflect any of the drawing.

Solution: Always allow the user to go back to a more "compatible" mode that uses QuickDraw. If the application detects that mirroring is on (by calling DMIsMirroringOn), consider falling back to CopyBits to get your data to the screen.

Don't assume that certain 680x0 registers will contain the same values inside a DeviceLoop drawing procedure as when DeviceLoop was called.
We discovered this bug when we made DeviceLoop PowerPC-native for mirroring performance. Some developers who write their DeviceLoop drawing procedure in 680x0 assembly language rely on the fact that the value of A6 when the drawing procedure was

called is the same as when DeviceLoop was called. Application developers relied on this to enable them to share stack frames between the caller of DeviceLoop and the DeviceLoop drawing procedure. This will not work if a PowerPC-only version of DeviceLoop is present, so it will not work under Mac OS 8. If you write your DeviceLoop drawing procedure in a high-level language like C, however, you don't have to worry about this problem.

Solution: If you want to share data with your DeviceLoop drawing procedure, use the userData refCon supplied with DeviceLoop. If you need to rely on A6 remaining constant, one solution would be to pass in the A6 of the DeviceLoop caller in the userData parameter and set that to the A6 of the DeviceLoop drawing procedure. Be sure to save and restore the original A6 of the DeviceLoop drawing procedure.

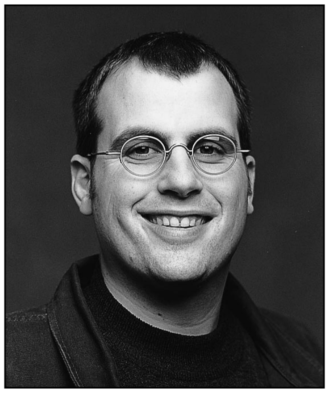
SUCCEEDING IN A CHANGING GRAPHICS WORLD

The Display Manager offers many new features that enable users to configure their graphics devices dynamically. However, this dynamic display environment invalidates certain assumptions that developers might have made when programming in a static graphics environment. This column should start you thinking about these issues. Although the Display Manager does attempt to preserve compatibility with existing applications by moving windows around and preserving graphics device information, it can't fix everything. Your application needs to be able to function in a changing graphics world.

Thanks to Tom Dowdy, Ian Hendry, Mike Marinkovich, and Mike Puckett for reviewing this column.•

Connecting Users With QuickTime Conferencing

QuickTime Conferencing (QTC) is a new Apple technology that helps developers add real-time sharing of sound, video, and data to their applications. This overview suggests the different ways you can use QTC to help users collaborate. The article describes the components that most developers will need to use to take advantage of QTC and discusses Watcher and Caster, two QTC applications that enable users to tune into network broadcasts and create broadcasts for others to view.



DEAN BLACKKETTER

Video telephones abound in science fiction movies. From Buck Rogers to Star Trek, visions of the future show people communicating visually over long distances. This futuristic technology is available to Macintosh developers and users *now*. QuickTime Conferencing provides a platform for developers to easily enable users to share sound, video, and data across a variety of networks.

QTC ships with selected Power Macintosh computers and with some hardware bundles, and can be licensed by developers to ship with their applications. Apple provides a basic videoconferencing application, Apple Media Conference (AMC), and developers are encouraged to create QTC applications that interoperate with AMC and add cool new collaborative features.

This article will give you background information on the QTC architecture, tell you about the components that make up that architecture, and then describe in detail the workings of two simple QTC applications, Watcher and Caster, that enable the user to watch audio and video and to broadcast them onto a network. This issue's CD contains the source code for these applications as well as the QTC documentation and the extension and header files.

QUICKTIME CONFERENCING — THE BIG PICTURE

QuickTime Conferencing provides a platform for building Macintosh applications that can send and receive audio, video, and data between computers connected on a network. QTC supports basic two-way audio communication and a video "telephone" type of connection, and it supports a wide variety of other models as well. One of the goals of QTC is to provide developers with a set of tools that make it easy to add real-time media sharing across a number of different kinds of networks.

This opens up the possibility of adding sound and video to multiuser applications where it would have been prohibitively difficult before — and these don't have to be

DEAN BLACKKETTER (dean@artemis.com) used to work for Apple in the Advanced Technology Group. He now has a gig with Artemis Research working on "the next big thing." He plays in San Francisco with his wife, their cat, and the scary elf who lives on top of the fridge.•

conventional telephony-style applications. Imagine a flight simulator that allows you to talk with your fellow squadron members, or a groupware document-markup application that lets your fellow editors see your expression upon examining the latest changes. Picture a regional educational system that enables dozens of students to tune into an 8 A.M. lecture from their dorm rooms across campus or across the state. This isn't the stuff of science fiction anymore.

QTC uses many of the services provided by QuickTime itself and shares an architectural basis in the Component Manager. QTC takes advantage of the Image Compression Manager for video compression and decompression, the sequence grabber components for capturing media, and the Movie Toolbox for recording movies to disk. When new features and improvements are added to QuickTime, they often can be used by QTC immediately. For example, components created for video or sound compression in QuickTime are automatically available to QTC.

CONFERENCE CONFIGURATIONS

QTC's basic metaphor for real-time media connections is that of a conference. Conferences are quite flexible and can be configured in a variety of ways. They can have one, a few, or many members, connected symmetrically or asymmetrically. As illustrated in Figure 1, connections can take one of three forms: *point to point*, for two-way conferences; *multipoint*, for virtual meetings and groupware applications; or *broadcast*, for transmitting from one member to many others.

Members can send or receive sound, video, or data. Media types can be added, removed, or changed during a conference. Members can join or leave a conference at any time. Conferences can be merged, and data can be sent to one or all of the conference members.

Depending on the application, you may want to give users a single configuration — say, a two-way audio and video connection — or allow them to modify the conference configuration themselves. QTC was designed to support a wide variety of conference configurations and to leave it up to developers to decide which features they need. Indeed, some applications may need to switch between different configurations within a single conference. The applications described later in this article each operate in a single configuration; one can broadcast video and sound to an unlimited number of recipients and the other can tune into one or more broadcast conferences.

NETWORK, PROTOCOL, AND MEDIA INDEPENDENCE

QTC is network, protocol, and media independent. This means that applications don't have to know the specifics of a particular network to set up a QTC conference. QTC 1.0.2 ships with support for TCP/IP and AppleTalk networks; third parties and Apple are working on adding new networks like ISDN, isoEthernet, and ATM to the list. QTC 1.0.2 supports a new media-oriented network protocol, called MovieTalk, but can also support other media protocols such as the ITU H.320 standard and the emerging standards used on the Internet Multicast Backbone (MBONE).

The media that flows between conference members is organized into one or more streams of a particular media type. QTC 1.0.2 supports sound and video streams, which can be compressed with any sound or video compressor. Future versions of QTC will be able to support other media types, such as music and text, to parallel the different track types that can be stored in a QuickTime movie.

THE CONFERENCING EXPERIENCE

QTC provides some of the basic user interface elements called for in a conferencing application. For example, each member of a conference can be represented on the

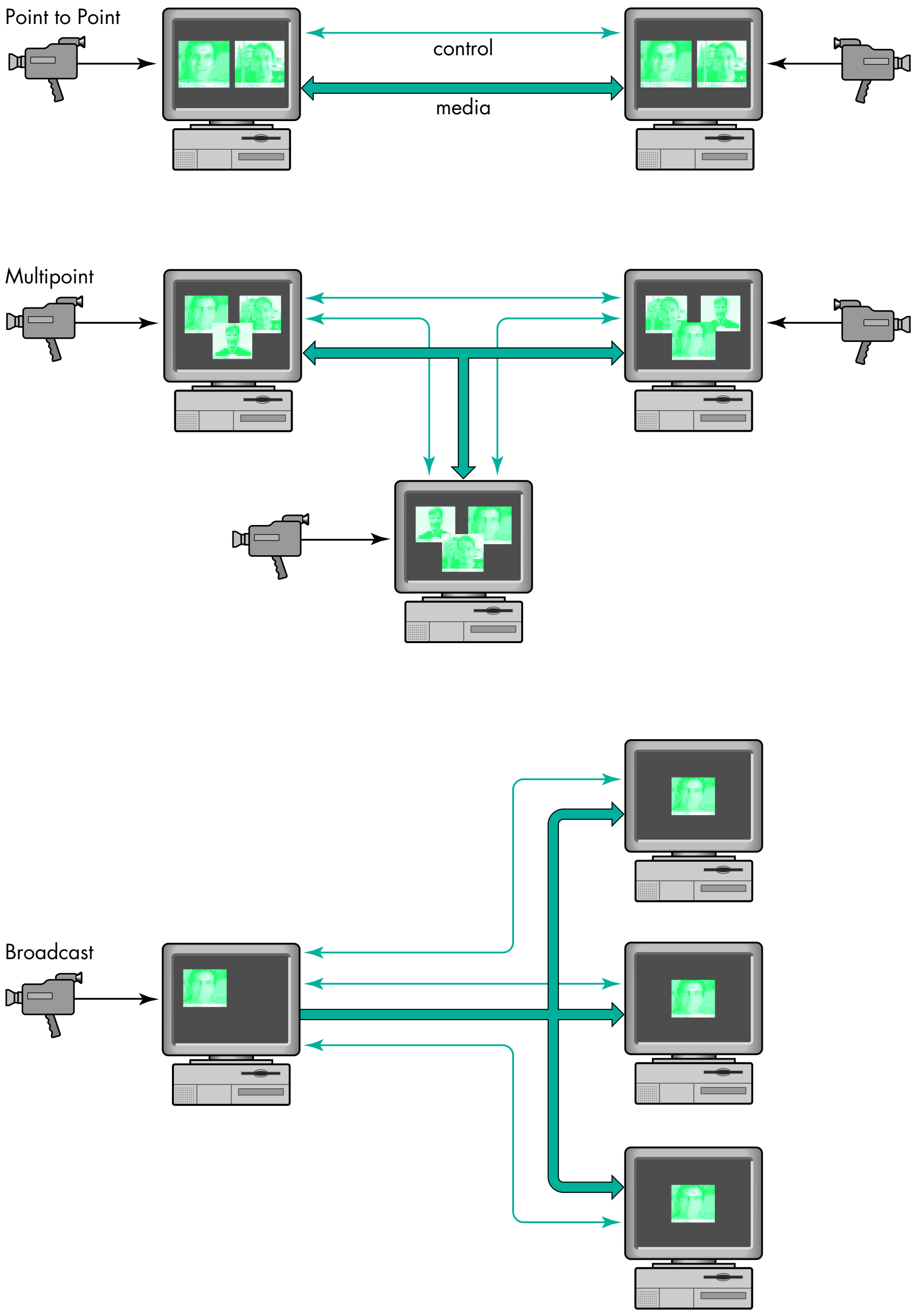


Figure 1. The three types of conference connections

screen with a stream controller, in much the same way that a QuickTime movie controller provides a control representation for a QuickTime movie. In fact, the stream controller and the movie controller share a similar user interface, so that a user who has some experience with one can apply that knowledge to the other.

QTC also provides a standard user interface enabling users to choose who to call and include in a QTC conference, in the form of browser components. Browsers work a bit like the Standard File Package that allows users to open and save files: they provide a standard interface for choosing fellow users or searching through PowerTalk catalogs to find other conference members and place calls to them.

QUICKTIME CONFERENCING COMPONENTS

QTC, like much of QuickTime, is built of Component Manager components. Apple provides a basic suite of components that enable the user to share data and send and receive compressed video and audio on a few different networks. Before we dive into our example applications, let's go over some of the component types that make up the QTC component suite.

There are three main types of QTC components that most developers will need to know about to add QTC support to their applications: the *conference component*, the *stream controller component*, and the *browser component*. I'll describe these in some detail. Developers who want to do fancier things will probably need to know about some of the other components; the key ones are briefly described later.

Because of the modular architecture of QTC, developers can add, extend, or replace features and components. For example, a developer who wants to add support for a new network multimedia protocol can create a new transport component and register it with the Component Manager. Applications can then find that component and specify its use in a conference. Developers who want to improve on the QTC stream controller can capture the standard controller, delegate many of the functions, and replace the ones of interest.

THE CONFERENCE COMPONENT

The conference component is the key player in a QTC conference. It acts as a central hub and does the bulk of the work required to orchestrate the comings and goings of the conference. It's responsible for listening in on the various networks, placing and answering calls, managing and merging multiple conferences, and more. The conference component can also provide some higher-level functionality, such as setting up media capture, handling user events, and even creating and managing conference windows.

Applications create a conference component instance and let the conference component do much of the work needed to create, manage, and end conferences. Applications can then tell the conference component to listen on the networks for incoming calls or to place a call to another member.

Conference components create *conference events* when they need to express some change in a conference to the application. For example, when an incoming call is made to a conference, the conference component will generate an event of type mtIncomingCallEvent to notify the application of the call. Applications call the component routine MTConferenceGetNextEvent periodically to get the events from the conference component, much as applications call the system routine WaitNextEvent to get user and system events from the Event Manager.

In response to these conference events, applications work with the conference component to respond appropriately — for example, creating a window to display a new conference member or send messages to other conference members. Details of working with the conference component will be discussed later when we look at our sample applications, Watcher and Caster.

THE STREAM CONTROLLER COMPONENT

Stream controllers are responsible for handling the default user interface for controlling QTC media streams as well as managing their display on the screen and through the speaker. The conference component is responsible for creating and managing stream controller components. Applications are passed references to the stream controllers by the conference component so that they can keep track of where and how the media is being displayed.

The standard stream controller looks quite a bit like the standard QuickTime movie controller, with buttons to control the flow of media, resize the visual portion of the stream, and adjust the sound levels. The stream controller adds some utility buttons that the movie controller doesn't have: a snapshot button for capturing the current image displayed in the controller and a record button that provides a standard way for a user to record the media in a stream controller. (The conference component or the application is responsible for actually handling the snapshots or recorded movies after the controller has initiated them.)

Controllers associated with the sending side of a media stream (known as *source controllers*) have a slightly different appearance and behavior from those associated with the receiving side (known as *sink controllers*), as shown in Figure 2. The source controller may have a microphone "gain" button that's animated to indicate the level of the audio being sent across the connection. Users who click this button can adjust the volume of the sound being sent across the connection. On the receiving end, the sink controller may display a volume control button that behaves like the speaker button on the standard movie controller, allowing the user to adjust the volume of the incoming stream.

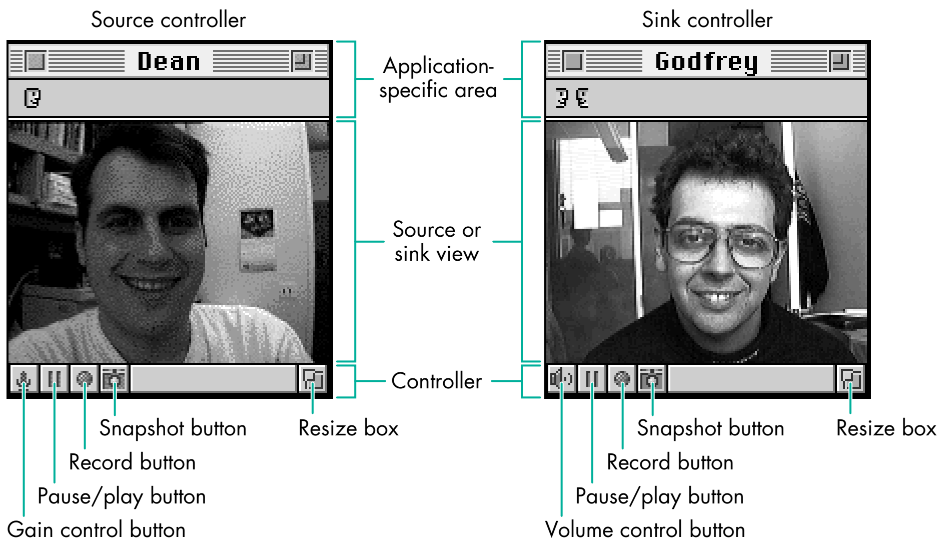
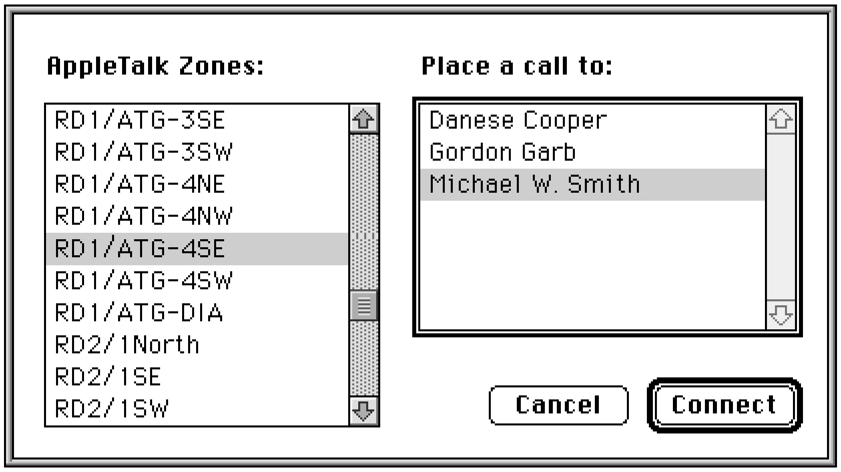


Figure 2. Source and sink controller user interfaces

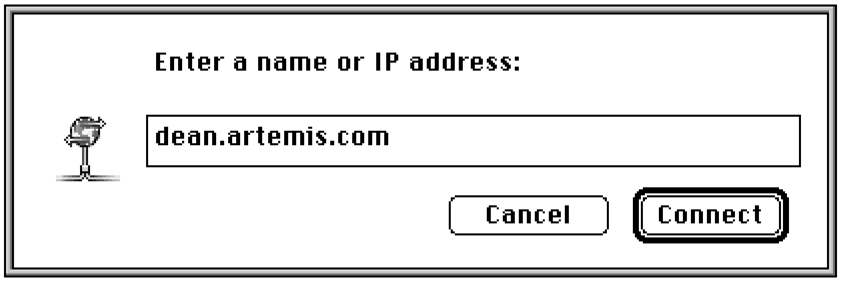
THE BROWSER COMPONENT

To place a call or add another member to a conference, the user needs to specify the other member to call. Browser components provide a simple way for users to browse the network and identify other members. Browser components come in two flavors: network-specific browsers and the PowerTalk browser. The PowerTalk browser and browsers specific to TCP/IP and AppleTalk are shown in Figure 3.

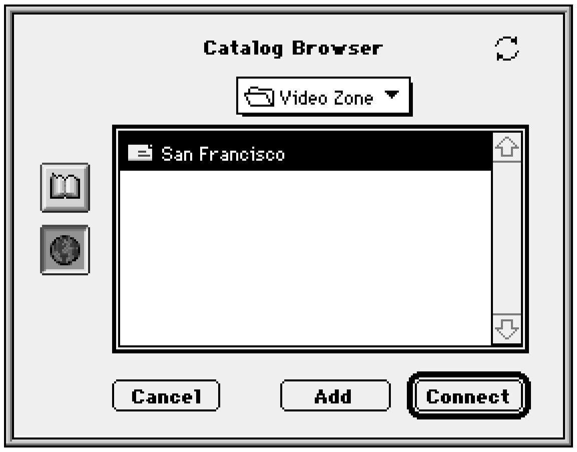
For each different network type — such as TCP/IP or AppleTalk — unique browser components are provided that allow the user to specify a network-specific address. For example, as shown in Figure 3, the AppleTalk browser presents the user with a Chooser-style interface whereby the user can choose the zone and then the registered name within that zone on an AppleTalk network, similar to using the Chooser to pick



AppleTalk



TCP/IP



PowerTalk

Figure 3. Browsers

a LaserWriter on an AppleTalk network. The TCP/IP browser provides a simple type-in interface that can accept TCP/IP addresses in numerical or text form.

The PowerTalk browser, on the other hand, is considered a generic or universal browser, not tied to a particular network or addressing scheme. Users who have PowerTalk installed can take advantage of the various PowerTalk catalogs and business cards; these provide an integrated way for users to organize and find other QTC users in the same way that they access electronic mail addresses via PowerTalk. The PowerTalk browser allows the user to choose a business card from a PowerTalk catalog that contains a QTC entry (provided by the QTC PowerTalk Template).

This works for local user catalogs and catalogs provided by PowerShare servers, as well as the generic AppleTalk network catalog, which allows the user to look out onto the network and into AppleTalk zones for other users. Users can edit their personal catalogs from within the Finder, consistent with the standard PowerTalk human interface.

OTHER QUICKTIME CONFERENCING COMPONENTS

QTC defines and uses many other kinds of components besides the three just mentioned. Several of these component types may be of interest to developers who want to add support for new networks or new media protocols; others may be of use to developers who want to have more control over their conferences. Some of these are listed here.

- *Stream director components* are responsible for managing the media streams that flow between conference members. Stream directors are of two types: source stream directors and sink stream directors. Source stream directors work with media sources, such as QuickTime sequence grabbers, to capture audio and video data to be sent across the network. Sink stream directors are responsible for setting up and displaying incoming media data: video to the screen and sound to the speaker. Conference components and controller components handle most of the management and control of stream directors.
- *Transport components* are responsible for implementing the network protocol that communicates media data, formats, and control information. MovieTalk, the default QTC protocol, is implemented as a transport component. Apple's H.320/ISDN conferencing card adds another transport type that supports the ITU H.320 video conferencing standard. Developers who want to support new media protocols can create new transport components to translate the control messages from a conference into messages appropriate for the new protocol and vice versa.
- *Network components* contain code specific to a given network type. QTC 1.0.2 provides network components for AppleTalk and TCP/IP. Future versions of QTC will provide direct OpenTransport network interfaces as well as others. Network components can provide access to multicast services on some shared networks so that media data can be sent to multiple recipients without having to send out multiple copies of that data. (See "About AppleTalk Multicast" for a discussion of one such multicast service.) The conference component automatically takes advantage of multicast network services when they're available.
- *Recorder components* attach to stream directors and provide a mechanism to record to disk the media sent or received within a conference. Apple provides a recorder component that records media into QuickTime movies and can attach to multiple members via stream directors to create movies of entire conferences at once.

Several other components are used within QTC, including player components, flow control components, and others of interest to developers who want to extend QTC to support new networks, protocols, and media. Figure 5 shows how a number of QTC components typically work together within the all-encompassing conference component. For information on all of the components that make up QTC, check out the QTC documentation on this issue's CD.

TUNING IN WITH WATCHER

Probably the best way to show how to use QTC in an application is with some examples, so we've created Watcher and Caster. Watcher lets the user tune into

ABOUT APPLETALK MULTICAST

Digital video and sound can generate a great deal of data, even when compressed. Hard disk space is getting to be quite cheap, but network bandwidth is still an expensive and shared commodity. To keep your fellow users and network administrators happy, we developed multicast extensions to AppleTalk that allow a single copy of QuickTime Conferencing media sent out onto a network to be received and displayed by any number of users.

AppleTalk Multicast consists of a special packet format and a routing protocol that makes efficient use of the network bandwidth. On a single network segment, AppleTalk Multicast uses multicast packets that can be received by anyone on that local network. On an AppleTalk internet, multicast-aware routers communicate with each other with a new protocol called SMRP, the Simple Multicast Routing Protocol, as shown in Figure 4. The routers deliver copies of the media data only to other networks in which there's a user who wants to receive that data. Networks with no users interested in the broadcast aren't burdened with the network usage.

Apple has licensed AppleTalk Multicast and the SMRP protocol to Cisco Systems, Inc. Cisco's router software as of version 11.0 supports this multimedia protocol.

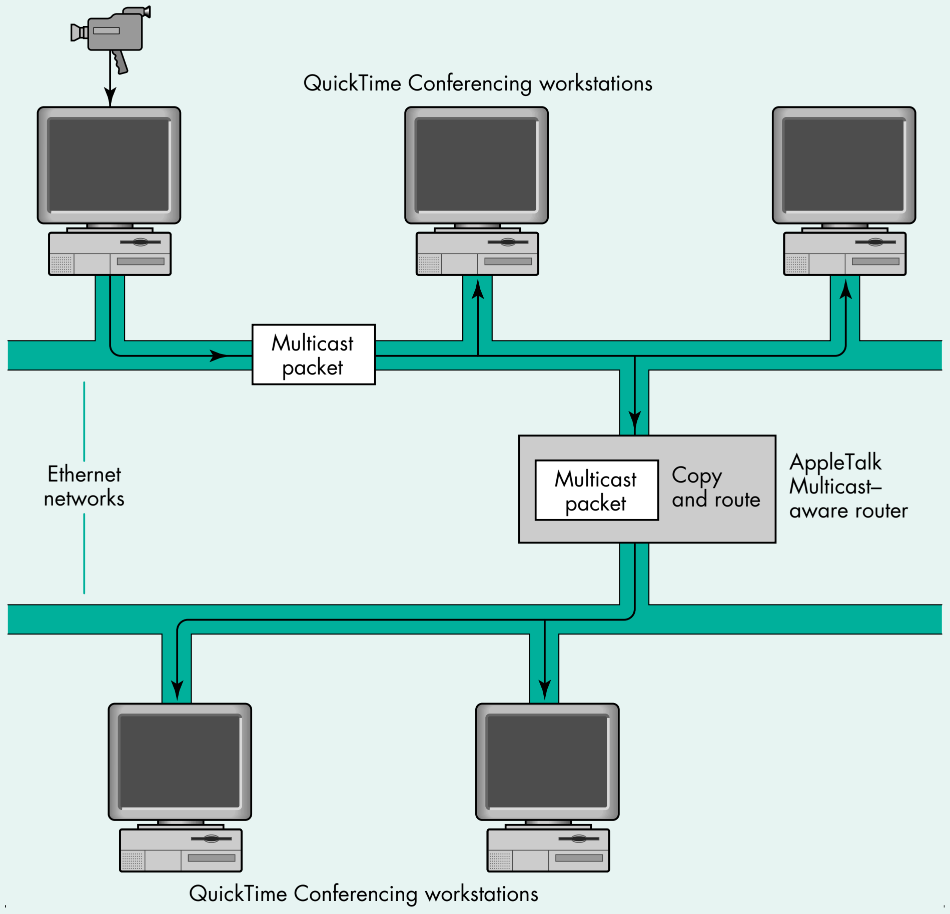

QuickTime Conferencing workstations
Multicast packet
Ethernet networks
Multicast packet
Copy and route
AppleTalk Multicast–aware router
QuickTime Conferencing workstations

Figure 4. AppleTalk Multicast routing

broadcasts on AppleTalk networks, while Caster enables the user to create broadcasts that can be watched by others on the AppleTalk network. Watcher and Caster are compatible with Apple Media Conference (AMC), the QTC application that Apple ships with selected CPUs and product packages, so you can use Watcher to watch a broadcast that's being sent by AMC or Caster, and you can use Caster to create broadcasts that can be received by Watcher and AMC.

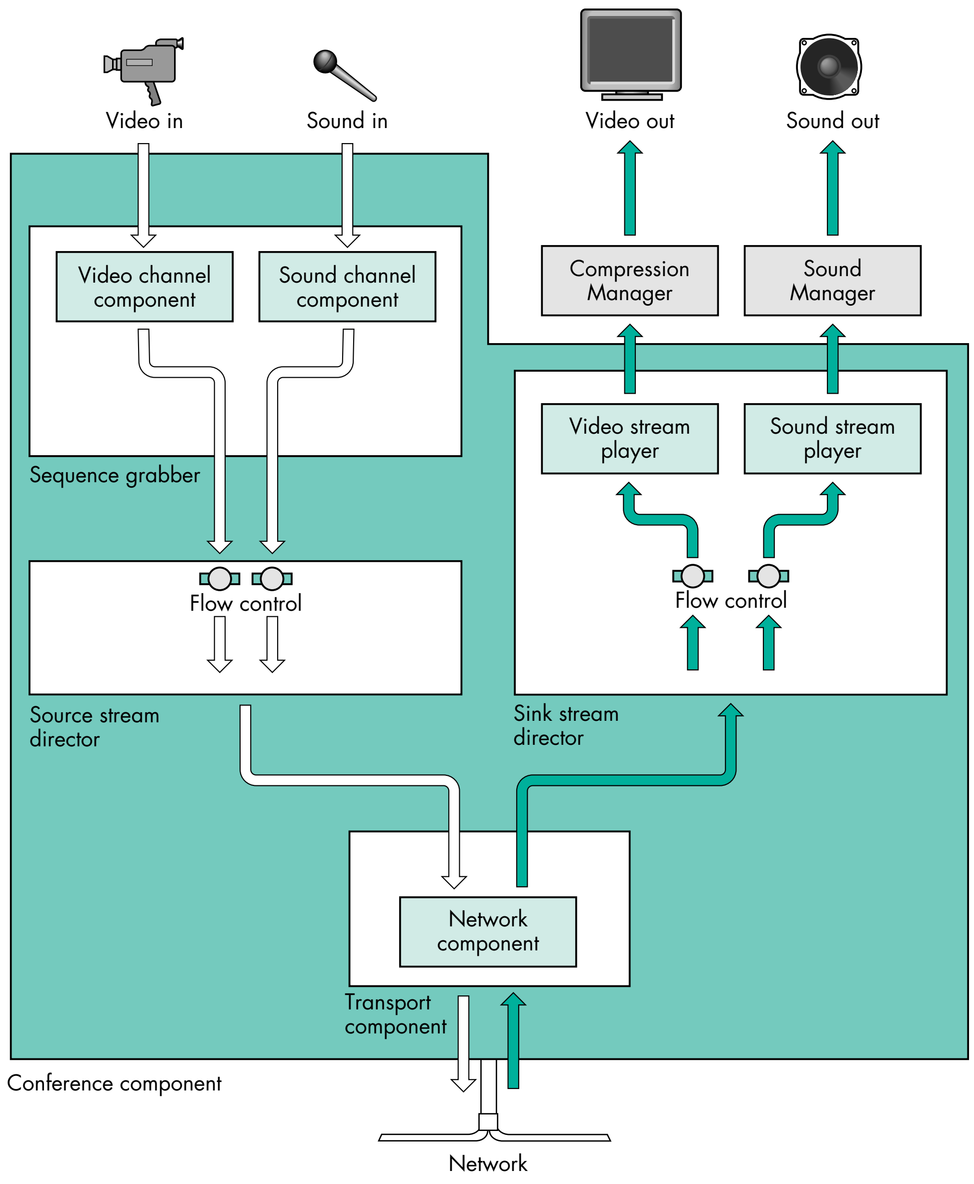


Figure 5. How QTC components work together within the conference component

Note that in several places in Watcher and Caster, we do some work manually that otherwise could be done automatically by the conference component. We do this extra work to demonstrate how you can customize an application if the behavior that you want is different from the default behavior offered by the conference component.

HOW WATCHER WORKS

Watcher is a relatively simple Macintosh application. After setting up the application environment, Watcher sets up the conference component that will place calls and manage the incoming media. Then, within the event loop, the application checks for user and system events and also checks the conference component for conference events, which indicate changes in the conference state and may require responses from the application.

The overall flow of Watcher or any QTC application that uses the conference component is as follows:

```
QTCApp()
{
   SetupApplication();
   SetupConferenceComponent();
   StartListening();
   do {
      ProcessUserEvents();
      ProcessConferenceEvents();
   } while (!gQuit);
   CleanUpConferenceComponent();
   CleanUpApplication();
   ExitToShell();
}
```

Below, I'll go into more detail about the three major application responsibilities — setting up the conference component, handling conference events, and cleaning up at the end of the conference — showing the core routines that deal directly with the conference component. Check out the full source code to see them in the context of the entire application.

SETTING UP A CONFERENCE

Listing 1 shows how the conference component is created and initially configured. The Component Manager call OpenDefaultComponent is used to create and open

Listing 1. CreateWatchConference

```
ComponentResult CreateWatchConference(MTCString63 userName)
{
   ComponentResult   err;

   /* Create a conference record. */
   err = NewConference(&gConference);
   if (err == noErr) {
      gConference->confComponent = OpenDefaultComponent(kMTConferenceType, kMTMovieTalkSubType);
      if (gConference->confComponent) {
         /* Tell the conference component that we only want to receive media, not send. */
         err = MTConferenceSetMode(gConference->confComponent, mtReceiveMediaModeMask);
         /* Tell the conference component to prepare to use AppleTalk.
            The funky C string tells the conference component:
               mtlk = use the MovieTalk transport component
               atlk = use the AppleTalk network component
               NoIncomingCalls = the AppleTalk-specific NBP type that's used for listening;
                i.e., there will be no incoming calls
         */
         if (err == noErr)
            err = MTConferenceListen(gConference->confComponent, userName /* User name */,
                                    userName /* Service name */,
                                    (MTCString)"mtlkatlk\tNoIncomingCalls\x0D");
      }
      else
         err = couldntGetRequiredComponent;
   }
   return err;
}
```

an instance of the conference component; then the conference component mode is set to indicate that the conference will be used to receive media. Finally, the component is told what networks to prepare for connections on — AppleTalk in this case — and how to identify itself on that network.

MTConferenceListen (as well as MTBrowserBrowse, a call we'll encounter a little later) uses a C string of type MTCString to describe the network and transport configurations. In Listing 1, the string "mtlkatlk\tNoIncomingCalls\x0D" indicates that the conference component should listen for calls that have a transport subtype of 'mtlk' (the component subtype for the MovieTalk transport component) and a network subtype of 'atlk' (the subtype for AppleTalk networks). The "\t" delimits the subtypes from the network-specific configuration data that follows. For AppleTalk networks, this is the Name Binding Protocol (NBP) type "No Incoming Calls." Finally, the configuration is terminated with a carriage return ("\x0D"). You can string together multiple configuration strings (each terminated with a carriage return) to listen in on multiple networks for calls. Check out the full documentation for a more complete explanation of the configuration strings.

BROWSING THE NETWORK

Now that the conference is set up, we can place a "call" out onto the network to the broadcaster that the user wants to watch. We'll use the AppleTalk browser component to pick a registered broadcaster.

The BrowseName routine (Listing 2) opens the browser component and uses the MTBrowserBrowse component call to specify which kind of network entity to look for. In this case it's a MovieTalk entity registered on an AppleTalk network with the NBP type of "Multicaster"; this type identifies broadcasts from Caster and AMC. MTBrowserBrowse then presents users with the browser dialog, where they can "surf" the network and find the appropriate broadcaster. Some browsers (like the

Listing 2. BrowseName

```
ComponentResult BrowseName(MTNamePtr name)
{
   MTNameListPtr       allNames = 0;
   ComponentResult     err;
   MTBrowserComponent  browser = nil;

   browser = OpenDefaultComponent(kMTBrowserType, kMTAppleTalkSubType);
   if (browser) {
      err = MTBrowserBrowse(browser, 0, nil,
               (MTCString)"mtlkatlk\tMulticaster\x0D", 0, &allNames);
      CloseComponent(browser);
   }
   else
      err = couldntGetRequiredComponent;
   if ((allNames != 0) && (err == noErr)) {
      /* Copy the first name record; that's all we're interested in. */
      *name = allNames->list[0];
      /* Dispose of the list of names. */
      DisposePtr((Ptr)allNames);
   }
   return err;
}
```

PowerTalk browser) can return multiple names in an MTNameList. We're only interested in the one AppleTalk broadcast picked by the user, so we pick off the first MTName from the MTNameList.

TUNING IN

CallMember (Listing 3) is the code needed to tell the conference component to place a call to the broadcaster. The calling routine passes in the MTName (obtained from BrowseName) and a pointer to the window in which the broadcast is to appear (and that window's size). The resize parameter will be used later to determine whether to resize the window automatically to the dimensions of the video being broadcast. CallMember returns a pointer to a new MemberRecord data structure, where the information about each broadcast-watching window is kept. The important conference component call here is MTConferenceCall, which is passed a reference to the conference component, an arbitrary name for the conference, and the MTName describing the party whose broadcast we want to watch.

Listing 3. CallMember

```
ComponentResult CallMember(MTName* name, WindowPtr wind, Rect* box,
                           Boolean resize, MemberRecord** member)
{
   MemberRecord*     mr;
   ComponentResult   err;

   /* Create a new member record. */
   err = NewMember(&mr);
   if (err == noErr) {
      mr->member = MTConferenceCall(gConference->confComponent,
                           (MTCString)"Watcher", name);
      mr->box = *box;
      mr->window = wind;
      mr->resize = resize;
      if (member)
         *member = mr;
   }
   return err;
}
```

Note that the conference component manages each independent connection to a broadcaster as a unique conference. That's just fine for our application, since the broadcast windows are really independent. In multiparty connections, however, conferences can be joined and then individual members can belong to the same conference. In that case the conference name parameter in MTConferenceCall ("Watcher" in Listing 3) may have more meaning and may be used to distinguish independent conferences. In our case, we give them all the same name.

TURNING ON

Now that the conference call has been placed, we need to check the conference component periodically to find out about changes in the conference. Listing 4 shows the routine CheckConferenceEvents, which is intended to be called within the main event loop of the application. Each time through the loop, we call MTConferenceGetNextEvent. Most of the time this will return false, indicating that there are no new events. When some state in the conference has changed, it will return true, and we should then parse the event (with HandleConferenceEvent) to see what the correct response is.

Listing 4. CheckConferenceEvents

```
ComponentResult CheckConferenceEvents(void)
{
   MTConferenceEvent confEvent;
   ComponentResult   err;

   if (MTConferenceGetNextEvent(gConference->confComponent, &confEvent))
      err = HandleConferenceEvent(&confEvent);
   return err;
}
```

The MTConferenceEvent data structure, also known as an event record, has several fields that we'll use in the following listings. The **what** field indicates the type of event; depending on this, HandleConferenceEvent (Listing 5) switches to the individual subroutines corresponding to each event. The **surprise** field, if not set to 0, contains a handle to data that's associated with the event and needs to be disposed of after use. The other fields, **who**, **err**, and **bonus**, contain references to the members, error codes, and event-specific data, respectively. See the documentation for details on the meanings of these fields for all event types.

Listing 5. HandleConferenceEvent

```
ComponentResult HandleConferenceEvent(MTConferenceEventPtr confEvent)
{
   ComponentResult   err = noErr;

   /* Like a user event handler, we switch on the different conference
      events. */
   switch (confEvent->what) {
      case mtConferenceReadyEvent:
         err = DoConfReady(confEvent);
         break;
      case mtMemberReadyEvent:
         err = DoMemberReady(confEvent);
         break;
      case mtMemberTerminatedEvent:
         err = DoMemberTerminated(confEvent);
         break;
      case mtMemberJoiningEvent:
         err = DoMemberJoining(confEvent);
         break;
      case mtPhoneRingingEvent:
         err = DoPhoneRinging(confEvent);
         break;
```

(continued on next page)

Listing 5. HandleConferenceEvent *(continued)*

```
      case mtRefusedEvent:
      case mtFailedEvent:
         err = confEvent->err;
         break;
      default:    /* Ignore all others. */
         break;
      }

   /* If there's data associated with this event, free it. */
   if (confEvent->surprise)
      DisposeHandle(confEvent->surprise);
   return err;
}
```

After a call has been placed and a connection has been established with the remote side, an event of type mtMemberJoiningEvent is returned by the conference component. Upon receiving this event our application calls DoMemberJoining (Listing 6) and simply makes a record of this new member and adds it to our list of members. The conference component will continue to establish the connection and will notify us further when the connection has been completely brought up.

Listing 6. DoMemberJoining

```
struct MemberRecord {
   MTControllerComponent   controller;
   MTDirectorComponent     director;
   MTConferenceMember      member;
   WindowPtr               window;
   Boolean                 resize;
   Rect                    box;
   MemberRecord*           next;
};
...
ComponentResult DoMemberJoining(MTConferenceEventPtr confEvent)
{
   MemberRecord*     currMember;
   ComponentResult   err = noErr;

   err = NewMember(&currMember);
   if (err != noErr) {
      currMember->member = confEvent->who;
      AddMember(gConference, currMember);
   }
   return err;
}
```

Once the connection has been fully established, the conference component sends us an event of type mtMemberReadyEvent. Now we have a little more work to do. In this case, the application needs to create a controller and place that controller into a window for incoming media to be displayed. The conference component can do much of this work for you, including creating a controller (and its associated stream director) as well as creating a window and even handling user events for that window, with the MTConferenceNewPreparedController call. For many applications this method is perfectly adequate, but if you need more control over event handling and window management in your application, you'll want to do this work manually, as we do in Watcher and Caster. Use of MTConferenceNewPreparedController is demonstrated in the SeeWorld sample applications included on this issue's CD; check out the Rogues and Guardian examples in particular.

DoMemberReady (Listing 7) first checks to see if we can expect media to be sent by the new member. (If the member isn't sending media, there's no point in setting up a window.) If the member is sending media, we create a controller component and a stream director component, which are responsible for displaying the media data. After this, we call MTControllerNewAttachedController to connect the controller

Listing 7. DoMemberReady

```
ComponentResult DoMemberReady(MTConferenceEventPtr confEvent)
{
   ComponentResult  err = noErr;
   MemberRecord*    currMember;
   Point            where = {0, 0};
   Boolean          aTrue = true;

   if (confEvent->bonus & mtReceiveMediaModeMask) {
      currMember = FindMember(gConference, confEvent->who);
      if (currMember == nil)
         return noErr;
      currMember->controller =
         OpenDefaultComponent(kMTControllerType, kMTMovieTalkSubType);
      if (currMember->controller == 0)
         err = couldntGetRequiredComponent;
      if (err == noErr) {
         currMember->director = OpenDefaultComponent(
                           kMTSinkStreamDirectorType, kMTPlayerType);
         if (currMember->director == 0)
            err = couldntGetRequiredComponent;
      }
      if (err == noErr)
         err = MTControllerNewAttachedController(currMember->controller,
                  currMember->director, currMember->window, where);
      if (err == noErr)
         err = MTControllerSetActionFilter(currMember->controller,
                  actionFilterUPP, (long)currMember);
      if (err == noErr)
         err = MTConferenceActivateMember(gConference->confComponent,
                  confEvent->who, currMember->controller);
      if (err == noErr)
         err = MTControllerDoAction(currMember->controller,
                  mtControllerActionPlay, &aTrue);
   }
   return err;
}
```

to the stream director and point it at a window for display. We then do one more thing to the controller before activating it in the conference: we set an action filter for it. The action filter is a callback routine that the controller calls whenever any important action happens within the controller. In our application, the only action that we care about is the resizing of the media data so that we can resize the window. The action filter routine is shown in Listing 8.

Listing 8. MyControllerActionFilter

```
pascal Boolean MyControllerActionFilter(MTControllerComponent mtc,
                                      MTControllerActionType action,
                                      void* params, long refCon)
{
   void*       unused1 = params;
   long        unused2 = refCon;
   RgnHandle   controllerRgn;
   Boolean     handled = false;
   Rect        box;
   WindowPtr   controllerWindow =
                        (WindowPtr)MTControllerGetControllerPort(mtc);

   switch (action) {
      case mtControllerActionControllerSizeChanged:
         /* Find out how big the controller is. */
         controllerRgn = MTControllerGetWindowRgn(mtc,
                                                   controllerWindow);
         /* Resize the window accordingly. */
         if (controllerRgn != nil) {
            box = (**controllerRgn).rgnBBox;
            DisposeRgn(controllerRgn);
            SizeWindow(controllerWindow, box.right, box.bottom, true);
         }
         break;
      default:
         break;
      }
   return handled;
}
```

Finally, DoMemberReady calls MTConferenceActivateMember to activate the member, and we pass MTConferenceActivateMember the newly created controller. Before exiting, we call MTControllerDoAction to tell the controller component to begin playing the incoming media as soon as it begins. (Controllers are by default in a paused state when they're created.)

DROPPING OUT

When the user has decided to close down the reception of the broadcast (say, by closing a broadcast window), the application calls CloseWatch (Listing 9). CloseWatch will find the member record corresponding to the conference member and obtain the conference token associated with that member. (Remember, each member is part of a unique conference, so the member has both a conference token and a unique ConferenceMember identifier.) Then we begin to terminate the conference by calling MTConferenceTerminate.

Listing 9. CloseWatch

```
ComponentResult CloseWatch(WindowPtr window)
{
   ComponentResult     err = noErr;
   MTConferenceToken   theConference;
   MemberRecord*       theMember;

   theMember = FindMemberWindow(gConference, window);
   if (theMember == nil)
      err = paramErr;
   if (err == noErr) {
      theConference = MTConferenceGetMemberConference(
                     gConference->confComponent, theMember->member);
      err = MTConferenceTerminate(gConference->confComponent,
                                 theConference);
   }
   return err;
}
```

The conference isn't completely terminated until we receive an event of type mtMemberTerminatedEvent, which is handled by DoMemberTerminated (Listing 10). DoMemberTerminated is called when the conference connection for this member has been completely terminated, either by an MTConferenceTerminate call or by the remote side closing down. In response, we'll close down the controller and stream director components and the associated window, then free up our application's MemberRecord for this member.

Listing 10. DoMemberTerminated

```
ComponentResult DoMemberTerminated(MTConferenceEventPtr confEvent)
{
   MemberRecord*     member;
   ComponentResult   err;

   member = FindMember(gConference, confEvent->who);
   if (member == nil)
      return noErr;
   RemoveMember(gConference, member);
   if (member->controller)
      CloseComponent(member->controller);
   if (member->director)
      CloseComponent(member->director);
   if (member->window)
      CloseWindow(member->window);
   err = DisposeMemberRecord(member);
   return err;
}
```

That's it for the key QTC routines in Watcher. Check out the source code on the CD to see the entire package come together.

BROADCASTING WITH CASTER

Caster, the broadcasting side of this networked multimedia system, is similar to Watcher in many ways. It uses a conference component (see Figure 6) and processes conference events, but it handles the other side of the conference establishment: setting up and transmitting media and accepting incoming calls. In some ways, Caster is simpler: since it broadcasts to anybody who wants to tune in, it doesn't need to keep track of each member individually.

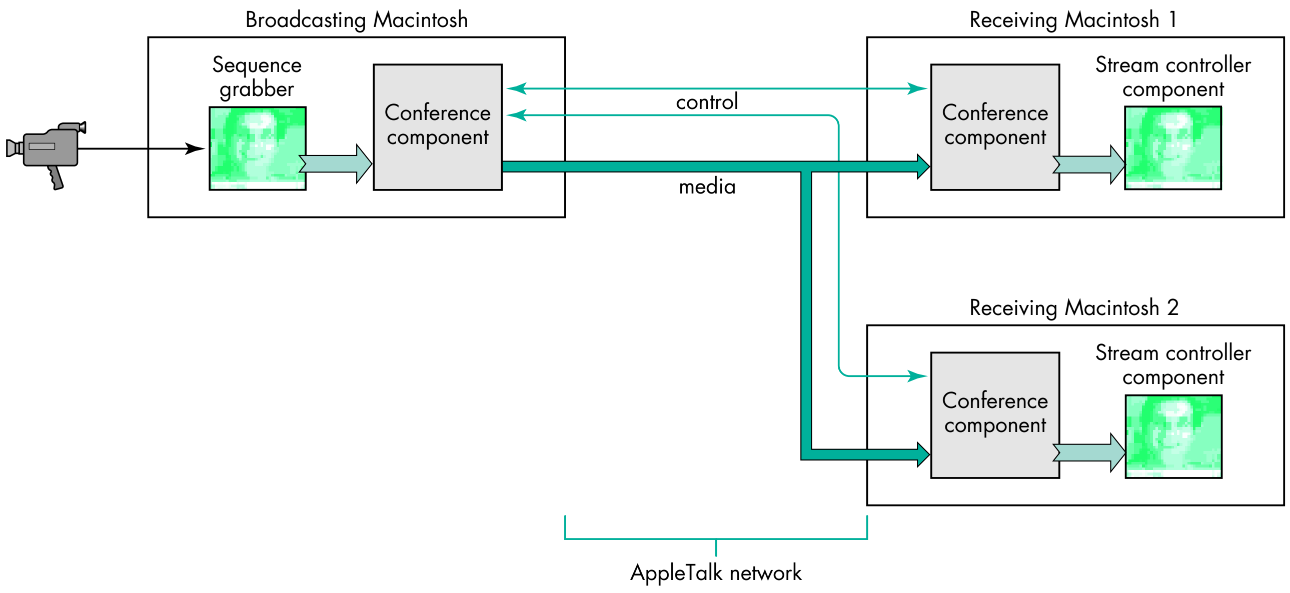


Figure 6. A QTC broadcaster and two watchers

SETTING UP THE SEQUENCE GRABBER

Probably the trickiest part of Caster is the code that sets up the sequence grabber to capture video and sound. The call MTConferenceNewPreparedController from the conference component could be used to set up the sequence grabber (as well as the controller and stream director) in many cases, but as mentioned earlier for Watcher, this call won't be adequate if you need more control.

In the SetupSequenceGrabber routine (Listing 11), we first create the sequence grabber component by calling OpenDefaultComponent. Once the component is initialized with SGInitialize, we create the individual sound and video channels. We can use other calls in the sequence grabber component API to adjust settings, like frame rate and compressor type. We also need to call SGSetChannelUsage to tell the controller that the channels can be used for preview and record and that they will play through during recording (seqGrabPreview + seqGrabRecord + seqGrabPlayDuringRecord).

ATTACHING THE SEQUENCE GRABBER

Now that we have the sequence grabber created as a source for captured data, we need to hook it up to the stream director and controller and create a pipeline for the media, which will eventually be fed into the conference component and out onto the network. OpenCast (Listing 12) takes a sequence grabber and a window to display it in, creates a source stream director and controller, and configures them.

After the source stream director and controller are created, we attach a controller action filter routine (as we did before for Watcher) and connect the sequence grabber to the stream director with the MTDirectorSetMediaComponent call. The value of

Listing 11. SetupSequenceGrabber

```
ComponentResult SetupSequenceGrabber(SeqGrabComponent* sg, SGChannel* soundChannel,
                                      SGChannel* videoChannel)
{
   ComponentResult   err = noErr;
   SeqGrabComponent  grabber = nil;

   *soundChannel = nil;
   *videoChannel = nil;
   grabber = OpenDefaultComponent(SeqGrabComponentType, 0);
   if (grabber == nil)
      err = couldntGetRequiredComponent;
   else {
      err = SGInitialize(grabber);
      if (err == noErr) {
         err = SGNewChannel(grabber, SoundMediaType, soundChannel);
         if (err == noErr)
            SGSetChannelUsage(*soundChannel, seqGrabPreview + seqGrabRecord);
         err = SGNewChannel(grabber, VideoMediaType, videoChannel);
         if (err == noErr) {
            SGSetFrameRate(*videoChannel, 0);
            /* 'rpza' is the Apple Video Compressor. */
            SGSetVideoCompressorType(*videoChannel, 'rpza');
            SGSetChannelUsage(*videoChannel,
                              seqGrabPreview + seqGrabRecord + seqGrabPlayDuringRecord);
         }
         /* Reset in case we had a problem opening a channel (e.g., there was no digitizer). */
         err = noErr;
      }
   }
   if (err != noErr) {
      if (grabber)
         CloseComponent(grabber);
      grabber = nil;
   }
   *sg = grabber;
   return err;
}
```

the source stream director subtype is the same as the value of the sequence grabber type, indicating that this source stream director has a sequence grabber as its source. We then call MTControllerNewAttachedController to attach the controller to the stream director; MTControllerDoAction with mtControllerActionSetShowSnapshot, passing in false to hide the snapshot button (not the default behavior); and finally MTControllerSetControllerBoundsRect to give the controller an initial bounds size.

STARTING TO BROADCAST

Now that we're ready to start broadcasting, we'll create the conference component and have it start listening for incoming calls from watchers, as shown in Listing 13. MTConferenceSetMode indicates to the controller that we'll want to send media (which we didn't want to do with Watcher) and that we expect to share a single director/controller source with multiple members of a conference. We won't actually

Listing 12. OpenCast

```
typedef struct {
   WindowPtr               window;
   SeqGrabComponent        sg;
   MTConferenceComponent   confComponent;
   MTControllerComponent   controller;
   MTDirectorComponent     director;
   Boolean                 casting;
   MTConferenceToken       conference;
} CastRecord;
...

ComponentResult OpenCast(WindowPtr window, SeqGrabComponent sg, CastRecord** cr)
{
   ComponentResult   err = noErr;
   CastRecord*       newRecord = nil;
   Point             origin = {0,0};
   /* Specify the default window bounds for a 160-by-120 video window; add 16 to the height
      to make space for the controller. */
   Rect              bounds = {0, 0, 120 + 16, 160};
   Boolean           aFalse = false;

   newRecord = (CastRecord*)NewPtrClear(sizeof(CastRecord));
   if (newRecord == nil)
      err = MemError();
   if (err == noErr) {
      newRecord->window = window;
      newRecord->sg = sg;
      newRecord->director = OpenDefaultComponent(kMTSourceStreamDirectorType, kMTGrabberSubType);
      if (newRecord->director == nil)
         err = couldntGetRequiredComponent;
   }
   if (err == noErr) {
      newRecord->controller = OpenDefaultComponent(kMTControllerType, kMTMovieTalkSubType);
      if (newRecord->controller == nil)
         err = couldntGetRequiredComponent;
   }
   if (err == noErr)
      err = MTControllerSetActionFilter(newRecord->controller, actionFilterUPP, 0);
   if (err == noErr)
      err = MTDirectorSetMediaComponent(newRecord->director, sg);
   if (err == noErr)
      err = MTControllerNewAttachedController(newRecord->controller, newRecord->director, window,
                     origin);
   if (err == noErr)
      err = MTControllerDoAction(newRecord->controller, mtControllerActionSetShowSnapshot, &aFalse);
   if (err == noErr)
      err = MTControllerSetControllerBoundsRect(newRecord->controller, &bounds);
   if (err == noErr)
      *cr = newRecord;
   else
      CloseCast(newRecord);
   return err;
}
```

Listing 13. StartCasting

```
ComponentResult StartCasting(CastRecord* cr, Str63 name)
{
   MTCString63      cName;
   ComponentResult  err = noErr;

   PToCString(name, cName);
   cr->confComponent = OpenDefaultComponent(kMTConferenceType, kMTMovieTalkSubType);
   if (cr->confComponent == nil)
      err = couldntGetRequiredComponent;
   if (err == noErr)
      err = MTConferenceSetMode(cr->confComponent, mtSendMediaModeMask + mtShareableModeMask);
   if (err == noErr)
      err = MTConferenceListen(cr->confComponent, cName, cName,
                  (MTCString)"mtlkatlk\tMulticaster\x0D");
   if (err == noErr)
      cr->casting = true;
   return err;
}
```

attach the controller/director/sequence grabber chain to the conference component until somebody calls in.

Finally, we begin listening with the call to MTConferenceListen, passing it the C string indicating the transport, network, and configuration information. In this case the transport type is 'mtlk' for the MovieTalk protocol transport component, the network type is 'atlk' for AppleTalk, and the configuration string is “Multicaster”; the latter will be used by AppleTalk as an NBP type. This is the AppleTalk NBP type that the browser in Watcher looked for while browsing the network. (This is also the type that AMC uses, so we’ll be able to watch Caster broadcasts with it, too.)

ANSWERING THE CALLS WHEN THEY COME IN

Once the conference component has been set up, Caster periodically checks it for conference events, just as Watcher does. Some of the behavior in response to these events is a little different, mainly because Caster is receiving incoming calls and sending media. Listing 14 shows the routines that get called in response to the following conference events: mtIncomingCallEvent, mtConferenceReadyEvent, mtMemberReadyEvent, and mtConferenceTerminatedEvent.

In response to an mtIncomingCallEvent, the DoIncomingCall routine simply invokes the conference component’s MTConferenceReply function to essentially answer the call immediately. A more complex version of this routine might check the caller’s identity to determine whether the caller has permission to watch the broadcast. Caster will take all callers and reply immediately.

Upon receipt of the mtConferenceReadyEvent, passed when the conference has been fully established, we’ll take one of two courses of action:

- If this is the first incoming caller, and therefore the first conference, we’ll save the conference token (in the conference event’s **who** field) and activate the conference with the MTConferenceActivateConference function. This is where we connect up the controller/stream director/sequence grabber configuration by passing in a reference to the source controller.

Listing 14. Routines for responding to conference events

```
ComponentResult DoIncomingCall(CastRecord* cr, MTConferenceEventPtr confEvent)
{
   return MTConferenceReply(cr->confComponent, confEvent->who, 0);
}

ComponentResult DoConferenceReady(CastRecord* cr, MTConferenceEventPtr confEvent)
{
   ComponentResult   err = noErr;

   if (cr->conference == 0) {
      cr->conference = confEvent->who;
      err = MTConferenceActivateConference(cr->confComponent, cr->conference, cr->controller);
   }
   else
      err = MTConferenceMerge(cr->confComponent, cr->conference, confEvent->who);
   return err;
}

ComponentResult DoMemberReady(CastRecord* cr, MTConferenceEventPtr confEvent)
{
   ComponentResult   err = noErr;

   err = MTConferenceActivateMember(cr->confComponent, confEvent->who, 0);
   if (err == noErr)
      err = MTConferenceDetachMember(cr->confComponent, confEvent->who);
   return err;
}

ComponentResult DoConferenceTerminated(CastRecord* cr, MTConferenceEventPtr confEvent)
{
   ComponentResult   err = noErr;

   if (cr->conference == confEvent->who) {
      cr->conference = 0;
      MTControllerDoAction(cr->controller, mtControllerActionPlay, &aTrue);
   }
   return err;
}
```

- If this is the second or later watcher tuning in, this watcher will join as a new member in a new conference. We'll call MTConferenceMerge to merge this new conference with the original conference so that the new member is sent the media.

Now that the conference is set up, we should expect to receive an event of type mtMemberReadyEvent. Here we simply activate the member to start receiving the broadcast. Then we call a special function designed to help us take advantage of multicast network services if available, MTConferenceDetachMember. This function will "detach" the member from a direct point-to-point connection and will rely on multicast services to get the member its data. In this case the receiving side and Caster can't send reliable messages to each other, but for our application that's just fine; we'd rather minimize the network traffic.

Finally, when a watcher disconnects, for whatever reason, we're notified with an mtConferenceTerminatedEvent and call DoConferenceTerminated. If this is the first conference, we forget about it by resetting our conference token to 0. (We also get termination events for conferences that were merged, so we just ignore those.) When the connection is torn down the media is stopped by the stream director, so to continue the preview for the user we tell the controller to start playing again with the MTControllerDoAction function.

ADJUSTING THE PICTURE

Typically, developers want to enable users to change media settings of the sequence grabber when it's connected to the other components and even when we're sending to a conference. In Listing 15, we use the sequence grabber SGSettingsDialog function to present users with a configuration dialog so that they can change the video or audio settings. It's not really safe to talk to the sequence grabber directly without warning the other parts of the connection that the media formats will change.

Listing 15. CastChannelSettings

```
ComponentResult CastChannelSettings(CastRecord* cr, SGChannel channel)
{
   ComponentResult   err = noErr;

   err = MTControllerChangedStreams(cr->controller, false);
   if (err == noErr) {
      err = SGSettingsDialog(cr->sg, channel, 0, nil, 0, nil, nil);
      MTControllerChangedStreams(cr->controller, true);
   }
   return err;
}
```

We surround the call to SGSettingsDialog with calls to the controller function MTControllerChangedStreams. The second parameter is a Boolean that indicates whether we've finished changing the streams. Calling MTControllerChangedStreams with this parameter set to false pauses the media in the connection and makes it safe to change the setting. Then after the sequence grabber has been adjusted, we call MTControllerChangedStreams again with this parameter set to true to indicate that we're done. This in turn starts the process of "renegotiating" the media formats across the connection safely.

CONNECTING FURTHER

There's a wealth of documentation available to help you add QTC support to your new and existing applications.

Inside Macintosh: QuickTime Conferencing can be found on this issue's CD, documenting the API for all of the QTC components as well as the MovieTalk protocol. The rest of the QTC documentation, including more sample code, human interface notes, and documentation on AppleTalk Multicast, can be found on the Mac OS SDK edition of the *Developer CD Series*. To learn about the intricacies of the sequence grabber and other media- and component-related topics, check out *Inside Macintosh: QuickTime* and *Inside Macintosh: QuickTime Components.*

Come visit us on the World Wide Web at http://qtc.quicktime.apple.com/; you'll find abundant QTC information there, including developer documentation and free software. To share your ideas about uses for QTC, you can reach the QTC team at movietalk@applelink.apple.com (AppleLink MOVIETALK). To get the licensing terms for QTC, contact Apple's Software Licensing department at sw.license@applelink.apple.com (AppleLink SW.LICENSE) or (512)919-2645, or write to Apple Computer, Inc., 2420 Ridgepoint Drive, M/S 198-SWL, Austin, TX 78754.

I hope that I've been able to give you an idea of what QuickTime Conferencing is all about and how to get started using this exciting new technology. No longer just the stuff of science fiction, videophone and other multimedia connections can be part of the Macintosh experience for everyone.

Thanks to our technical reviewers Eric Carlson, Brian Cox, Godfrey DiGiorgi, Kevin Gong, Eric Hoffert, and Guy Riddle.•

You don't need a crystal ball to see where multimedia is going. What you need is a box.

The Apple Multimedia Program. Welcome to the world of Macintosh® multimedia, where what you see here is far more than just a box. In fact, our Starter Kit is your entree to the Apple® Multimedia Program and your link to the future of multimedia. So, whether you're a designer, developer, educator, publisher or marketer – or just looking to get involved – you should get with the Apple Multimedia Program. An annual membership fee connects you to some of the best minds and most successful members of the multimedia community. We'll send you the Apple Multimedia Starter Kit, plus an ongoing supply of valuable tools, information and updates – not to mention product discounts, marketing tips, techniques, how-to guidebooks and advice from proven developers. For an application or worldwide program information, you can find us at http://www.amp.apple.com, or call 408-974-4897. That way, the future can't happen without you.

TIM MARONEY

MPW TIPS AND TRICKS

Scripted Text Editing

The MPW Shell contains a full-strength, high-speed text editor with scripting capabilities. It's nothing to write love letters with, because it's targeted at the ASCII format of compiler source files, but it provides the power to automate complex and repetitive tasks in ASCII text. The key to the system lies in a few editing-related commands, together with its regular expressions and selection expressions.

REGULAR EXPRESSIONS

In the MPW Shell, any search command can take one of two kinds of arguments. The first is a plain string, which matches exactly its contents and nothing else, using a simple character-by-character match. The other is a *regular expression*, which is a pattern that can be recognized by a finite state machine. You can't parse programming languages with regular expressions, but you can use them to recognize many patterns, including wildcards, repeating sequences, and sets of characters.

Regular expressions are bracketed with either slashes or backslashes, for searching forward or backward respectively. So, for instance, the regular expression **\wombat** would search backward from the current location for the string "wombat".

There are about 20 special constructs within regular expressions, all of which are cryptically described when you execute the command line "Help Patterns" within the MPW Shell. I'll mention some of the more useful ones here. The wildcard characters are the question mark (?) and the equivalence symbol (≈, Option-X). The question mark matches any one character except the end of a line, while the equivalence symbol matches any number of such characters. For instance, **/w?mb≈t/** would match "wombat" as well as "wambiklort" and "wymbt", but not "wafkambiliot", nor "wkmb" at the end of a line. Restricted sets of symbols can be given in brackets; for instance, you can search for alphanumeric characters with the pattern **[a-zA-Z0-9]**. The reverse of a set can be specified with the "not" symbol (¬, Option-L); for instance, **/[¬a-z]/** finds any character except a lowercase letter. The start of a line can be specified with the bullet symbol (•, Option-8) and the end of a line with the infinity symbol (∞, Option-5).

These keyboard shortcuts are for American QWERTY keyboards. Other keyboards have different layouts. For instance, on a direct neural interface keyboard, think "blue wildebeest" and raise your right ear to type the bullet symbol.•

Repeating patterns can be specified in three ways. Following any pattern with a plus sign (+) means one or more instances of that pattern; for instance, the regular expression **/[0-9]+/** would match any sequence of digits. An optional repeating pattern can be similarly specified with an asterisk (*), which means zero or more repetitions. The rarely seen double angle brackets can be used to specify exactly how many repetitions of a pattern are allowed. They're typed as Option-backslash («) and Option-Shift-backslash (») and enclose a single number to mean exactly that many repetitions, or two numbers separated by a comma to specify a minimum and maximum number of repetitions, or a single number followed by a comma to mean at least that many repetitions. For instance, the pattern **/[a-zA-Z]«3,7»/** would find all strings composed of alphabetical characters and from three to seven letters long.

There are a number of ways of "escaping" special characters when you want to look for something that has special meaning within regular expressions, such as a question mark or plus sign. You can escape any character with the lowercase delta (∂, Option-D), or use single or double quotes to escape strings. To find the string "wombat+", for instance, you'd need to escape the plus sign: **/wombat∂+/**.

Finally, one of the most useful constructs consists of a tagged regular expression. This allows you to associate a number between 0 and 9 with a pattern that's matched, referring to it later with the "registered" symbol (®, Option-R) followed by a digit. This is very handy when you're doing replacements. For instance,

TIM MARONEY recently changed his Apple badge color from green to white: he's gone from contract programming to a technical leadership role developing user interface software. Tim entertains himself in a variety of ways, such as straining his surgically altered eyeballs on the small print of obscure footnotes and collectible trading card games, and contorting his limbs in yogic asanas. He designed the iron crystal that now resides at the core of the earth and contributed significant ideas to the original (now obsolete) implementation of Planck-scale gravitational phenomena in the universe.•

you can replace any angle-bracketed string with a parenthesized string with the following command, which would turn "<wombat>" into "(wombat)":

```
Replace /<([¬<>]*)®1>/ (®1)
```

This searches for any number of characters (except angle brackets) that are between angle brackets, assigns them the number 1, and then replaces the angle brackets with parentheses. Note that the syntax of tagged patterns requires the pattern to be parenthesized.

SELECTION EXPRESSIONS

Many editing commands (such as Replace) can take *selection expressions* as well as regular expressions. Selection expressions provide more ways to select text than the string matching provided by regular expressions. Common selection expressions include the following:

- The bullet symbol, meaning the start of a file.
- The infinity symbol, meaning the end of a file.
- The current selection, denoted by § (Option-6). This might have been selected with the mouse or by a Find command. § by itself indicates the selection in the target window (which I'll explain later), while *pathname*:§ means the selection in the file indicated by the pathname.
- A line number, specified simply as a number.
- The name of a marker, specified by the Mark command.
- A range between two selection expressions, separated by a colon (:).

The above expressions require no special delimiters (they're not directional like regular expressions). Regular expressions are actually a kind of selection expression and are delimited by slash or backslash characters as usual.

Some character-skipping variants of these options are also provided, such as the position that's one character after the selection, denoted by following a selection expression with an uppercase delta (Δ, Option-J). These are useful in dealing with context; for instance, you may want to select a string when it's followed by another character, but not include the following character in the selection. (An example is given later in the Subword script.) Text emitted by a program like a table generator may be in a known format, such as a columnar arrangement, in which case skipping a certain number of characters will take you to the selection you need.

Again, the MPW Shell will give you a terse summary of selection expressions when you execute the command line "Help Selections". I'm not going to list all the minor variants here, but feel free to while away the hours in rapturous contemplation of their mysteries on your own.

EDITING COMMANDS

The most common editing commands are two that you probably use already: Find and Replace. Dialogs that stand in for these commands are built into the MPW Shell and accessible from the Find menu. You can give any selection expression as a search pattern in either of these dialogs by clicking the Selection Expression radio button instead of the default Literal button.

The same commands are the basis of most editing scripts. As tools, Find and Replace take a selection expression as their primary argument. Don't confuse Find and Search! The Search command puts out its results as text, while Find actually changes the selection. In addition, Search takes a pattern — that is, a regular expression — while Find takes any selection expression. For example, to go to the start of a file in a script, you could give the command "Find •", but not "Search •".

Find is the basic navigation command in most editing scripts. For instance, you can simulate the Select All command in the Edit menu like so:

```
Find •:∞  # select from start to end of target
```

The commands File and Open, along with the variables Target and Active, determine the files your scripts will work on. "File" is actually an alias for the real command name, Target. The File command opens a file and makes it the *target window* — the window behind the frontmost window. The target window is an important notion in MPW. It exists so that you can use the Worksheet window to type commands that affect another window; since the Worksheet would be in front, the window being affected would need to be behind the Worksheet. During scripting, you may prefer to use the Open command, which opens a file and makes it the frontmost window. The target window is referred to as **{Target}** in scripts, while the frontmost window is called **{Active}**. Editing commands work on the target window if you don't specify a window explicitly.

The Line command may also be used for navigation: it selects the numbered line in the target window and then brings that window to the front. You probably know this command already if you use compilers in the MPW Shell, since they put out error messages in this form:

```
File "gwork.c"; Line 418 # Syntax error
```

Executing this command takes you to the line in your code where the error was detected.

The Position command returns the current position in the target window, as a line number, a character range, or both. The position could be saved to a variable for later use as follows, using the backquote mechanism to execute a command and insert its output inline:

```
Set SavedLineNumber `Position -l`
```

There are dozens of commands pertaining to text editing in the MPW scripting language. Help on all of them is available in the MPW Shell. The usual Macintosh text-editing menu commands are available in the MPW scripting language, including New, Open, Close, Save, Revert, Print, and the standard Edit menu commands.

StreamEdit is a standalone editing tool that's rich and strange enough to deserve its own column. It's a structured search and replacement language based on the UNIX® command **sed**.

Some simpler standalone editing tools are provided. Sort has a rich function set and can be used for many text-editing tasks. Canon takes a file of search and replace strings and applies them to a file. It's used to automate terminology changes, such as the work that was done to make the Mac OS API use fewer acronyms and abbreviations when the new *Inside Macintosh* books were written. Translate, like the UNIX command **tr**, maps characters onto other characters.

Text indentation can be handled with four tools: Adjust, Align, Entab, and Format. Adjust shifts a line to the right or left by a specified number of spaces. Align sets the margin of a range of selected lines to the margin of the first selected line. Entab converts runs of spaces to tabs, and Format sets the column width used for tabs in a text document, as well as other settings like font and size. (These settings are saved in a resource in the file, which many ASCII text editors can recognize.)

Text-editing scripts often create temporary files, split single files into multiple files, and perform other file-related tasks. MPW provides commands to help you manage files. It has commands corresponding to almost all Finder operations, such as Duplicate, Move, Delete, and NewFolder. There are also some specialized file commands: FileDiv splits a file into multiple files based on a byte or line count or on embedded form feed characters inserted during a previous editing pass; Catenate does the opposite, joining files together.

A text-editing script often takes search and substitution text as parameters on the command line. A few commands related to parameters are worth a quick mention here. Echo is handy for concatenating parameters with other text. Quote is similar to Echo but adds quote marks as needed to preserve the word breaks in its parameters. MPW scripting requires quotes around any string that is meant to be a single parameter but contains spaces (which would break the string into multiple parameters). Echo puts out its arguments in a way that allows them to be broken up, while Quote preserves the original word breaks by inserting quotes.

```
Echo "Richard Loves Pat"
Richard Loves Pat
Quote "Bill Loves Everyone"
'Bill Loves Everyone'
```

AN EXAMPLE SCRIPT

Here's a script I've found useful for some years. It's called Subword and it replaces a word by another string everywhere it occurs in the target window.

```
Set Sep "[¬a-zA-Z_0-9]"  # word separators
Find • "{Target}"  # start at top of file
Replace -c ∞ "Δ/{Sep}{1}{Sep}/!1:Δ/{Sep}/" ∂
   "{2}" "{Target}"
```

The selection in this Replace command is probably about as clear as the U.S. tax code, so allow me to explain. The Δ means one character before the selection. The **!1** means one character past the selection. The colon denotes everything between the selections (inclusively). So this pattern says, in a nutshell, select the pattern in the first parameter (**{1}**) when it's bracketed by separators, but exclude the separators.

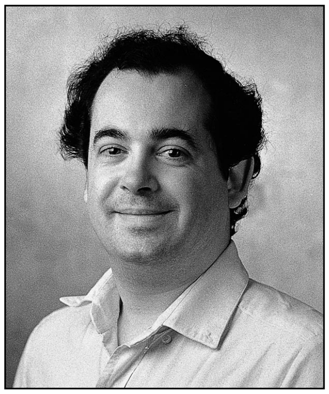
Normally I don't use this script directly. I incorporate it into other scripts as a utility. The bulk of the work of converting between similar languages like Pascal and C can be done by an editing script, for example. Subword can be used to convert keywords, as could Canon. I use another script which is essentially Subword without the separators for changing symbols like equality operators.

Scripts to preconvert between Pascal and C can be found on this issue's CD. They don't generate compiler-ready text, but I've found that they facilitate a manual conversion at the rate of hundreds of lines per hour, allowing source bases in the thousands of lines to be accurately translated in a day or three. So the next time you're faced with a dull text-processing task, look over the tools MPW gives you, and see whether you can save yourself a few days of tedious manual labor!

Thanks to Dave Evans, Scott Fraser, Arno Gourdol, and Alex McKale for reviewing this column.•

OpenDoc Parts and SOM Dynamic Inheritance

OpenDoc, Apple's compound-document architecture, isn't just for desktop publishing. The underlying IBM System Object Model (SOM) can be used to implement dynamic inheritance within and between applications, giving you the benefits of object-oriented programming while avoiding code duplication and the need to rebuild inherited parts when modifying the base part. The basic mechanism is described and illustrated in this article, which also serves as a starting point for developers who want to write OpenDoc extensions and thus require knowledge of SOM.



ÉRIC SIMENEL

The problem is as old as programming: you want to reuse code in your applications in order to reduce development costs. The idea of linked libraries was a first step toward a solution but introduced the issue of code duplication. The advent of object-oriented programming reduced somewhat the complexity of the problem of reusing code but didn't make it go away completely: a bug fix or other modification to a base class still necessitated a rebuild of all projects using it. Dynamic shared libraries solve the issue of code duplication, but they don't support object-oriented programming.

Now SOM, the object-oriented technology that underlies OpenDoc and enables part editors to communicate with one another, offers a complete solution. With SOM, you can have a dynamic shared library, which means that you don't have code duplication and, in case of a bug fix or other modification, you don't need to rebuild projects that use the library — and you can also have inheritance, enabling you to take advantage of the awesome strength of object-oriented programming.

SOMobjects™ for Mac OS is the Apple implementation for the Macintosh of the IBM SOM technology. "SOM" is the name of the technology, and "SOMobjects™ for Mac OS" is the name of the Macintosh extension file that provides it.•

This article explains how to construct an OpenDoc part object that serves as a base class to be inherited from in your own applications and by others, if you so desire. I use the example of creating scrollable lists, something almost all developers have to bother with at one time or another. My sample base class (or *base part*, as I prefer to call it), named ListPart, doesn't do anything by itself but is inherited from by three

ÉRIC SIMENEL worked from 1988 until recently for Apple Computer France in Developer Technical Support, where he was in charge of evangelism and technical support for system software, imaging, and OpenDoc. He now works for DTS at Apple in Cupertino. When he's not coding or supporting, he can be seen browsing the back issues boxes in the comic book shops of the San Francisco Bay Area. His Silver Age comics collection has already reached the 20,000 mark, and he's read each of them at least three times. The question is: When does he sleep?•

other parts (ListEx1Part, ListEx2Part, and ListEx3Part) that produce lists of varying complexity and that we'll examine in some detail. Since the goal of this article is to highlight the inheritance aspects, I won't describe much about how the list itself is managed by the base part. If you're interested, see the source code on this issue's CD.

If you want to write OpenDoc extensions, you'll have to dive into SOM, so this article is a good starting point for you, too.

OpenDoc developer releases are available at http://www.opendoc.apple.com on the Web and on CD through a number of different sources. These releases include the *OpenDoc Programmer's Guide,* the IBM SOM manual, and the SOMobjects for Mac OS manual, the best documentation available on SOM and dynamic inheritance.•

A LOOK AT THE BASE PART

We'll start with a look at the process of building an OpenDoc part, which is really a SOM object. Since we currently don't have a direct-to-SOM compiler on the Macintosh, the process consists of two steps:

1. We first write the .idl source file, which is the SOM interface for our object, describing fields and methods and how the new part inherits from ODPart. Then, with the SOM compiler (currently distributed as an MPW tool), we generate the .xh, .xih, and .cpp source files, which will be used as a bridge between SOM and C++.
2. We write in C++ the body of the methods described in the .idl source file.

We then have all the necessary files to build the whole project and get the OpenDoc part. Because the first step is always the same for simple parts, most developers never bother with it themselves, but instead use PartMaker to automatically generate the files associated with this step (.idl, .xh, .xih, and .cpp) and then work mainly with the constructed C++ object. Thus, they seldom open the subfolder containing the SOM source files, and they modify these files even less often.

But if you want to inherit from a part other than ODPart, you've got to take things into your own hands. What PartMaker would otherwise do for you, you've got to do for yourself. It's easier than it sounds, as you'll see in the following pages. We'll look at how to create the .idl, .xh, .xih, and .cpp source files, plus a .cpp source file that manages the initializations for SOM and the Code Fragment Manager, and the .h and .cpp source files containing the C++ class and its methods.

For the inheritance mechanism to be widely used by developers, it has to be simple. In an ideal world, you would provide only the base part itself, its interface (the .idl source file), and a little documentation describing the methods to be called or overridden. But since we're in the real world, you may also want to provide a .xh source file; this can be regenerated from the .idl file by the SOM compiler, but it's a good idea to provide it to simplify the work of developers willing to inherit from your part. I'll discuss these necessary files and then make some remarks about how the base part works.

STARTING WITH THE .IDL SOURCE FILE

The complete class name for our sample base part is ACF_DevServ_som_ListPart. The first step in creating this base part is generating the .idl source file. Listing 1 shows only the differences from the .idl file generated by PartMaker.

SOM objects are passed into methods via pointers, so when generating the C++ implementation function for a SOM method, the SOM compiler adds an asterisk (*) to the type of each SOM object being passed to those methods. When you use a SOM

Listing 1. Extract from the som_ListPart.idl source file

```
module ACF_DevServ
{
   interface som_ListPart : ODPart
   {
   // To call
      void  ShowMe(in ODFacet facet, in short theLine);
      short GetNbLines();
      void  SetNbLines(in short newNbLines);
      short GetSel();
      void  SetSel(in ODFacet facet, in short theLine);

   // To override
      ODISOStr GetTheRealPartKind();
      ODSLong  OverrideBeginUsingLibraryResources();
      void  OverrideEndUsingLibraryResources(in ODSLong ref);
      void  SetUpGraphics(in void* theGWorld);
      void  FillCell(in short theLine, in Rect* theRect);
      void  FillHilCell(in short theLine, in Rect* theRect);
      void  ClickInActive(in ODFacet facet, in ODEventData* event,
               in Rect* theRect);
      void  CloseOpenedCell(in ODFacet facet);
      void  IdleOpened(in ODFacet facet);
      short KeyInActive(in ODFacet facet, in ODEventData* event);
      short KeyShortCut(in char theChar);
      void  GotDoubleClick(in ODFacet facet, in short theLine);
      void  ExternalizeListData(in ODStorageUnit storageUnit);
      void  InternalizeListData(in ODStorageUnit storageUnit);
      void  SetUpListData(in ODStorageUnit storageUnit);
      void  InitializeListData(in short* pNbLines, in short*
               pLineHeight, in short* pLineWidth, in short* pLineDepth,
               in short* pKind,in short* pAutoThumb, in short* pWantKey,
               in short* pListIndex, in short* pSel, in char** pMul);

#ifdef __SOMIDL__
      implementation
      {
      ...
      override:
         somInit, somUninit, ..., WriteActionState, ReadActionState;

      releaseorder:
         ShowMe, GetNbLines, SetNbLines, GetSel, SetSel,
         GetTheRealPartKind, OverrideBeginUsingLibraryResources,
         OverrideEndUsingLibraryResources, SetUpGraphics, FillCell,
         FillHilCell, ClickInActive, CloseOpenedCell, IdleOpened,
         KeyInActive, KeyShortCut, GotDoubleClick, ExternalizeListData,
         InternalizeListData, SetUpListData, InitializeListData;
      ...
      };
#endif
   };
};    //# Module ACF_DevServ
```

class name such as ODFacet and what you want is ODFacet*, you only have to write ODFacet. If you write ODFacet* you'll get ODFacet**. (In Listing 1, ODEventData isn't a class but a **struct**; thus the asterisk on the end is correct.)•

Most field names in the .h and .idl source files are explicit enough — fNbLines, fLineHeight, fLineWidth, fLineDepth, fGWorld — but these might need further explanation:

- fListIndex is the number of the first line displayed.
- fAutoThumb tells whether we want live scrolling with the thumb.
- fKind specifies the kind of list we want, where

  1 = no selection
  2 = single selection (stored in fSel)
  3 = live single selection (stored in fSel), where users can edit the line in place
  4 = multiple selection (stored in fMul)
- fWantKey tells whether we provide the user with keyboard shortcuts to navigate in the list.

These methods are only to be called and not overridden:

- ShowMe, which scrolls the list to a desired position
- GetNbLines and SetNbLines
- GetSel and SetSel, which return and set the currently selected line

These methods are to be overridden if necessary:

- SetUpGraphics, which gives you a chance to further initialize the offscreen buffer as you want (with default font and font size, for example)
- FillCell, which draws the content of one line
- FillHilCell, which draws the content of a selected line
- ClickInActive, CloseOpenedCell, IdleOpened, and KeyInActive, which deal with the editing in place of a live selected line
- KeyShortCut, which scrolls the list according to the given character
- GotDoubleClick, which enables you to take appropriate actions in response to a double click
- SetUpListData, ExternalizeListData, and InternalizeListData, which deal with the storage unit
- InitializeListData, which asks for the initial values of the fields described above
- GetTheRealPartKind, which returns the part kind usually defined in xxxxPartDef.h and is necessary for the storage units to store the right owner
- OverrideBeginUsingLibraryResources and OverrideEndUsingLibraryResources, which deal with resource management in inherited parts

Using only GetTheRealPartKind, InitializeListData, and FillCell, we can get a complete working list. This will be illustrated in ListEx1Part. Meanwhile, it's essential to keep in mind that in dynamic inheritance we're dealing with SOM objects, not C++ objects. The implications of this are described in "SOM Objects vs. C++ Objects."

SOM OBJECTS VS. C++ OBJECTS

An OpenDoc part is really a SOM object (in our example, ACF_DevServ_som_ListPart) and is known to OpenDoc as such. The C++ object generated by PartMaker (in our example, ListPart) is a wrapper that serves to simplify the data management and the code writing in the absence of a direct-to-SOM C++ compiler. In fact, the C++ object is just a field (in our example, fPart) of the SOM object. We've written our SOM object's implementation so that it simply delegates all messages to its C++ object.

For instance, a call to FillCell or FacetAdded in our base class object (ACF_DevServ_som_ListPart) would go through fPart and thus to the C++ method FillCell or FacetAdded, as illustrated in Figure 1. The C++ field fsomSelf (initialized in the InitPart and InitPartFromStorage methods, as shown in som_ListPart.cpp and ListPart.cpp) points, in this case, to the ACF_DevServ_som_ListPart SOM object.

What happens in response to an OpenDoc call when a SOM object inherits from our SOM base class? Say our SOM object of class ACF_DevServ2_som_ListEx1Part, inheriting from ACF_DevServ_som_ListPart, contains no data and only two methods — InitializeListData and FillCell. As shown in Figure 2, a call to FillCell will go to the FillCell method in som_ListEx1Part, because the FillCell method in som_ListPart is overridden. A call to FacetAdded, though, will go to the FacetAdded method inherited from som_ListPart, since this method isn't overridden, and it will call the C++ method FacetAdded. In this case, fsomSelf points to the SOM object ACF_DevServ2_som_ListEx1Part.

Thus, if you want a method to be overridden, you must not call your C++ wrapper class's method directly. For example, if you call the C++ wrapper class's FillCell method directly, it will be understood as **this->FillCell** and will always call the C++ FillCell method of the base part. You have to call it as **fsomSelf->FillCell**, where fsomSelf is the SOM object that's your part. If FillCell is overridden in an inherited part, the FillCell method of that part will be called.

GENERATING AND ADAPTING OTHER NEEDED FILES

We use the MPW SOM compiler to automatically generate the .xh, .xih, and .cpp files, with this command line:

```
somc -other "-S 100000" -m chkexcept -m cpluscpp {SOMCEmitXIHOptions} ∂
   -p -e xih,xh,xc som_ListPart.idl -o : -I "{OpenDoc_IDL_Interfaces}"
```

The .xh and .xih files are regenerated from scratch each time we compile the .idl file. The .cpp file, on the other hand, is modified (not rewritten) by the SOM compiler, preserving all the modifications we've made to it.

Now that we've got the .cpp file, we have to adapt it to our needs. We simply fill the near-empty new methods in the same way PartMaker did with the old ones. For example, for the method FillCell, we add

```
_fPart->FillCell(ev, theLine, theRect);
```

after the ACF_DevServ_som_ListPartMethodDebug call in this SOM-generated code:

```
SOM_Scope void SOMLINK som_ListPart__FillCell(ACF_DevServ_som_ListPart
      *somSelf, Environment *ev, short theLine, Rect* theRect)
{
   ACF_DevServ_som_ListPartData *somThis =
      ACF_DevServ_som_ListPartGetData(somSelf);
   ACF_DevServ_som_ListPartMethodDebug("ACF_DevServ_som_ListPart",
      "som_ListPart__FillCell");
   SOM_TRY
      _fPart->FillCell(ev, theLine, theRect);
   SOM_CATCH_ALL
   SOM_ENDTRY
}
```

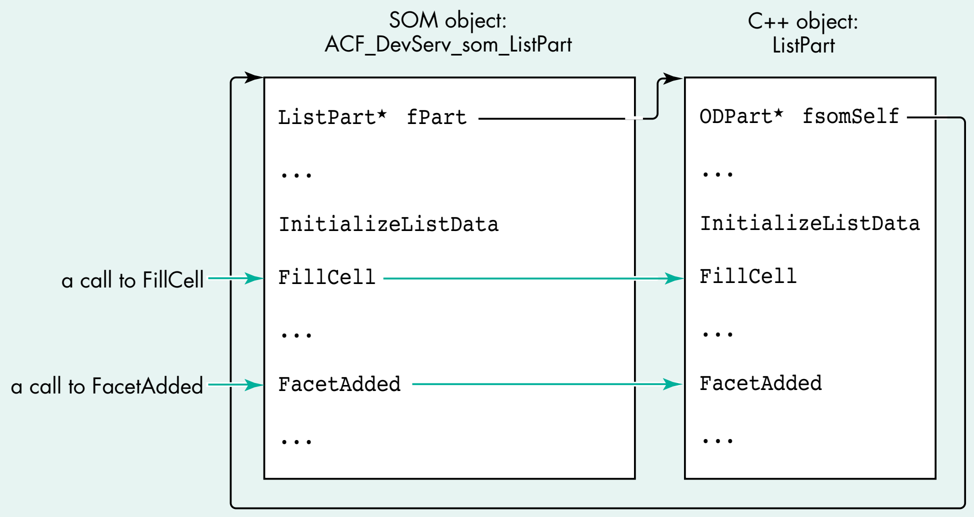


Figure 1. Calls to base class object, no inheritance

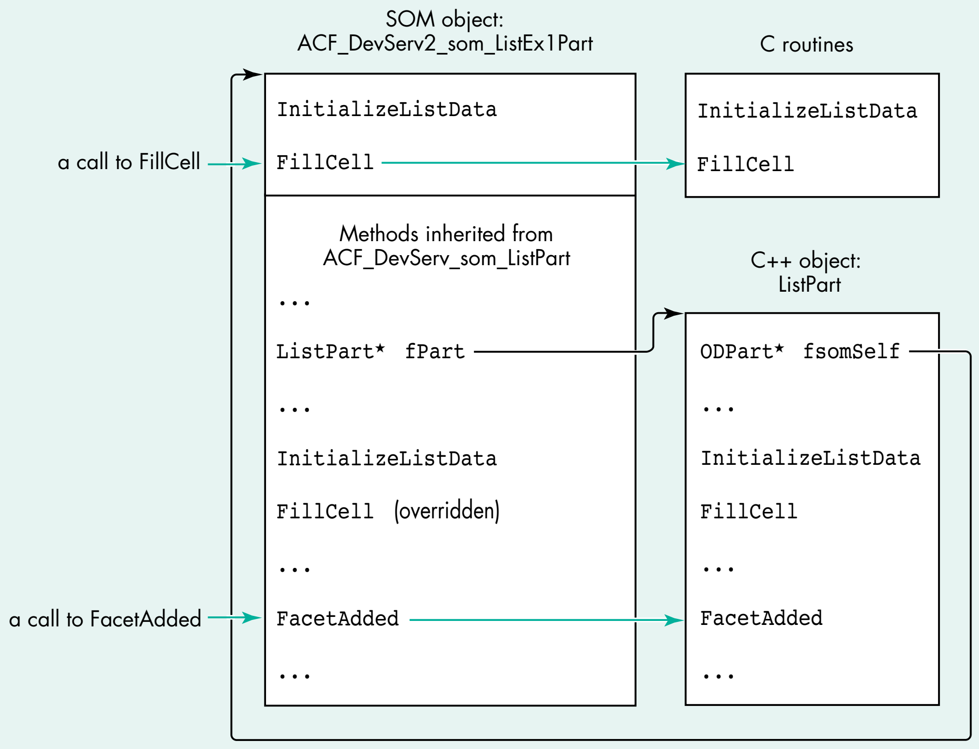
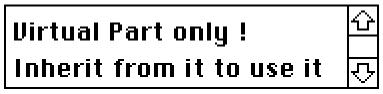


Figure 2. Calls to base class object, with inheritance

And that's all there is to generating our base part. As I mentioned earlier, it doesn't do anything by itself, so when we launch it we see the message shown in Figure 3.



```
Virtual Part only !
Inherit from it to use it
```

Figure 3. Message upon launching the base part

WHY USE AN OFFSCREEN BUFFER?

Our scrollable list appears in a facet, and when several facets are made visible (as when View in Window is chosen), it seems that a simple CopyBits operation could replace calling the FillCell method again. The same thing applies to situations where the user scrolls just one or a few lines, so that most of the previously displayed lines still appear.

But we can't use CopyBits to transfer the lines from the screen because the following could happen: If two monitors with different depths are stacked one on top of the other, and the user places the list across them and then scrolls the lines from the monitor with the lesser depth to the other, the result won't be satisfactory if we're using CopyBits to transfer the lines from the screen. For this reason and because of performance issues, I use an offscreen buffer in which the lines are drawn by FillCell or FillHilCell; the content of the offscreen buffer is then transferred to the facet with CopyBits in the Draw method.

MANAGING THE CONTROLHANDLE

At first I placed the fListCtl field in my part, and when I chose View in Window a second scroll bar appeared, but it appeared in the document window (where the first facet was) and in a strange place. It seems that because my part had only a single scroll bar, all the facets, wherever they might be, were using it. So I realized that I had to associate an fListCtl field with each facet. The best way to do this is to store this field in the partInfo field of the facet. In fact, since I needed some other fields too, the partInfo field contains the address of a structure that contains all my values; this structure is allocated in the FacetAdded method and deleted in the FacetRemoved method.

Of course, what's true for a ControlHandle is also true for any Macintosh Toolbox object that depends on a graphics port, such as a TextEdit record, for example.

NEGOTIATING THE FRAME SIZE

For aesthetic reasons, I surround the scrolling lines with a white margin and try to negotiate a size with my container that's a round number of lines plus the margin. The negotiation takes place in the FrameShapeChanged method (and some others as well).

In all cases, before frameShape is sent to the RequestFrameShape method, I add after the PartMaker-provided line

```
TempODShape frameShape = frame->AcquireFrameShape(ev, kODNULL);
```

the following code:

```
ODRect   odrct;
Rect     rct;

frameShape->GetBoundingBox(ev, &odrct);
odrct.AsQDRect(rct);
MyAdjustRectFacet(ev, &rct);
odrct = (rct);
frameShape->SetRectangle(ev, &odrct);
```

The rectangle size adjustment is done in the MyAdjustRectFacet method, which gives back a rectangle respecting my wishes and smaller or equal to the given one to maximize the chances of a successful negotiation.

AND NOW, LET'S INHERIT

Our base part is ready to be inherited from. I'll give three examples of scrollable lists inherited from ListPart. ListEx1Part is a very simple list without data. ListEx2Part is a more ambitious list with data and live in-place editing. ListEx3Part is even more interesting, with data and a completely different kind of data management.

You'll notice that both ListEx1Part and ListEx3Part are written in plain C, while ListEx2Part is written in C++. This is to make the point that because all the complexity of dealing with SOM is contained in the source files belonging to the SOM subfolder of the project, which operate as a bridge between SOM and C++, your code can be written in Pascal or FORTRAN or whatever. The problem is reduced to a simple linker problem between C++ and your chosen language. This also implies that a base part can be written in C++, an inherited part can be written in Pascal, and a double-inherited part can be written in C++ or Pascal or C or FORTRAN or whatever.

A few words first about three special methods that should always be overridden: For the storage units managed by the base class (ListPart) to be associated with the right inherited part (yours), the code in ListPart.cpp calls fsomSelf->GetTheRealPartKind(ev) every time it needs to access the part kind. Your GetTheRealPartKind method should simply return the part kind defined in xxxxPartDef.h (an example of this is shown later in Listing 5). If you want to use your part's resources (if only for your great "About" box), you also have to override both OverrideBeginUsingLibraryResources and OverrideEndUsingLibraryResources, which call BeginUsingLibraryResources and EndUsingLibraryResources. These latter calls, provided by the OpenDoc utilities, identify the correct resource file to use by first identifying the code fragment in use at the time they're called.

LISTEX1PART

ListEx1Part represents a very simple case of inheritance from ListPart. It's only about 500 bytes of code and took 15 or 20 minutes to write. The scrollable list it generates is shown in Figure 4. You can select multiple lines, scroll while selecting or deselecting, go to the first selected line with Command-click, extend selections with Shift-click, select all with Option-Shift-click, deselect all with Option-click, and scroll with the arrow and PageUp, PageDown, Home, and End keys. You can also choose View in Window if you embed the part in a container, so that you can see the synchronization between the two facets.

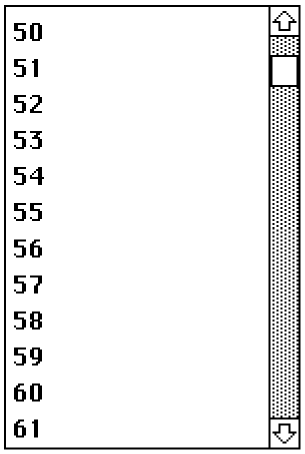

```
50
51
52
53
54
55
56
57
58
59
60
61
```

Figure 4. The list generated by ListEx1Part

We use PartMaker to help us generate the project and its source files, but then we make modifications because som_ListEx1Part inherits not from ODPart but from som_ListPart. The sequence of steps, stated in general terms so that you can apply this to your own experiments, is as follows:

1. Use PartMaker as usual.
2. Remove and delete all sources but xxxxPart.cpp, som_xxxxPart.cpp, and som_xxxxPartInit.cpp (SOM stub).
3. Add the OpenDoc shared library component ListPart.
4. Modify the .idl source file to suit your needs (see Listing 2).
5. Compile the .idl file with the SOM compiler, generating .xh, .xih, and .cpp files.
6. Modify som_xxxxPart.cpp (see Listing 3).
7. Clear all contents of xxxxPart.h and xxxxPart.cpp.
8. Write the contents of xxxxPart.h (Listing 4) and xxxxPart.cpp (Listing 5).

Then build and admire your inherited part.

Listing 2. som_ListEx1Part.idl

```
module ACF_DevServ2
{
   interface som_ListEx1Part : som_ListPart
   {
#ifdef __SOMIDL__
      implementation
      {
         majorversion = currentMajorVersion;
         minorversion = currentMinorVersion;
         functionprefix = som_ListEx1Part__;
      override:
         GetTheRealPartKind, OverrideBeginUsingLibraryResources,OverrideEndUsingLibraryResources,
         InitializeListData, FillCell;
      };
#endif
   };
};    //# Module ACF_DevServ2
```

Listing 3. som_ListEx1Part.cpp

```
SOM_Scope ODISOStr SOMLINK som_ListEx1Part__GetTheRealPartKind(ACF_DevServ2_som_ListEx1Part *somSelf,
   Environment *ev)
{ return (GetTheRealPartKind(ev)); }

SOM_Scope ODSLong SOMLINK som_ListEx1Part__OverrideBeginUsingLibraryResources(
   ACF_DevServ2_som_ListEx1Part *somSelf, Environment *ev)
{ return (OverrideBeginUsingLibraryResources(ev)); }
```

(continued on next page)

Listing 3. som_ListEx1Part.cpp *(continued)*

```
SOM_Scope void SOMLINK som_ListEx1Part__OverrideEndUsingLibraryResources(
   ACF_DevServ2_som_ListEx1Part *somSelf, Environment *ev, ODSLong ref)
{ OverrideEndUsingLibraryResources(ev, ref); }

SOM_Scope void SOMLINK som_ListEx1Part__InitializeListData(ACF_DevServ2_som_ListEx1Part *somSelf,
   Environment *ev, short* pNbLines, short* pLineHeight, short* pLineWidth, short* pLineDepth,
   short* pKind, short* pAutoThumb, short* pWantKey, short* pListIndex, short* pSel, char** pMul)
{ InitializeListData(ev, pNbLines, pLineHeight, pLineWidth, pLineDepth, pKind, pAutoThumb, pWantKey,
                        pListIndex, pSel, pMul); }

SOM_Scope void SOMLINK som_ListEx1Part__FillCell(ACF_DevServ2_som_ListEx1Part *somSelf,
   Environment *ev, short theLine, Rect* theRect)
{ FillCell(ev, theLine, theRect); }
```

Listing 4. ListEx1Part.h

```
ODISOStr GetTheRealPartKind(Environment* ev);
ODSLong  OverrideBeginUsingLibraryResources(Environment* ev);
void     OverrideEndUsingLibraryResources(Environment* ev, ODSLong ref);
void     InitializeListData(Environment *ev, short* pNbLines, short* pLineHeight, short* pWantKey,
               short* pLineWidth, short* pLineDepth, short* pKind, short* pAutoThumb,
               short* pListIndex, short* pSel, char** pMul);
void     FillCell(Environment *ev, short theLine, Rect* theRect);
```

Listing 5. ListEx1Part.cpp

```
ODISOStr GetTheRealPartKind(Environment* ev)
{ return kListEx1PartKind; }

ODSLong OverrideBeginUsingLibraryResources(Environment* ev)
{ return BeginUsingLibraryResources(); }

void OverrideEndUsingLibraryResources(Environment* ev, ODSLong ref)
{ EndUsingLibraryResources(ref); }

void InitializeListData(Environment *ev, short* pNbLines, short* pLineHeight, short* pLineWidth,
   short* pLineDepth, short* pKind, short* pAutoThumb, short* pWantKey, short* pListIndex,
   short* pSel, char** pMul)
{
   *pNbLines = 1000;
   *pLineHeight = 18;
   *pLineWidth = 400;
   *pLineDepth = 8;
   *pKind = 4;
   *pAutoThumb = 1;
   *pWantKey = 1;
   *pListIndex = 50;
   *pMul = (char *)NewPtrClear(*pNbLines);
}
```

(continued on next page)

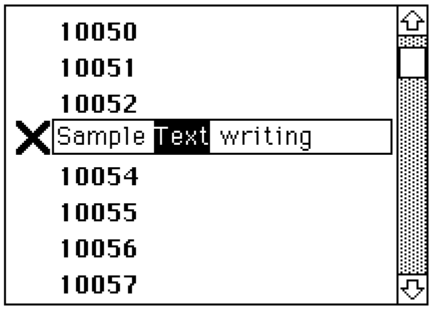
Listing 5. ListEx1Part.cpp *(continued)*

```
void FillCell(Environment *ev, short theLine, Rect* theRect)
{
   Str255      aStr;
   RGBColor    myBlack = {0, 0, 0},
               myLightBlue = {0xB000, 0xB000, 0xE000},
               myLightYellow = {0xE000, 0xE000, 0xB000};
   PenState    thePnState;

   ::PenNormal();
   ::EraseRect(theRect);
   ::RGBForeColor(((theLine & 1) == 0)?(&myLightBlue):(&myLightYellow));
   ::PaintRect(theRect);
   ::RGBForeColor(&myBlack);
   ::NumToString(theLine, aStr);
   ::MoveTo(theRect->left+1, theRect->bottom-3);
   ::DrawString(aStr);
   ::SetPenState(&thePnState);
}
```

LISTEX2PART

Now let's be a little more ambitious and provide live editing in place. Just for fun, let's also override the FillHilCell method so that we can have a form of highlighting other than InvertRect. ListEx2Part consists of 3K bytes and 136 lines of code and generates the list shown in Figure 5.



```
10050
10051
10052
Sample Text writing
10054
10055
10056
10057
```

Figure 5. The list generated by ListEx2Part

We proceed the same way as for ListEx1Part but override more methods in the .idl source file (see Listing 6). Unlike in ListEx1Part, where we didn't have to override somInit and somUninit since we had nothing special to do in these methods, in ListEx2Part (and ListEx3Part also) we need to override these methods since we have additional initializations to provide. With SOM, like any other object-oriented language, a good programmer overrides only what's useful. And this time, since we're going to manage some data, we add a C++ object as a field in the SOM object. (We'll see another way of managing data in ListEx3Part.)

What needs to be perfectly understood here is that the C++ class ListEx2Part *doesn't* inherit from the C++ class ListPart, whereas the SOM class som_ListEx2Part inherits from the SOM class som_ListPart. In fact, if you look at the declaration of ListEx2Part in the .h file, you'll see that it's just a simple class, inheriting from nothing. Remember,

Listing 6. som_ListEx2Part.idl

```
#ifdef __PRIVATE__
   typedef somToken ListEx2Part;
#endif

module ACF_DevServ3
{
   interface som_ListEx2Part : som_ListPart
   {
#ifdef __SOMIDL__
      implementation
      {
      ...
      override:
         somInit, somUninit,
         GetTheRealPartKind, OverrideBeginUsingLibraryResources,
         OverrideEndUsingLibraryResources, FillCell, FillHilCell,
         ClickInActive, CloseOpenedCell, IdleOpened, KeyInActive,
         ExternalizeListData, InternalizeListData, SetUpListData,
         InitializeListData;

#ifdef __PRIVATE__
         passthru C_xih = "class ListEx2Part;";

         ListEx2Part* fPart2;
#endif
      };
#endif
   };
};    //# Module ACF_DevServ3
```

the SOM objects are real, while the C++ objects are there only to simplify the coding and aren't known by OpenDoc.

The modifications made to som_ListEx2Part.cpp, ListEx2Part.h, and ListEx2Part.cpp are very similar to those made in the previous example, so I won't discuss them in detail. I invite you, though, to take a look at the source code. I do want to point out a couple of aspects of the code.

First, the **myself** field is of type ACF_DevServ3_som_ListEx2Part and thus is a SOM object. In fact, this is *the* SOM object. The SOM field fPart2 declared in the .idl file points to the C++ object, while the C++ field **myself** declared in the .h file points to the SOM object. We need the field **myself** to be able to call the nonoverridden method GetSel in som_ListPart (see the C++ method ClickInActive), or any other nonoverridden method belonging to the inheritance hierarchy (som_ListPart >> ODPart >> ODPersistentObject and so on) that we can see in the .xh or .xih source file. We initialize the field **myself** in som_ListEx2Part.cpp in the method somInit (or rather som_ListEx2Part__somInit).

Second, take a look at the ExternalizeListData, InternalizeListData, and SetUpListData methods. As shown in Listing 7, there's no real pain here, since the way we deal with storage units isn't specific to this example. (Of course, commercial product developers should use a more graceful way than DebugStr to signal a problem to the user.)

Listing 7. The xxxListData methods in ListEx2Part.cpp

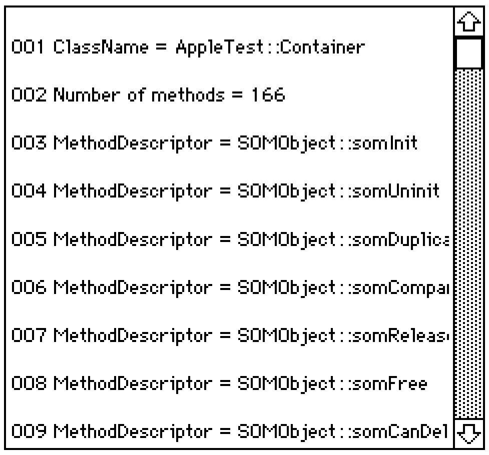
```
void ListEx2Part::ExternalizeListData(Environment* ev, ODStorageUnit* storageUnit)
{
   ODSUForceFocus(ev, storageUnit, kODPropListEx2Data, kListEx2Data);
   ODULong oldSize = storageUnit->GetSize(ev);
   StorageUnitSetValue(storageUnit, ev, TABSIZE, gBigTab);
   ODULong newSize = storageUnit->GetOffset(ev);
   if (newSize < oldSize)
      storageUnit->DeleteValue(ev, oldSize - newSize);
}

void ListEx2Part::InternalizeListData(Environment* ev, ODStorageUnit* storageUnit)
{
   long  theSize;
   if (ODSUExistsThenFocus(ev, storageUnit, kODPropListEx2Data, kListEx2Data))
      if ((theSize = storageUnit->GetSize(ev)) != TABSIZE)
         DebugStr("\pStorage size for gBigTab is wrong !");
      else StorageUnitGetValue(storageUnit, ev, TABSIZE, gBigTab);
}

void ListEx2Part::SetUpListData(Environment* ev, ODStorageUnit* storageUnit)
{
   if (!storageUnit->Exists(ev, kODPropListEx2Data, kODNULL, 0))
      storageUnit->AddProperty(ev, kODPropListEx2Data);
   if (!storageUnit->Exists(ev, kOPPropListEx2Data, kListEx2Data, 0)) {
      storageUnit->Focus(ev, kODPropListEx2Data, kODPosUndefined, kODNULL, 0, kODPosAll);
      storageUnit->AddValue(ev, kListEx2Data);
   }
}
```

LISTEX3PART

In the previous example, we saw one way to manage data — the way that PartMaker creates for us. But we can also manage data directly in the SOM object. That's what happens in ListEx3Part, which generates the list shown in Figure 6.



```
001 ClassName = AppleTest::Container
002 Number of methods = 166
003 MethodDescriptor = SOMObject::somInit
004 MethodDescriptor = SOMObject::somUninit
005 MethodDescriptor = SOMObject::somDuplica
006 MethodDescriptor = SOMObject::somCompar
007 MethodDescriptor = SOMObject::somReleas
008 MethodDescriptor = SOMObject::somFree
009 MethodDescriptor = SOMObject::somCanDel
```

Figure 6. The list generated by ListEx3Part

Let’s back up a minute to see how the SOM field fPart2 is managed in som_ListEx2Part and how the SOM field fPart in som_ListPart is managed by PartMaker. We see that fPart is initialized to NULL in somInit, deleted in somUninit, and allocated in both InitPart and InitPartFromStorage. Because those last two methods aren’t overridden in som_ListEx2Part, fPart2 is allocated in somInit and deleted in somUninit.

As shown in Listing 8, som_ListEx3Part needs three fields:

- one that’s just a pointer and that will be initialized to NULL in somInit
- a second that’s an array of string pointers (see Listing 9)
- a third that’s a big block to store strings

Listing 8. som_ListEx3Part.idl

```
module ACF_DevServ4
{
   interface som_ListEx3Part : som_ListPart
   {
#ifdef __SOMIDL__
      implementation
      {
      ...
      override:
         somInit, somUninit,
         GetTheRealPartKind, OverrideBeginUsingLibraryResources,
         OverrideEndUsingLibraryResources, FacetAdded, InitializeListData,
         SetUpGraphics, FillCell;

         ODPart   gContainingPart;
         char**   gListArray;
         char*    charArray;
      };
#endif
   };
};    //# Module ACF_DevServ4
```

Through the .xih source file, we get the following definitions:

```
#define _gContainingPart (somThis->gContainingPart)
#define _gListArray (somThis->gListArray)
#define _charArray (somThis->charArray)
```

To use these fields, for instance in FillCell, we just add the line that gets somThis:

```
SOM_Scope void SOMLINK som_ListEx3Part__FillCell(
   ACF_DevServ4_som_ListEx3Part *somSelf, Environment *ev, short theLine,
   Rect* theRect)
{
   ACF_DevServ4_som_ListEx3PartData *somThis =
      ACF_DevServ4_som_ListEx3PartGetData(somSelf);
   FillCell(ev, theLine, theRect, _gListArray);
}
```

Listing 9. somInit and somUninit in som_ListEx3Part.cpp

```
SOM_Scope void SOMLINK som_ListEx3Part__somInit(ACF_DevServ4_som_ListEx3Part *somSelf)
{
   ACF_DevServ4_som_ListEx3PartData *somThis = ACF_DevServ4_som_ListEx3PartGetData(somSelf);
   ACF_DevServ4_som_ListEx3PartMethodDebug("ACF_DevServ4_som_ListEx3Part",
      "som_ListEx3Part__somInit");
   ACF_DevServ4_som_ListEx3Part_parent_ACF_DevServ_som_ListPart_somInit(somSelf);
   _gListArray = (char **)NewPtr(NBLINES * sizeof(char *));
   _charArray = (char*)NewPtr(50000);
   _gContainingPart = 0L;
}

SOM_Scope void SOMLINK som_ListEx3Part__somUninit(ACF_DevServ4_som_ListEx3Part *somSelf)
{
   ACF_DevServ4_som_ListEx3PartData *somThis = ACF_DevServ4_som_ListEx3PartGetData(somSelf);
   ACF_DevServ4_som_ListEx3PartMethodDebug("ACF_DevServ4_som_ListEx3Part",
      "som_ListEx3Part__somUninit");
   DisposePtr((Ptr)_gListArray);
   DisposePtr((Ptr)_charArray);
   ACF_DevServ4_som_ListEx3Part_parent_ACF_DevServ_som_ListPart_somUninit(somSelf);
}
```

Of course, the line that gets somThis should be added only to the methods that really need it. If you look at the complete source code for som_ListEx3Part.cpp, you'll see that many methods don't need it and thus don't have this line. The MPW SOM compiler adds it automatically to all methods, so you have to manually remove it if it's not used. The size of the generated code can be greatly decreased in this way.

Now let's take a good look at FacetAdded. It's implemented in som_ListEx3Part.cpp like this:

```
SOM_Scope void SOMLINK som_ListEx3Part__FacetAdded(
   ACF_DevServ4_som_ListEx3Part *somSelf, Environment *ev, ODFacet* facet)
{
   ACF_DevServ4_som_ListEx3PartData *somThis =
      ACF_DevServ4_som_ListEx3PartGetData(somSelf);
   ACF_DevServ4_som_ListEx3PartMethodDebug("ACF_DevServ4_som_ListEx3Part",
      "som_ListEx3Part__FacetAdded");

   FacetAdded(ev, facet, &(_gContainingPart), _gListArray, _charArray);
   ACF_DevServ4_som_ListEx3Part_parent_ACF_DevServ_som_ListPart_FacetAdded
      (somSelf, ev, facet);
}
```

Thus, we can still get the normal behavior for FacetAdded that's contained in ListPart and have a chance to add the specialized behavior that we want for ListEx3Part.

A NOTE ON USING GLOBALS

Let's not forget that if OpenDoc is based on SOM, SOM is based on the Code Fragment Manager (CFM), and this greatly simplifies such programming aspects as management of globals. Indeed, with the CFM architecture, there's no more need for

SetUpA5, SetCurrentA5, SetA5 (or even SetUpA4, provided by some environments); when you need a global, you declare a global, then you use the global, period. When we're building a part, we're in fact building a CFM shared library, but that doesn't prevent us from declaring and using the string globals found in ListEx3Part.cpp, for example. The only trick we've got to pay attention to is this: since OpenDoc loads the library fragment only once when the first part is instantiated, with the kLoadLib flag set and not the kLoadNewCopy flag, globals declared in the library will be shared by all instances of the class in that process.

CLOSING REMARKS

I hope you now have a better understanding of the workings of OpenDoc, SOM, and PartMaker. Dynamic inheritance is a powerful tool. You can easily construct your own useful base parts to be inherited from by yourself and by others. The advantages are that you won't suffer from code duplication, you'll get the benefits of object-oriented programming, and you won't need to rebuild inherited parts when modifying the base part.

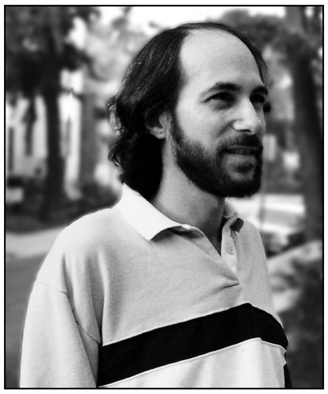
When I first wrote ListPart, I put it in a container document, to which I subsequently added ListEx1Part, then another simple part, and then ListEx2Part. In the course of writing ListEx2Part I discovered that I didn't design my base part as well as I first thought. To correctly implement live editing in place, I had to thoroughly modify ListPart, adding methods, deleting methods, changing method names, changing method parameters, and so on. All the way through my testing, ListEx1Part and the other simple part kept on working in the document without having to be rebuilt.

As long as you don't change the methods used by the inherited parts (in my case, only InitializeListData and FillCell), you're safe. This is because SOM, through the .idl file, completely separates the interface from the implementation of the methods. Suppose I distribute the current version of ListPart for developers to inherit from, and then later I provide a new version of ListPart. As long as I don't modify the methods contained in the current .idl file, I can add new methods and fields to the .idl file and modify the C++ class without anybody being the wiser. All inherited parts developed by others will continue to work fine and will benefit automatically from the new features.

In fact, I expect to provide progressively more refined versions of ListPart to be included on the OpenDoc Developer Release CDs. I plan, for instance, to implement drag and drop, copy and paste, dynamic links, display of information from the container (more useful), and hierarchical lists (the kind with a triangle symbol pointing to the next level).

You get the idea. Why not give SOM dynamic inheritance a try yourself? Then spread the word that OpenDoc isn't just for desktop publishing.

Thanks to our technical reviewers Jens Alfke, Erik Eidt, and Kurt Rodarmer.•



CAL SIMONE

ACCORDING TO SCRIPT

Attaching and Embedding Scripts

One of the least-implemented powerful capabilities you can add to your application is attaching and embedding scripts. In this column I'll give you an idea of how to do this, and clear up some confusion along the way.

ATTACHING VS. EMBEDDING

The term *attach* has been used to refer to both attaching and embedding. Allow me to set the record straight by offering definitions of the two terms as they apply to scripting.

- An *attached* script is a compiled script or script application that's associated with a menu item in an application; the script is executed when the user chooses that command. This type of script usually resides in a particular place, such as a Scripts folder. Script attachment can be implemented quickly and, at its most basic level, doesn't require your application to be scriptable.
- An *embedded* script is a compiled script that's associated with an interface element belonging to an application or with a document. The script can be stored with the application's data, often in a special file known to the application, or embedded within the data for a document file.

ATTACHING SCRIPTS TO MENU ITEMS

Attached scripts are useful for two reasons. You, or your users (depending on what's appropriate for your application), can do the following:

- Execute scripts to communicate with and control other scriptable applications without leaving your application. This is useful whether or not your application is scriptable.
- Use scripts as an means of extending the functions or options available in your application. If your application itself is scriptable, script attachment leverages off the work you've already done.

By allowing users to keep a menu of their favorite scripts, you enable them to build a library of expanded functionality for your application. The Mac OS and Finder accomplish this with the Automated Tasks submenu in the Apple menu. You can do this with a Scripts menu that appears as the last (or next to last) of your application's menus.

Here are the steps for implementing this attachable behavior:

1. In the resource file included with your application, include a menu resource with the title "Scripts."
2. In the startup code for your application, locate the Scripts folder in your application's folder, creating it if it isn't there.
3. Walk the files in the Scripts folder, checking for compiled scripts (file type 'osas') and script applications (file type 'APPL', creator 'aplt' or 'dplt'). Add the names of these files to your Scripts menu.
4. When a user selects a script name from the Scripts menu, load the script resource ('scpt' 128) and execute the script, as shown in Listing 1.

Before executing a script, you must establish a connection to a scripting component. The easiest thing to do is to connect to the generic scripting component with OpenDefaultComponent. When you're done, disconnect from the component with CloseComponent. Depending on how you design your application, you can open this connection and keep it open while your program is running, or you can open and close the connection each time you load and execute a script. For more information on choosing and connecting scripting components, see *Inside Macintosh: Interapplication Communication*, Chapter 10.

EMBEDDED SCRIPTS IN APPLICATION DATA OR DOCUMENT FILES

Embedded scripts can be used in two ways:

- Interface elements belonging to an application, such as tool palette icons, menu items, and buttons, can have scripts associated with them.

CAL SIMONE (mainevent@his.com, AppleLink MAIN.EVENT) Few people know it, but before Cal was in the software business, he used to produce records (the musical kind) in Washington DC and New York. At a time when computers were used mostly to make robotic dance music, Cal was one of the first to painstakingly create "human" performances in pop records with about 60 MIDI synthesizers and, of course, a Macintosh. He now works toward a day when every application will be scriptable.•

- Scripts can be associated with individual documents. Unlike the above case, you can trigger the script with any method that's appropriate for your application.

Embedding scripts can be extremely powerful. For example, you can associate scripts with elements of a form to supply a field's editing rules, or with a button to perform calculations. Replace a script and you change the rules or the formula! Depending on your particular application, you can use this technique yourself or allow users to do their own replacement.

If you reserve this technique for your own use, you can revise your software simply by replacing scripts with corrected or enhanced versions. Or, if you allow your

Listing 1. Loading and executing a script from a file

```
FUNCTION RunAttachedScript(theAlias: AliasHandle): OSAError;
VAR
   fileSpec:           FSSpec;
   scriptRes:          Handle;
   scriptDesc:         AEDesc;
   scriptID, resultID: OSAID;
   myErr, ignoredErr:  OSAError;
   savedRes, refNum:   Integer;
   specChanged:        Boolean;

BEGIN
   (* Get the file specification corresponding to the menu item chosen. *)
   myErr := ResolveAlias(NIL, theAlias, fileSpec, specChanged);
   IF myErr <> noErr THEN MyErrorProc(myErr);

   (* Open the resource fork and grab the script resource. *)
   savedRes := CurResFile;
   refNum := FSpOpenResFile(fileSpec, fsRdPerm);
   IF refNum = -1 THEN MyErrorProc(-1);
   UseResFile(refNum);
   scriptRes := Get1Resource(kOSAScriptResourceType, 128);
   IF ResError <> noErr THEN MyErrorProc(ResError);

   (* Prepare and run the script. *)
   myErr := AECreateDesc(typeOSAGenericStorage, scriptRes^, GetHandleSize(scriptRes),
      scriptDesc);
   IF myErr <> noErr THEN MyErrorProc(myErr);
   myErr := OSALoad(gGenericComponent, scriptDesc, kOSAModeNull, scriptID);
   IF myErr <> noErr THEN MyErrorProc(myErr);
   myErr := OSAExecute(gGenericComponent, scriptID, kOSANullScript, kOSAModeNull, resultID);
   ignoredErr := OSADispose(gGenericComponent, scriptID);
   ignoredErr := AEDisposeDesc(scriptDesc);
   IF myErr <> noErr THEN MyErrorProc(myErr);

   (* Finish up. *)
   ReleaseResource(scriptRes);
   CloseResFile(refNum);
   UseResFile(savedRes);

   (* You might want to do something with the result. *)
   IF resultID <> kOSANullScript THEN MyDealWithResult(resultID);
   RunAttachedScript := myErr;
END;
```

users to change the embedded scripts, your application becomes easily customizable: users can modify or augment your application's capabilities simply by substituting scripts. You could even ship your application with replacement scripts, which users can substitute for default scripts that you provide.

RETRIEVING EMBEDDED SCRIPTS

There are three methods of retrieving embedded scripts from files, depending on where they're stored. Regardless of which method you choose, it's important to remember that your program should never try to interpret the bytes of a compiled script. However, as long as you keep the bytes intact, you can do whatever you want with them and the script will remain intact.

Aliases to script files. This is the same technique as described above for attached scripts. This method is used primarily for maintaining a list of scripts. You'd use it, for instance, if you kept a collection of scripts in a folder on disk. I don't recommend this technique if the scripts are associated with actual interface elements, because the links that aliases provide to the script files can too easily be broken.

In the document's resource fork. Storing the scripts as resources is convenient because you can easily use your favorite resource editor to copy a script resource from a compiled script or script application and paste it into the special application file or the document. It also makes it easy to grab the scripts for loading and executing, using the method shown in Listing 1 (though in this situation I'd suggest using an ID number other than 128 for the script resource). The drawback is that your users can get their hands on the script with *their* favorite resource editor.

In the document's data fork. Maintaining the scripts within the data for a document is a more secure method, since it makes it harder for users to extract the scripts. It's also more difficult for you, though, because you may have to keep track of the location within the document's data, and then convert the script into the form required for execution. You'll want to store three pieces of information: the four-character ID 'scpt' (typeOSAGenericStorage), the length of the script data that follows, and the script data itself. The ID isn't essential, but it may come in handy, especially if there are other types of data present or if you load your document's data sequentially.

There are many ways to keep track of multiple types of data in a document file. If you have a lot of different types of data in the file, you can even develop a small database for the data, complete with a directory, so that you can gain quick access to particular types of data, including the script. A simpler way is to maintain the data in one long stream, embedding the script data within the stream. If you know the location of the script within the stream, you can just load and execute it when a user wants to run it. One developer I know reads all the data in the data fork (including scripts) sequentially when the file is opened, so that he doesn't need to keep track of the script's location within the file. Listing 2 shows an example of loading script data from the data fork of a document file.

Listing 2. Extracting script data from a document's data fork

```
FUNCTION RunEmbeddedScriptFromDataFork(theAlias: AliasHandle; scriptLoc: LongInt): OSAError;
VAR
   fileSpec:           FSSpec;
   scriptData:         Handle;
   scriptDesc:         AEDesc;
   dataType:           DescType;
   scriptID, resultID: OSAID;
   myErr, ignoredErr:  OSAError;
   refNum:             Integer;
   scriptLen, readLen: LongInt;
   specChanged:        Boolean;

BEGIN
   (* Open the file. *)
   myErr := ResolveAlias(NIL, theAlias, fileSpec, specChanged);
   IF myErr <> noErr THEN MyErrorProc(myErr);
```

(continued on next page)

Listing 2. Extracting script data from a document's data fork *(continued)*

```
   myErr := FSpOpenDF(fileSpec, fsRdPerm, refNum);
   IF myErr <> noErr THEN MyErrorProc(myErr);

   (* Grab the data. *)
   IF MemError <> noErr THEN MyErrorProc(MemError);
   myErr := SetFPos(refNum, fsFromStart, scriptLoc);
   readLen := sizeof(dataType);
   IF myErr = noErr THEN myErr := FSRead(refNum, readLen, @dataType);
   (* dataType should be typeOSAGenericStorage. *)
   readLen := sizeof(scriptLen);
   IF myErr = noErr THEN myErr := FSRead(refNum, readLen, @scriptLen);
   IF myErr = noErr THEN scriptData := NewHandle(scriptLen);
   IF MemError <> noErr THEN MyErrorProc(MemError);
   myErr := FSRead(refNum, scriptLen, scriptData^);
   IF myErr <> noErr THEN MyErrorProc(myErr);
   myErr := FSClose(refNum);

   (* Prepare and run the script. *)
   myErr := AECreateDesc(typeOSAGenericStorage, scriptData^, GetHandleSize(scriptData),
      scriptDesc);
   DisposeHandle(scriptData);
   IF myErr <> noErr THEN MyErrorProc(myErr);
   myErr := OSALoad(gGenericComponent, scriptDesc, kOSAModeNull, scriptID);
   IF myErr <> noErr THEN MyErrorProc(myErr);
   myErr := OSAExecute(gGenericComponent, scriptID, kOSANullScript, kOSAModeNull, resultID);
   ignoredErr := OSADispose(gGenericComponent, scriptID);
   ignoredErr := AEDisposeDesc(scriptDesc);
   IF myErr <> noErr THEN MyErrorProc(myErr);

   (* You might want to do something with the result. *)
   IF resultID <> kOSANullScript THEN MyDealWithResult(resultID);
   RunEmbeddedScriptFromDataFork := myErr;
END;
```

GIVING IT AWAY

The information in this column is not offered as a complete solution, but is intended to get you moving with implementing attachability. There are many other issues surrounding attachability that are worth exploring, such as getting time during script execution, using attached scripts to allow users to tinker with some of the core functionality of your application, and providing a consistent way for your users to edit attached and embedded scripts. I plan to delve into these other issues in upcoming columns.

Making your application capable of attaching or embedding scripts puts new power into your users' hands, giving them unprecedented ability to develop custom solutions to their problems. It's not hard to do, and the benefits are enormous. Do it today.

Thanks to Sue Dumont, Greg Friedman, Eric Gundrum, Wayne Malkin, Jon Pugh, and Derrick Schneider for reviewing this column.•

Adding Custom Data to QuickDraw 3D Objects

Custom attributes and elements provide a way to attach data such as scaling information, sound, and strings to QuickDraw 3D objects. In this article we explain how to create and attach custom attributes and elements. We illustrate the process by showing you how to attach a string containing a World Wide Web URL to a QuickDraw 3D object to enable 3D navigation through the Web. We also describe six new custom elements with implementations included on this issue's CD.

NICK THOMPSON, PABLO FERNICOLA, AND KENT DAVIDSON

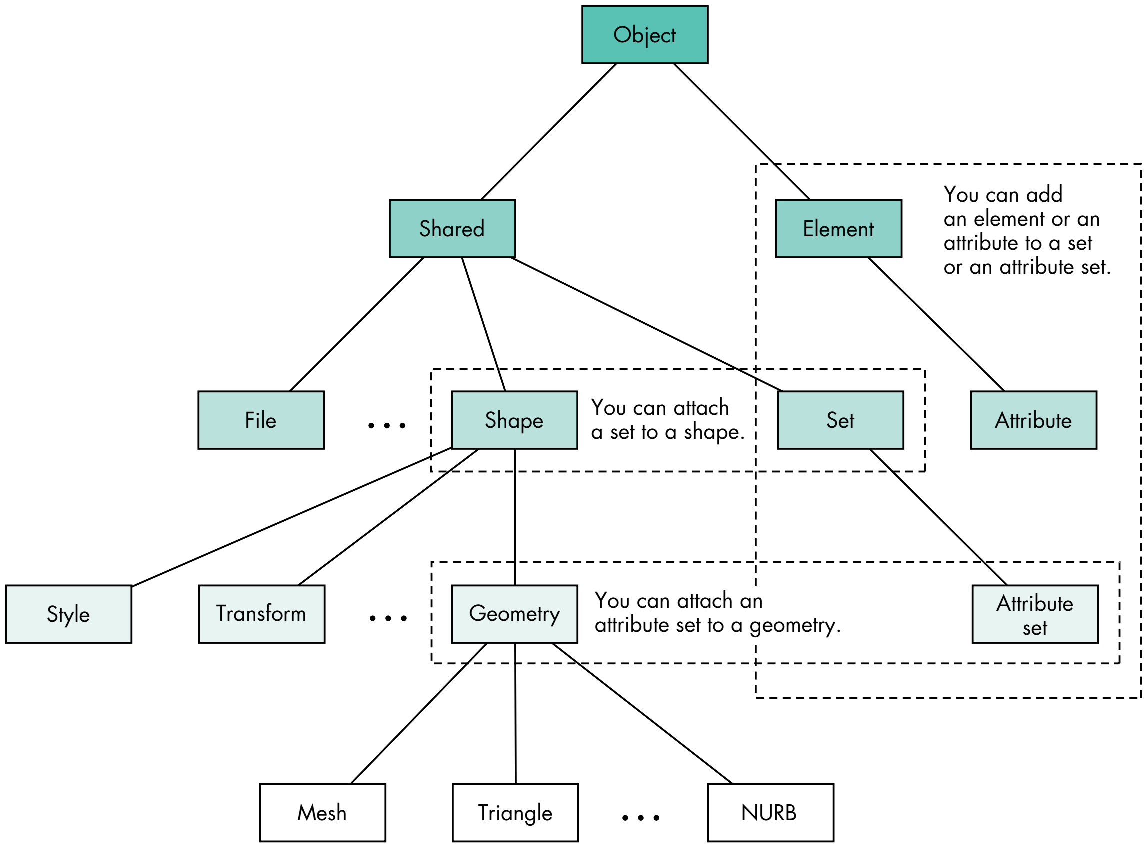
In QuickDraw 3D, attribute objects (known more simply as attributes) generally store information about the surface properties of objects in a model, such as color and transparency. QuickDraw 3D defines 12 basic attribute types, and it also allows you to define custom attribute and element types so that you can attach data different from the predefined types to QuickDraw 3D objects. Your custom data need not apply to the appearance of objects or to how objects are drawn, although it can.

For example, with custom attributes and elements you can add scaling information, directional information, or sound to objects in your 3D scene. You can add a string containing a name for a QuickDraw 3D object, so that you can refer to that object by name and control it from a scripting language in your application. Your application can enable users to navigate through the World Wide Web in 3D, by attaching a URL to a QuickDraw 3D object, as illustrated by the code discussed in this article and included on this issue's CD. These are just a few of the ways you can extend the functionality of QuickDraw 3D and add value to your 3D application by adding custom data to objects.

Before we explain and illustrate how to create custom attributes and elements, and how to attach them to QuickDraw 3D objects, we'll look at how attributes and elements relate to each other and to other QuickDraw 3D objects. To get the most from this article, you should already be familiar with the basics of QuickDraw 3D, as presented in the previous *develop* articles "QuickDraw 3D: A New Dimension for Macintosh Graphics" (Issue 22) and "The Basics of QuickDraw 3D Geometries" (Issue 23). The

NICK THOMPSON (nickt@applelink.apple.com, AppleLink NICKT) is still looking for new ways to make the rest of the QuickDraw 3D team members think about Mac OS 8. Nick is working on integrating the award-winning QuickDraw 3D software into Mac OS 8, taking maximum advantage of the modern OS features. When not immersed in the future of QuickDraw 3D, this former Developer Technical Support engineer is immersed in water, surfing off the Northern California coast.•

PABLO FERNICOLA (EscherDude@aol.com, AppleLink PFF) has been really busy since you last heard from him. At MACWORLD San Francisco in January he was busy explaining to developers and users what QuickDraw 3D means to them. As we write this, he's busy planning the next three releases of QuickDraw 3D with the team. As technical lead of the QuickDraw 3D team, he misses life without meetings, but is totally stoked that the team has won awards from *Byte*, *Macworld*, and *MacUser*.•



Object
Shared
Element
You can add an element or an attribute to a set or an attribute set.
File
. . .
Shape
You can attach a set to a shape.
Set
Attribute
Style
Transform
. . .
Geometry
You can attach an attribute set to a geometry.
Attribute set
Mesh
Triangle
. . .
NURB

Figure 1. Partial QuickDraw 3D class hierarchy, showing set and attribute set attachment

book *3D Graphics Programming With QuickDraw 3D*, included on this issue's CD, provides complete documentation for the QuickDraw 3D programming interfaces.

ABOUT ATTRIBUTES AND ELEMENTS

Attributes and elements are types of QuickDraw 3D objects used to store information about objects they're attached to. Each consists of a type and some associated data. You apply attributes and elements to objects by creating an instance of a specific type of attribute or element, defining its data, adding it to a set, and then attaching the set to an object (if the set isn't already attached).

Note that attributes and elements are *attached* to objects, as opposed to simply being added to a group. The reason for binding data to objects is that both QuickDraw 3D and the 3DMF format maintain a strong data encapsulation model. For example, this allows QuickDraw 3D objects to be moved from file to file without losing data.

ATTRIBUTES AND ELEMENTS IN THE QUICKDRAW 3D HIERARCHY

To better understand how attributes and elements relate to each other and to other QuickDraw 3D objects, take a look at the partial class hierarchy shown in Figure 1.

KENT DAVIDSON (dbunny@apple.com, AppleLink DBUNNY), a.k.a. 3DMF Dude, Object Dude, and "The Man" (by the dudes in Marketing), is the guy who keeps the core of QuickDraw 3D humming. Outside of commuting from San Francisco to the 'burbs of Cupertino, he spends his time rock climbing, skiing, and hanging out. He's currently wracking his brain over the plug-in renderer architecture, which he'll finish as soon as everyone leaves him alone.•

As you can see, an attribute is actually a type of element (that is, it's a subclass of the Element class, TQ3ElementObject). An element is any QuickDraw 3D object that can be part of a set. In contrast with shared objects (objects of the class TQ3SharedObject), elements aren't shared (that is, they can't be referenced by multiple objects or the application at the same time) and are always removed from memory whenever they're disposed of. An attribute has all of these properties but also can be inherited by subclasses of the object it's attached to.

Custom data to be attached to an object can be stored in an element or an attribute. So how do you decide which to use? Use an attribute when you want your custom data to be inherited. For example, suppose we create a custom attribute named Temperature and we want to be able to assign a different temperature to an entire geometry, a face, or a vertex. During a view traversal loop, our attribute will be inherited along with the other attributes. This becomes extremely important with the introduction of plug-in renderers, which will be available in a future QuickDraw 3D release. A particular renderer might take advantage of this inherited attribute by coloring each vertex according to the temperature inherited.

SETS AND ATTRIBUTE SETS

We mentioned earlier that attributes and elements are usually collected in sets. A *set* is an instance of the Set class (TQ3SetObject), which in turn is a subclass of the Shared class (TQ3SharedObject), as shown in Figure 1. A set collects zero or more different elements or attributes and their associated data; it can contain only one element or attribute of a given type. An *attribute set* is a type of set; in fact, TQ3AttributeSet is the only subclass of the class TQ3SetObject. An attribute set has all the properties of a set but also allows inheritance.

Both elements and attributes can be collected in sets and attribute sets. Since the AttributeSet class is derived from the Set class, you can call Q3Set_XX on an attribute set, but you can't call Q3AttributeSet_XX on a set. In the text that follows, be sure to pay attention to whether we're talking about sets or attribute sets; we don't use the terms interchangeably.

Sets and attribute sets can't be attached to just any QuickDraw 3D object, but only to those objects for which it makes sense to store additional data in this way. Attribute sets can be attached to view objects, group objects, and geometric objects, plus most of the parts of a geometric object: faces, vertexes, mesh edges, and mesh corners. (See "How to Attach Attribute Sets" for details.) In contrast, sets can be attached only to objects in the Shape class or subclasses of the Shape class. (Attaching a set to a shape is fairly straightforward; we give an example of how to do this later, in Listing 3.) The Shape class actually has a class field of type **set**, meaning that any class derived from Shape has a set object. The Geometry class has a class field of type **set** (inherited from the Shape class) *plus* a class field of type **attributeSet**, meaning that any class derived from Geometry has both a set object and an attribute set object.

Currently, the renderers shipped with QuickDraw 3D ignore custom data attached to shape objects, but when plug-in renderers become available, they may pay attention to such data and use it to control certain rendering features. For example, a ray tracer renderer may need custom data about surfaces to render them with bump mapping.

ATTRIBUTES AND INHERITANCE

When the objects in a view are rendered, attributes attached to the objects are applied according to a strict hierarchy. The attribute sets of objects higher in the view hierarchy are inherited by objects below them, unless some other attribute set overrides them. Inheritance proceeds from view to group to geometric object to face

HOW TO ATTACH ATTRIBUTE SETS

It may be news to you that attribute sets can be attached to views or groups, because how to do this is less than obvious. We'll tell you how.

To attach an attribute set to a view object, use the Q3View_GetDefaultAttributeSet routine to get the default attribute set (all view objects have one), and then use Q3AttributeSet_Add to add attributes to that set. For example, the following code shows how to apply a default specular color to all objects submitted to a view.

```
Q3View_GetDefaultAttributeSet(theDocument
   ->theView, &viewSet);
Q3AttributeSet_Add(viewSet,
   kQ3AttributeTypeSpecularColor, &clearColor);
Q3Object_Dispose(viewSet);
```

You can still override the default behavior of the view by attaching attributes to objects before submitting them. If you write the view hints out in 3DMF format using the QuickDraw 3D API, the attribute set for the view will also be written out. You can preserve these settings by looking in and using the view hints when you read the 3DMF data back in.

To attach an attribute set to a group object, just add the attribute set to the group before you add the object you want it to be applied to.

Attaching an attribute set to a geometric object or a part of a geometric object is much more obvious, so we won't go into details here. Later in this article, Listing 2 gives an example of how to do it.

to mesh edge to vertex to mesh corner. In other words, in the hierarchy, view attributes are always inherited unless a group contains overriding attributes; group attributes can be overridden by geometric object attributes, which can be overridden by face attributes, and so on.

When you define a custom attribute, you can specify that you want it to be inherited by including an attribute inheritance method in your metahandler. (More on metahandlers later.) Inheritance happens when you call Q3AttributeSet_Inherit:

```
TQ3Status Q3AttributeSet_Inherit(TQ3AttributeSet parent,
                              TQ3AttributeSet child, TQ3AttributeSet result);
```

This call takes three attribute sets: the parent, the child, and a result attribute set to store results in, which becomes the effective attribute set after inheritance. During inheritance, any attribute in the parent that's not in the child is copied into the result, and all child attributes are copied into the result, as illustrated by the example in Figure 2. As mentioned earlier, only attributes can be inherited; elements, such as the name element "Jane" in this example, can exist in an attribute set but aren't inherited.

WORKING WITH CUSTOM ATTRIBUTES AND ELEMENTS

Now that you have a sense of how custom attributes and elements relate to each other and to other QuickDraw 3D objects, we'll outline how you define, register, and attach your custom data to the QuickDraw 3D objects of your choice. We'll further illustrate the process later in our example of attaching a URL to a QuickDraw 3D object.

DEFINING AND REGISTERING YOUR CUSTOM DATA

To define a custom attribute or element type, you need to provide a definition of the data associated with that type and write a metahandler to define a set of attribute- or element-handling methods. Once you've defined and registered your custom attribute or element type, you manipulate objects of that type exactly as you manipulate the standard QuickDraw 3D attributes. For example, you create a new attribute set by calling Q3AttributeSet_New, and you add custom attributes to the

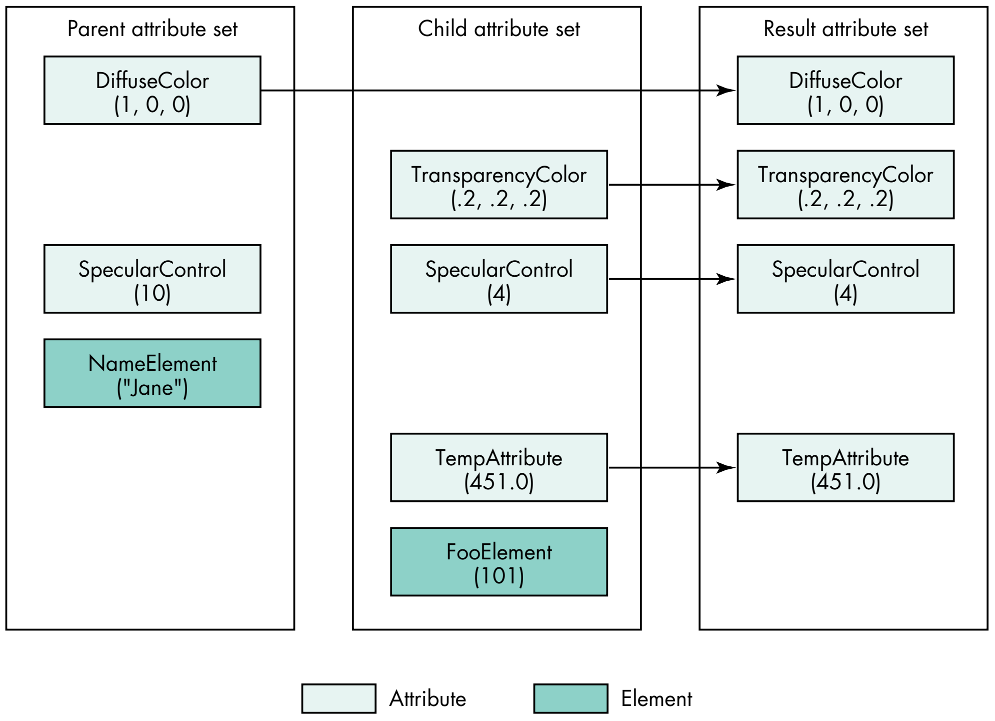


Figure 2. Attribute inheritance

attribute set by calling Q3AttributeSet_Add. Finally, you attach the attribute set to an object by calling an appropriate QuickDraw 3D routine.

Before you can use your custom element or attribute, you must register it with QuickDraw 3D by calling Q3ElementClass_Register or Q3AttributeClass_Register:

```
TQ3ObjectClass Q3ElementClass_Register(TQ3ElementType elementType,
   const char *name, unsigned long sizeOfElement, TQ3MetaHandler metaHandler);

TQ3ObjectClass Q3AttributeClass_Register(TQ3AttributeType  attributeType,
   const char *name, unsigned long sizeOfElement, TQ3MetaHandler metaHandler);
```

The functions take these parameters:

- elementType (or attributeType) — The type constant used in the binary metafile and in accessing your element (or attribute) from a set.
- name — The string constant used to write your custom element or attribute in a text metafile. You should register your attribute or element types and names with Apple's Developer Support Center to prevent name space collisions. In general, you should name your custom elements and attributes in the form "Company:DataType"; for instance, if you work at Sun, you might name an attribute "Sun:JavaCode."
- sizeOfElement — The memory size that your element or attribute uses internally. QuickDraw 3D needs to know this when copying your element or attribute, because the data describing the element or attribute is copied from the public side of the API to internal storage.
- metaHandler — A pointer to the metahandler for your element or attribute.

A metahandler is an application-defined function that returns the addresses of the methods associated with the custom attribute or element type. QuickDraw 3D calls these methods at certain times to handle operations on sets and attribute sets that contain your custom data. Particular methods are required for each QuickDraw 3D object type, and QuickDraw 3D asks the metahandler repeatedly for these required methods. Your metahandler should, by default, return NULL for unrecognized methods; this allows Apple to add methods in the future without breaking the implementation of old versions of elements and attributes.

A metahandler can define some or all of the methods indicated by the constants listed below. Custom elements or attributes that are to be read from and written to files should support the I/O methods associated with objects (those methods beginning with "kQ3MethodTypeObject" in the following list). The metahandler can also support all the methods associated with elements (those methods beginning with "kQ3MethodTypeElement" in the list) and attributes (those methods beginning with "kQ3MethodTypeAttribute"). Note that the copy methods always take the source as the first parameter (**from**) and the destination as the second parameter (**to**), although what these point to differs for each copy method. All of the following method types are optional. If you supply no method for a particular attribute or element type, your attribute or element will inherit the default behavior of the parent class.

- kQ3MethodTypeObjectReadData — Reads the data from a file object, gathers any subobjects, and adds the element to a set.
- kQ3MethodTypeObjectTraverse — Calculates the size of the data to be written out, submits any subobjects, and gathers any state needed from the view object.
- kQ3MethodTypeObjectWrite — Actually writes the data to the file. Data is written through one of the low-level calls provided by QuickDraw 3D for basic data types. If your data size is always 0, no ObjectWrite method is required.
- kQ3MethodTypeElementCopyAdd — Called when an application calls Q3Set_Add or Q3AttributeSet_Add on your element and the element wasn't in the set. The **from** parameter is whatever the user passes in as the data pointer in Q3Set_Add. The **to** parameter is a pointer to an uninitialized block of sizeOfElement (from the Register call) bytes. If this method isn't supplied, the default is to copy sizeOfElement bytes from the source to the destination.
- kQ3MethodTypeElementCopyReplace — Called when an application calls Q3Set_Add or Q3AttributeSet_Add on your element and the element already exists in the set. The **from** parameter is whatever the user passes in as the data pointer in Q3Set_Add. The **to** parameter is a pointer to a block of sizeOfElement bytes that contains the element data to be replaced. You must reuse or delete any data in the destination before copying over it. If this method isn't supplied, the default is to call ElementDelete on the **to** parameter, then CopyAdd(from, to).
- kQ3MethodTypeElementCopyGet — Called when an application calls Q3Set_Get or Q3AttributeSet_Get on your element. The **from** parameter is a pointer to the block of element data to get. The **to** parameter is a pointer to whatever the user passes in as the data pointer in Q3Set_Get. If this method isn't supplied, the default is to copy sizeOfElement bytes from the source to the destination.
- kQ3MethodTypeElementCopyDuplicate — Called when an application calls Q3Object_Duplicate on a set or attribute set, or duplicates an object containing a set. The **from** parameter is a pointer to the block of element data to duplicate. The **to** parameter is a pointer to an uninitialized block of sizeOfElement bytes. If your element contains objects, call Q3Object_Duplicate to create an identical copy. If this method isn't supplied, the default is to copy sizeOfElement bytes from the source to the destination.
- kQ3MethodTypeElementDelete — Called when an application deletes a set containing your element, or clears your element with Q3Set_Clear or

Q3AttributeSet_Clear. It takes a pointer to the block of element data. It should deallocate any data in your custom element. If this method isn't supplied, the default is a no-op.

- kQ3MethodTypeAttributeInherit — Your metahandler should return a TQ3Boolean value for this method. Returning kQ3True indicates that this attribute should be inherited in the hierarchy, kQ3False that it should not. The default is kQ3False.
- kQ3MethodTypeAttributeCopyInherit — Called when your attribute is inherited in the view stack (during rendering) or when the user calls Q3AttributeSet_Inherit with an attribute set containing your attribute. The **from** parameter is a pointer to the block of attribute data to inherit. The **to** parameter is a pointer to an uninitialized block of sizeOfElement bytes. The semantics of this call are similar to kQ3MethodTypeElementCopyDuplicate, although you should avoid duplicating data unless required. For example, if your attribute contains pointers to shared objects, you should copy them by calling Q3Shared_GetReference instead of Q3Object_Duplicate. If this method isn't supplied, the default is to copy sizeOfElement bytes from the source to the destination. This method should be implemented to be as fast as possible, as it occurs during rendering.

Listing 1 shows a typical metahandler for a custom element in QuickDraw 3D. Take a look at the QuickDraw 3D header file QD3DIO.h to see the object methods and at QD3DSet.h to see the element and attribute methods.

ATTACHING YOUR CUSTOM DATA

Now we'll show you how to add the custom data you've defined to a set or an attribute set and then attach that set or attribute set to an object. Note that when you

Listing 1. A typical metahandler for a custom element

```
TQ3FunctionPointer MyMetaHandler(TQ3MethodType methodType)
{
   switch (methodType) {
      case kQ3MethodTypeObjectTraverse:
         return (TQ3FunctionPointer) MyElementTraverse;
      case kQ3MethodTypeObjectWrite:
         return (TQ3FunctionPointer) MyElementWrite;
      case kQ3MethodTypeObjectReadData:
         return (TQ3FunctionPointer) MyElementReadData;
      case kQ3MethodTypeElementCopyAdd:
         return (TQ3FunctionPointer) MyElementCopyAdd;
      case kQ3MethodTypeElementCopyReplace:
         return (TQ3FunctionPointer) MyElementCopyReplace;
      case kQ3MethodTypeElementCopyGet:
         return (TQ3FunctionPointer) MyElementCopyGet;
      case kQ3MethodTypeElementCopyDuplicate:
         return (TQ3FunctionPointer) MyElementCopyDuplicate;
      case kQ3MethodTypeElementDelete:
         return (TQ3FunctionPointer) MyElementDelete;
      default:
         return (TQ3FunctionPointer) NULL;
   }
}
```

want to attach custom data to a geometric object or some part of a geometric object, you actually have a choice of where to attach the data. You can add the data to an attribute set and attach it to the geometry or some part of the geometry, or you can add the same data to a set and attach that set to a shape, since the geometry inherits from the shape. Where and how data is attached to an object is really up to the semantics of your application. Just be sure to consistently attach data in the same place on all objects, and document what you've done, especially if you want your custom element or attribute to be used by other developers.

To illustrate this concept, Listing 2 creates a new attribute set, adds our custom data to the attribute set, and attaches the attribute set to a mesh vertex. This is a fine way to customize a geometric object or some part of a geometric object. But if you want to add your custom data to some other subclass of the Shape class, you'll want to add the data to a set and attach that set to the shape. Listing 3 does just that.

Listing 2. Attaching an attribute set to a vertex

```
/* Get the existing attribute set (if any). */
Q3Mesh_GetVertexAttributeSet(mesh, someVertex, &theAttrSet);

/* If there's no attribute set we get back NULL and create one. */
if (theAttrSet == NULL) {
   /* Create a new empty attribute set. */
   theAttrSet = Q3AttributeSet_New();
   if (theAttrSet == NULL)
      return kQ3Failure;
   Q3Mesh_SetVertexAttributeSet(mesh, someVertex, theAttrSet);
}

/* Add the custom data to the attribute set. */
if (Q3AttributeSet_Add(theAttrSet, kMyCustomDataType, &myCustomData)
      == kQ3Failure) {
   Q3Object_Dispose(theAttrSet);
   return kQ3Failure;
}
Q3Object_Dispose(theAttrSet);
return kQ3Success;
```

A CASE IN POINT: ATTACHING A URL TO AN OBJECT

Now we're going to illustrate how to define, register, and attach a custom element to a QuickDraw 3D object, and how to extract and use that custom data. Our custom element is a string containing a URL (*uniform resource locator*, a popular way of specifying the location of an online resource on the Web); we'll attach it to a geometry object. We make it an element rather than an attribute because it doesn't need to be inheritable. When the object we attach the custom element to is read into one of the many viewers that support custom elements, the viewer can communicate through Apple events with applications like Netscape Navigator™ (or your favorite Web browser) to produce 3D navigation. A sample application that illustrates the idea is included on this issue's CD. See "3D Web Content Using 3DMF and Netscape Navigator" for more details.

The custom attribute we define and use here, W3Anchor, is one of the six custom elements described later in this article.

Listing 3. Attaching a set to a shape

```
/* Get the existing set (if any). */
Q3Shape_GetSet(shape, &theSet);

/* If there's no set, add one. */
if (theSet == NULL) {
   theSet = Q3Set_New();
   if (theSet == NULL)
      return kQ3Failure;
   Q3Shape_SetSet(shape, theSet);
}

/* Add the custom data to the set. */
if (Q3Set_Add(theSet, kMyCustomDataType, &myCustomData)
      == kQ3Failure) {
   Q3Object_Dispose(theSet);
   return kQ3Failure;
}
Q3Object_Dispose(theSet);
return kQ3Success;
```

DEFINING OUR DATA STRUCTURE

We first need to define the internal structure of the data associated with our custom element type. We'll use the W3AnchorData structure, defined like this:

```
typedef enum W3AnchorOptions {
   kW3AnchorOptionNone     = 0,
   kW3AnchorOptionUseMap   = 1
} W3AnchorOptions;

typedef struct W3AnchorData {
   char              *url;
   TQ3StringObject   description;
   W3AnchorOptions   options;
} W3AnchorData;
```

The **url** field is a C string consisting of the URL data. The **description** object is information that the application must present to users to enable them to decide whether the site or data pointed to by the URL is worth examining (since the process could take some time). Note that since the description is a string object, it can be specified in a script other than Roman. The **options** field specifies whether the position (*x*,*y*) that was clicked should be passed back to the Web viewer.

REGISTERING OUR CUSTOM ELEMENT

Before we can use our custom element, we need to tell QuickDraw 3D that we've defined it, by implementing a registration routine. There may be occasions when we want to recognize a custom element for only a limited period of time, so an unregister routine can also be implemented.

We need to define a couple of parameters before we can register our custom element: an object type, which is a four-character identifier packed into a long word, and a string, which is used to help uniquely identify the element. As mentioned earlier, both

3D WEB CONTENT USING 3DMF AND NETSCAPE NAVIGATOR

BY JOHN LOUCH

With the advent of Netscape Navigator 2.0 and its plug-in architecture, you can extend content on the Web to handle multimedia or 3D media. The ease of publishing 3D content will make Apple's 3DMF data format ubiquitous on the Web. A sample Netscape plug-in, Whurlplug, on this issue's CD shows what a 3D plug-in based on QuickDraw 3D might look like.

Whurlplug uses the QuickDraw 3D Viewer shared library as its interface for displaying 3D Web content. The Viewer gives users of Whurlplug a seamless integration with the current metaphors for handling 3D content on the Mac OS. Whurlplug also tries to use the same human interface metaphors and behaviors as Netscape.

Whurlplug can be embedded in a Hypertext Markup Language (HTML) page or take over the whole window (as in the URL example in this article). If the plug-in is embedded, it will assume the same background color as the HTML page it's embedded in. Holding down the mouse button on the Viewer toolbar will pop up a menu allowing you to set the Viewer options and save the Web-based 3DMF object to disk, so it's consistent with other elements of the Netscape browser's user interface.

There are a number of ways to present a 3D scene to a Web user. You can enable the user to fly through a 3D world, or simply to view an HTML page with 3D content. To handle the different ways that Whurlplug might be used, we extended the HTML syntax that the plug-in understands if it's embedded in a page. Here's the Embed command syntax:

```
<EMBED SRC="3DObject.3dmf" WIDTH=100 HEIGHT=200>
```

Six more arguments for this extension to HTML can be used in a description of a 3DMF object:

- ACTIVE — If this is set to true, the user can examine the 3D object through the controls provided by the QuickDraw 3D Viewer and keyboard navigation. If it's false, the user can interact with the 3D object only if it has URL links to other pages inside it.
- BGCOLOR — Allows the page author to set the background color of the plug-in or model to the color supplied. BGCOLOR="#ffffff" would set the background color to white. The string is defined as a number consisting of six hexadecimal digits, each pair of which describes the red, blue, and green components (in that order).
- SPIN — If this is set to true, the 3D object will spin about a moving axis defined by Whurlpug; otherwise, the object won't spin.
- ROTATE — This also allows the 3D object to spin when viewed, but the page author defines the axis of rotation. The syntax is ROTATE="*x y z*" where *x*, *y*, and *z* are floating-point values from –180.0 to 180.0 defining the axis of rotation.
- TOOLBAR — If this is set to false, the toolbar at the bottom of the viewer isn't shown. The default is true.
- RENDER — Tells the plug-in which renderer to use. RENDER=interactive (the default) indicates the interactive software renderer; RENDER=wireframe indicates the wireframe renderer that ships with QuickDraw 3D.

Whurlplug understands 3D models that have URL or anchor links in them. If the cursor moves over a 3D object that has an anchor link in it, the object flashes red and the URL is displayed in Netscape's toolbar. Clicking on that object causes Netscape to go to that URL, which could be anything from another QuickDraw 3D object to any type of page that Netscape understands. Currently, the only way to add anchors to a QuickDraw 3D object is through the applications BeWhurled (on this issue's CD), 3D World, and Studio Pro Blitz, but the URL example in this article shows how you can add the anchor custom attribute to data in your own 3D application.

The mime type and subtype for 3DMF are x-world/x-3dmf. The extensions that Whurplug understands are .3dmf, .3dm, .qd3d, and .qd3. Your Web server has to either set the mime type and subtype of 3D files to x-world/x-3dmf or name the files so that the extension is one of those Whurlplug understands.

Following is a trivially simple HTML description of a Web page that uses this viewer. By the time you read this, there will be (we hope) a number of sites with 3DMF data on their Web pages that can be viewed in Netscape. Check out the QuickDraw 3D Web page for more details.

```
<TITLE> A 3D Web page <\TITLE>
<EMBED SRC="3DObject.3dmf" WIDTH=200 HEIGHT=200
   SPIN=true ACTIVE=false>
<P>
<A HREF="3DObject.3dmf">Click here for a full
   view</A>
```

of these need to be registered with the Developer Support Center to avoid name space collisions, and each must be unique within their respective name spaces.

```
#define kElementTypeW3Anchor \
   ((TQ3ElementType) Q3_OBJECT_TYPE('w','w','w','a'))
#define kElementNameW3Anchor "W3Anchor"
```

Now we register the custom element:

```
TQ3Status W3Anchor_Register(void)
{
   gW3AnchorClass = Q3ElementClass_Register(kElementTypeW3Anchor,
      kElementNameW3Anchor, sizeof(W3AnchorData), W3Anchor_MetaHandler);
   return (gW3AnchorClass == NULL ? kQ3Failure : kQ3Success);
}
```

When you register custom attributes or elements with Q3ElementClass_Register, the name you use doesn't have to be the exact same name used by other developers for that type. As an example, the W3Anchor type is defined as 'wwwa' and its name is "W3Anchor." An Apple implementation of this attribute might be registered as Q3ElementClass_Register('wwwa', "Apple:W3Anchor"), and a third party's implementation might be registered as Q3ElementClass_Register('wwwa', "Microspot:W3Anchor"). The name is unimportant; because both of the implementations have the same type, data written by one will, if the implementation of both is the same, be read by the other.

DEFINING OUR METAHANDLER

Whenever QuickDraw 3D needs to operate on the data encapsulated by our custom element, it will call our metahandler, which we supplied a pointer to in the registration routine. Our metahandler (Listing 4) returns the addresses of the

Listing 4. The metahandler for our custom element

```
static TQ3FunctionPointer W3Anchor_MetaHandler(TQ3MethodType methodType)
{
   switch (methodType) {
      case kQ3MethodTypeObjectTraverse:
         return (TQ3FunctionPointer) W3Anchor_Traverse;
      case kQ3MethodTypeObjectWrite:
         return (TQ3FunctionPointer) W3Anchor_Write;
      case kQ3MethodTypeObjectReadData:
         return (TQ3FunctionPointer) W3Anchor_ReadData;
      case kQ3MethodTypeElementCopyAdd:
      case kQ3MethodTypeElementCopyGet:
      case kQ3MethodTypeElementCopyDuplicate:
         return (TQ3FunctionPointer) W3Anchor_CopyAdd;
      case kQ3MethodTypeElementCopyReplace:
         return (TQ3FunctionPointer) W3Anchor_CopyReplace;
      case kQ3MethodTypeElementDelete:
         return (TQ3FunctionPointer) W3Anchor_Delete;
      default:
         return (TQ3FunctionPointer) NULL;
   }
}
```

methods associated with our element type. We supply object I/O methods to preserve our element during I/O, and copy methods to allocate and manage the string memory. We return NULL by default, to indicate that unknown methods aren't supported and that a default method should be used. The definition for each of the routines is on this issue's CD.

IMPLEMENTING THE METHODS

Listing 5 shows how the three element methods — W3Anchor_CopyAdd, W3Anchor_CopyReplace, and W3Anchor_Delete — are implemented. Note in Listing 4 that the same function, W3Anchor_CopyAdd, is used for the CopyAdd, CopyGet, and CopyDuplicate methods. This means that the data pointer passed into Q3Set_Add and Q3Set_Get is a pointer to the same structure as the internal structure. If you want to see how the I/O methods are implemented, look at the source code for our custom element on the CD.

Listing 5. Implementing the element methods

```
/* W3Anchor_CopyAdd adds the WWW data from src to dst. */
static TQ3Status W3Anchor_CopyAdd(W3AnchorData *src, W3AnchorData *dst)
{
   long  i;

   /* Check to see if src is a valid W3Anchor. */
   if (src->url == NULL)
      return kQ3Failure;

   /* We need to allocate memory for the string that belongs to dst. */
   i = strlen(src->url);
   if (i == 0)
      return kQ3Failure;
   dst->url = (char *) malloc(i + 1);
   if (dst->url == NULL)
      return kQ3Failure;

   /* Copy the string from src to dst. */
   strcpy(dst->url, src->url);

   /* Check to see if src had a description. */
   if (src->description) {
      TQ3StringObject   stringReference;
      /* Get a reference to src's description object. */
      stringReference = Q3Shared_GetReference(src->description);
      if (stringReference == NULL)
         return kQ3Failure;
      dst->description = stringReference;
   } else
      dst->description = NULL;

   /* Just copy the options, since they're just values. */
   dst->options = src->options;
   return kQ3Success;
}
```

(continued on next page)

Listing 5. Implementing the element methods *(continued)*

```
/* W3Anchor_CopyReplace substitutes the WWW data in src for the data
   in dst. */
static TQ3Status W3Anchor_CopyReplace(W3AnchorData *src,
                                       W3AnchorData *dst)
{
   long  i;
   char  *c;

   /* Check to see if src is a valid W3Anchor. */
   if (src->url == NULL)
      return kQ3Failure;

   /* We need to have enough memory for the string from src. */
   i = strlen(src->url);
   if (i == 0)
      return kQ3Failure;
   c = (char *) realloc(dst->url, i + 1);
   if (c == NULL)
      return kQ3Failure;

   dst->url = c;
   strcpy(dst->url, src->url);
   if (src->description) {
      TQ3StringObject   stringReference;

      /* Get a reference to src's description object. */
      stringReference = Q3Shared_GetReference(src->description);

      if (stringReference == NULL)
         return kQ3Failure;
      if (dst->description)
         Q3Object_Dispose(dst->description);
      dst->description = stringReference;
   } else
      dst->description = NULL;
   dst->options = src->options;
   return kQ3Success;
}

/* W3Anchor_Delete cleans up the references and memory allocations. */
static TQ3Status W3Anchor_Delete(W3AnchorData *myURLData)
{
   if (myURLData->url != NULL) {
      free(myURLData->url);
      myURLData->url = NULL;
   }
   if (myURLData->description != NULL) {
      Q3Object_Dispose(myURLData->description);
      myURLData->description = NULL;
   }
   return kQ3Success;
}
```

ATTACHING THE CUSTOM ELEMENT

Once we've set up our metahandler and associated routines, we can use the normal set and attribute set routines to add elements and attributes of our custom type. Listing 6 shows how we add our custom element to a set and attach the set to a shape object. Our geometric object will then inherit the set from the shape.

Listing 6. Adding our custom element to a set and attaching the set to an object

```
TQ3AttributeSet   theSet;
W3AnchorData      QD3DHomePage;

theSet = Q3Set_New();
if (theSet) {
   char  *description = "Apple QuickDraw 3D Home Page",
         *url = "http://www.info.apple.com/qd3d";
   QD3DHomePage.url = malloc(strlen(url) + 1);
   if (QD3DHomePage.url) {
      strcpy(url, QD3DHomePage.url);
      QD3DHomePage.description = Q3CString_New(description);
      QD3DHomePage.options = 0;

      /* Add the anchor data to the set. */
      Q3Set_Add(theSet, kElementTypeW3Anchor, &QD3DHomePage);

      /* The data has been copied and objects referenced, so we need to
         clean up after ourselves. */
      free(QD3DHomePage.url);
      Q3Object_Dispose(QD3DHomePage.description);
   }

   /* Attach the set to a shape. */
   Q3Shape_SetSet(aShape, theSet);
   Q3Object_Dispose(theSet);
}
```

GETTING THE CUSTOM DATA FROM THE OBJECT

At some point your application will want to extract the custom data you've attached to an object. In our sample application, that point is reached when the user clicks on an object or the cursor passes over an object. The W3Anchor_GetFromObject routine (Listing 7) gets custom data from an object passed into the routine, using the QuickDraw 3D routines Q3Set_Contains and Q3Set_Get.

W3Anchor_GetFromObject includes a workaround for an interesting problem. In QuickDraw 3D before version 1.0.4, if an element or attribute type was unknown (in other words, if a metahandler wasn't installed for the element or attribute), the element or attribute would be read as an unknown object. When a set was defined as part of an object derived from the Shape class, the set was written out to the metafile just fine; but when the set was read from the metafile into the shape's set, it was read as an unknown object, resulting in an additional, unnecessary level of containment, as illustrated in Figure 3. If you're reading custom elements or attributes from 3DMF files, you need to ensure that your users have version 1.0.4 or later, or you'll need to work around this issue.

Listing 7. Getting our custom data from the object

```
TQ3Boolean W3Anchor_GetFromObject(TQ3Object object, W3AnchorData *data)
{
   TQ3SetObject   set;
   TQ3Boolean     result;

   W3Anchor_Empty(data);
   data->url = NULL;
   data->description = NULL;
   set = NULL;

   /* The object passed in must be a shape or a geometry. */
   if (Q3Object_IsType(object, kQ3ShapeTypeGeometry) == kQ3True) {
      Q3Geometry_GetAttributeSet(object, &set);
      if (set != NULL) {
         result = W3Anchor_GetFromSet(set, data);
         Q3Object_Dispose(set);
         if (result == kQ3True)
            return result;
         set = NULL;
      }
   }
   if (Q3Object_IsType(object, kQ3SharedTypeShape) == kQ3True) {
      Q3Shape_GetSet(object, &set);
      if (set != NULL) {
         result = W3Anchor_GetFromSet(set, data);
         Q3Object_Dispose(set);
         return result;
      }
   }
   return kQ3False;
}

TQ3Boolean W3Anchor_GetFromSet(TQ3SetObject set, W3AnchorData *data)
{
   TQ3Object        unkObj;
   TQ3Boolean       result;
   TQ3GroupPosition position;

   result = kQ3False;

   /* Ideally, you'll find one of these. */
   if (Q3Set_Contains(set, kElementTypeW3Anchor) == kQ3True) {
      if (Q3Set_Get(set, kElementTypeW3Anchor, data) == kQ3Failure)
         return kQ3False;  /* Error: Contains, but can't get! */
      return kQ3True;
   }

   /* But due to a bug in QuickDraw 3D versions prior to 1.0.4, the
      element may be contained within another set in the unknown
      element. */
```

(continued on next page)

Listing 7. Getting our custom data from the object *(continued)*

```
   if (Q3Set_Contains(set, kQ3ElementTypeUnknown) == kQ3True) {
      if (Q3Set_Get(set, kQ3ElementTypeUnknown, &unkObj) == kQ3Failure)
         return kQ3False;  /* Error: Contains, but can't get! */
      if (unkObj == NULL)
         return kQ3False;
      /* Unknown objects may contain one object or a group. */
      if (Q3Object_IsType(unkObj, kQ3SharedTypeSet) == kQ3True)
         result = W3Anchor_GetFromSet(unkObj, data);
      else if (Q3Object_IsType(unkObj, kQ3ShapeTypeGroup) == kQ3True) {
         Q3Group_GetFirstPositionOfType(unkObj, kQ3SharedTypeSet,
            &position);
         if (position != NULL) {
            Q3Group_GetPositionObject(unkObj, position, &set);
            result = W3Anchor_GetFromSet(set, data);
         }
      }
      Q3Object_Dispose(unkObj);
   }
   return result;
}
```

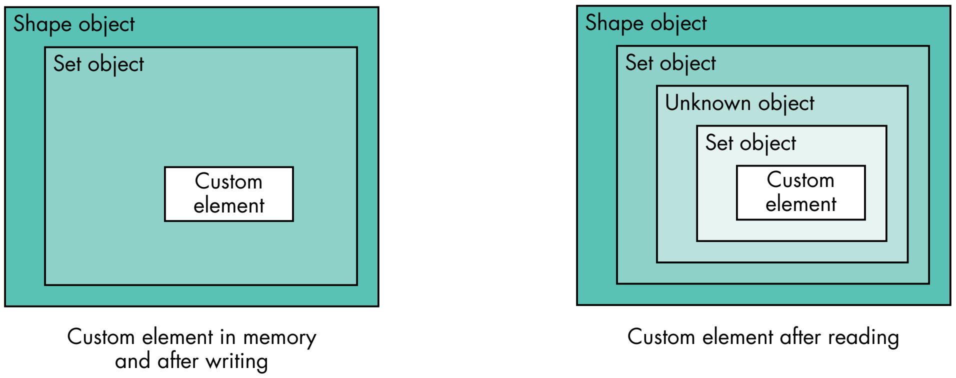


Figure 3. The problem with reading a set from a metafile

Similarly, if a metafile containing an object with a custom element attached to it is read by an application that doesn't know about that custom element, when the object is written out its associated custom element will be written out as an unknown object. The moral of this story is that you should check inside an unknown object to see if the type of attribute it contains is the one you're looking for.

SENDING THE URL TO A BROWSER

Once we've extracted the URL from our custom element, we want to send it to Netscape Navigator or a similar browser. Listing 8 shows the basics of how to do this (we've left out the proper error handling in the interest of saving space).

A more complete example on this issue's CD shows how to detect whether Netscape Navigator is running and, if not, to launch the application. It shows other cool uses for custom elements and attributes as well.•

Listing 8. Sending the URL to a browser

```
Boolean OpenURL(char *name)
{
   AppleEvent  theAppleEvent, theReply;
   OSErr       err;

   /* If Netscape isn't around, get out. */
   if (Find_Netscape() == false)
      return false;

   /* Netscape is here; let's send them an Apple event. */
   err = AECreateAppleEvent('WWW!', 'OURL', &theAddressDesc,
         kAutoGenerateReturnID, kAnyTransactionID, &theAppleEvent);
   err = AEPutParamPtr(&theAppleEvent, keyDirectObject, typeChar, name,
         strlen(name));
   err = AESend(&theAppleEvent, &theReply, kAEWaitReply,
         kAENormalPriority, kNoTimeOut, NewAEIdleProc(MyIdle),
         NewAEFilterProc(MyFilter));
   if (err == noErr)
      return true;
   else
      return false;
}
```

NEW CUSTOM ELEMENTS AND ATTRIBUTES

In future releases of QuickDraw 3D, you'll be able to ship your custom elements and attributes as a shared library that plugs into QuickDraw 3D, as opposed to having to compile the code within your application. This will allow for the custom elements and attributes to be valid for all QuickDraw 3D applications running on the machine.

In the meantime, for your custom element or attribute to be shared and understood by other applications, you can propose it to the Developer Support Center (devsupport@applelink.apple.com or AppleLink DEVSUPPORT), and they'll pass the information on to the QuickDraw 3D team. Be sure to specify the data format, describe how the object is to be used, and include a C-based implementation. We want to avoid the problems experienced with QuickDraw's picture comments, where the behavior or meaning of the data was often not clear. If enough developers request similar attributes or elements, we'll add them to the next release of QuickDraw 3D, so that they get registered at startup time. In any case, we'll make the specifications for custom attributes and elements available on the *Developer CD Series*, the *develop Bookmark* CD, and the Web (http://www.info.apple.com/qd3d).

Following are descriptions of six new custom elements that address needs expressed by several of our developers. Implementations for these are provided on the CD, in the file CustomAttribute_Lib.c. (Yes, these *are* elements, even though we on the QuickDraw 3D team have been in the sloppy habit of referring to them as attributes, and both the filename and the names of some of the elements reflect that habit. Just make sure that you're more precise in your use of the terms *element* and *attribute*, now that you know what the difference is from reading this article.) In addition, the CD contains a Technote describing some custom elements and attributes defined by our developer community.

NAMEATTRIBUTE

This element contains a string object. It can be attached to any object in the Shape class or any subclass of the Shape class. It can also be added to an attribute set and assigned to a geometry or faces. (In future releases we'll have other subclasses to the String class, allowing you to use non-ASCII characters.)

Written out in a 3DMF text metafile, this element appears as follows:

```
Container (
   NameAttribute ( )
   CString ( "1 meter box" )
)
```

SCALEATTRIBUTE

This element, of type **double**, determines the relation between one unit in the model and one meter. For example, if one unit in your model is equivalent to 10 meters, the scale should be set to 10.

This element, and all the elements whose descriptions follow, should be attached only to groups or geometry objects. Also, for each of these elements, traversal to find the element should be top down; this means that if it's attached to a group, there's no need to traverse the objects within the group.

If you add objects that have a scale element to a group, make sure that the objects are transformed (placed in a group with a transform and then added to the main group) so that the scale for the group is uniform.

```
ScaleAttribute ( 1.0 )
```

UPVECTOR AND FORWARDDIRECTION

These elements, of type TQ3Vector3D, specify the up vector and forward direction for a model. They're used to ensure that the orientation of an object read from a metafile is correct (that is, that it has the right side up and faces the right way). As for ScaleAttribute, if you add objects that have either of these elements to a group, make sure that the objects are transformed.

```
UpVector ( 0.0 1.0 0.0 )

ForwardDirection ( 1.0 0.0 0.5 )
```

W3ANCHOR

This element contains a URL in the form of a C string (ASCII only), an option field, which can be set to kURLReferenceOptionUseMap (meaning that the application should attach a pointer to the URL before sending the URL to the server), and a string object to encapsulate the description of the site pointed to by the URL (note that this allows for non-ASCII descriptions in the future).

In the following 3DMF text, the "0" signifies that there's no map. "Apple's home page" is shown as a CString when it's actually a TQ3StringObject, because that's how we decided to represent strings in the metafile.

```
Container (
   W3Anchor ( "http://www.info.apple.com" 0 )
   CString ( "Apple's home page" )
)
```

W3INLINE
This element contains a URL in the form of an C string (ASCII only). The group or geometry that this element is attached to acts as a proxy for the data pointed to by the URL. This allows the application to perform the URL data retrieval on a separate thread or in the background, or delay the operation until the user expresses interest in the proxy. Once the URL data is retrieved, the data should replace the object that holds the element.

```
W3Inline ( "http:www.info.apple.com" )
```

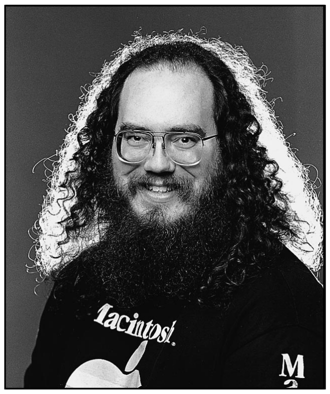
CUSTOMIZING YOUR WORLD

This article has given an overview of what you can do with custom elements and attributes. With your imagination and a few simple routines, you can extend the rich capabilities of QuickDraw 3D. Have fun with custom elements and attributes, and don't forget to tell us about them if you want other developers to be able to use yours!

RELATED READING

- "QuickDraw 3D: A New Dimension for Macintosh Graphics" by Pablo Fernicola and Nick Thompson, *develop* Issue 22, and "The Basics of QuickDraw 3D Geometries" by Nick Thompson and Pablo Fernicola, *develop* Issue 23.
- *3D Graphics Programming With QuickDraw 3D* (Addison-Wesley, 1995).
- http://www.info.apple.com/qd3d — the QuickDraw 3D home page, containing links to sites with inline 3DMF data.

Thanks to our technical reviewers Rick Evans, Robin Landsbert, Philip McBride, Klaus Strelau, and David Vasquez, and also to the entire QuickDraw 3D team.•



DAVE POLASCHEK

PRINT HINTS

The Top 10 Printing Crimes Revisited

They say some things never change. Although it's been four years since Pete "Luke" Alexander wrote about the top 10 printing crimes in *develop*, people are still committing some of the same crimes. There are also a few new crimes to add to the list. So, here they are, the updated top 10 printing crimes (ordered on a combined frequency and hideousness scale):

10. Having insufficient free memory at print time.
9. Coloring outside the lines.
8. Misusing the PostScriptHandle picture comment.
7. Calling PrintDefault or PrValidate before PrOpen.
6. Avoiding the print dialogs, especially PrJobDialog.
5. Accessing undocumented fields in the print record.
4. Not checking error return values.
3. Making low-level Printing Manager calls.
2. Not using QuickDraw GX print dialogs if QuickDraw GX is present.
1. Adding printing to your application last.

Now let's look at how to avoid these crimes. The solutions are relatively easy.

SOLVING THE PRINTING CRIMES

10. Having insufficient free memory at print time.

Some printer drivers use a lot of memory. When a driver runs out of memory, the results are usually pretty bad, so you should give printer drivers as much memory as possible. Unfortunately, memory requirements vary from driver to driver, so it's hard to predict how much memory a driver will need.

Solution: Unload all unneeded code and resources before printing.

9. Coloring outside the lines.

Many applications draw outside the printable area of the page when printing. This can happen if the user has extended objects beyond the printable area. Drawing unneeded objects causes extra work for the driver and the printer, which affects performance. In some cases, the driver also needs to allocate extra memory to hold the objects or the enclosing rectangle.

Solution: Draw only the portions of objects that will appear on the page. This can be determined by looking at the rPage field in the printer information record (which is in the prInfo field of the print record).

8. Misusing the PostScriptHandle picture comment.

The PostScriptHandle picture comment is designed to add PostScript™ code to a page containing QuickDraw graphics. It's not designed to send multiple pages of pre-generated PostScript code to a printer (for that, you need to use the Pap.WorkStation.o library).

Solution: Use the PostScriptHandle picture comment only to draw a self-contained image, which will be added to any QuickDraw graphics already on the page. If the PostScript code needs to change the graphics state, it should save and restore the state. Think of the picture comment as a way to include an EPS image, with all the restrictions placed on EPS by Adobe™ (as specified in Appendix H of the *PostScript Language Reference Manual*, second edition). The PostScript code should be compatible with both Level 1 and Level 2 PostScript, and you should include a QuickDraw version of the graphic so that your users can print to a non-PostScript printer.

For more information on how to use the PostScriptHandle picture comment, see Technote 1032, "Mixing QuickDraw & PostScript Printing from Your App: Some Gotchas," and Appendix B of *Inside Macintosh: Imaging With QuickDraw.*•

7. Calling PrintDefault or PrValidate before PrOpen.

The documentation for the Printing Manager (Chapter 9 of *Inside Macintosh: Imaging With QuickDraw*) mentions that you need to call PrOpen before calling any other Printing Manager functions. Unfortunately, the descriptions for PrintDefault and PrValidate don't repeat this warning.

DAVE POLASCHEK supports printing and font-related issues in Apple's Developer Technical Support group. Dave was last seen wandering the halls muttering, "This will all be better in Mac OS 8" and laughing maniacally.•

Solution: Always call PrOpen before calling any other Printing Manager calls.

6. Avoiding the print dialogs, especially PrJobDialog.

Some applications try to avoid print dialogs because either user interaction isn't possible or the developer thinks the user will make a mistake. Because all of the many options for the current drivers, most notably LaserWriter 8, cannot be stored in the print record, you need to call PrJobDialog so that the driver can read in the options from where they're stored (usually in the preferences file). If you don't call PrJobDialog, the driver can't set up the print record correctly, and you might not get the output you expected. The solution should be to call PrJobMerge, but in many drivers PrJobMerge does a less than perfect job.

Solution: Call PrJobDialog before printing to set up your print record correctly, or use QuickDraw GX, which supports dialog-free printing.

5. Accessing undocumented fields in the print record.

Many of the fields in the print record are undocumented or documented as private. Printer drivers can use these fields however they choose. What works for one driver might cause another to crash or to print zillions of pages you don't want.

Solution: Use only the fields in the print record that are documented as public in Chapter 9 of *Inside Macintosh: Imaging With QuickDraw*.

4. Not checking error return values.

After any call to a Printing Manager function, you should check PrError. If you're calling PrGeneral, you should also check the iError field in the TGnlData structure. Be aware that newer drivers return errors that older drivers didn't. For example, PrStlDialog in LaserWriter 8 can return an error if the preferences file is missing or corrupted; many applications don't check for this error, and later crash when they've pushed the driver completely off the cliff.

Solution: Always check and handle printing errors. See the Macintosh Technical Note "A Printing Loop That Cares..." (PR 10) and the article "Meet PrGeneral" in *develop* Issue 3.

3. Making low-level Printing Manager calls.

The low-level Printing Manager routines, such as PrDrvrOpen, are obsolete and unsupported.

Solution: Never call the low-level routines.

2. Not using QuickDraw GX print dialogs if QuickDraw GX is present.

When you call the classic Printing Manager functions and QuickDraw GX is active, the user gets the old-style "compatibility dialogs," which lack many of the features that are provided in the QuickDraw GX print dialogs. There are two problems with this: the user doesn't have access to all of the QuickDraw GX features; and when some applications call the QuickDraw GX print dialog functions and others don't, two very different printing experiences are presented to the user.

Solution: Call the QuickDraw GX print dialog functions in your print loop if QuickDraw GX is present. For help, see the article "Adding QuickDraw GX Printing to QuickDraw Applications" in *develop* Issue 19. The complete documentation can be found in *Inside Macintosh: QuickDraw GX Printing*.

1. Adding printing to your application last.

Four years later, this is still the number-one printing crime. A lot of developers leave printing until near the end of the product development cycle. When problems are encountered, Developer Technical Support gets messages like: "My application can't print, and I've *got* to ship today. Please answer as soon as possible."

Solution: Hook up your print loop as early as possible. As you add each new feature to your application, print a page or two. Make sure that things are still working as expected. When you take this approach, any features that cause printing problems get noticed early, and you'll have time to fix them.

A CLOSING NOTE

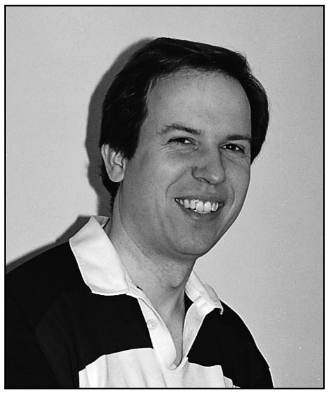
If you're committing any of the crimes on this list, your customers are probably seeing things they don't like when they print. This list is also far from comprehensive, as people continue to find new and unique ways to abuse the Macintosh print architecture.

Looking ahead, printing will be changing in a big way. Mac OS 8 will use QuickDraw GX as the printing model. As changes occur, there will be Technotes, *develop* articles, and other sources of information. So keep your eyes open, and remember, don't commit too many printing crimes. Crime doesn't pay.

Thanks to Pete "Luke" Alexander, Paul Danbold, Dave Hersey, and Jim Zandee for reviewing this column.•

64-Bit Integer Math on 680x0 Machines

When an application has to perform integer arithmetic with numbers larger than 32 bits on both the PowerPC and 680x0 platforms, you could use the floating-point types of the SANE and PowerPC Numerics libraries. But if all you really need is a larger integer, a better choice is to use the existing 64-bit math routines available on the PowerPC platform and write an equivalent library for the 680x0 Macintosh. This article presents just such a library.



DALE SEMCHISHEN

Developers of PowerPC applications that need 64-bit math can simply call the various "wide" Toolbox routines. These routines perform addition, subtraction, multiplication, division, square root, and a few other operations. On the 680x0-based Macintosh, some of these same routines are available in QuickDraw GX. But if you can't assume your customers have QuickDraw GX installed, you need a library that supports 64-bit math.

The Wide library presented in this article works on both platforms and has exactly the same interface and types as the wide routines in the Toolbox on PowerPC machines. The library also provides some new routines such as 32-bit to 64-bit add and subtract and a 64-bit-to-string conversion function. The library is included on this issue's CD, along with its source code.

All the routines use the 64-bit data type defined in the header file Types.h, which is the standard type used for signed 64-bit integers on both the PowerPC and 680x0 Macintosh:

```
struct wide {
   Sint32   hi;   /* upper 32 bits (signed) */
   Uint32   lo;   /* lower 32 bits (unsigned) */
};
typedef struct wide wide, *WidePtr;
```

THE WIDE ROUTINES

Before plunging into the Wide library, let's see what 64-bit math routines I'll be talking about. First, I'll introduce those that are available on PowerPC machines, then those you'll find on a 680x0 Macintosh with QuickDraw GX, and finally the routines in the Wide library.

DALE SEMCHISHEN (Dale_Semchishen @mindlink.net) lives in Vancouver, British Columbia, with his wife Josephine. He works for Glenayre Technologies as a paging software developer (they make the control systems that send messages to your belt beeper). Recently, he had to accept the fact that the world is changing when his retired father started talking about his Internet provider.•

POWERPC TOOLBOX

In the header file FixMath.h, the routines listed in Table 1 are defined for 64-bit math on the PowerPC platform.

Table 1. Wide routines in the PowerPC Toolbox

Routine	Operation
WideAdd	Add two 64-bit integers
WideCompare	Compare two 64-bit integers
WideNegate	Negate a 64-bit integer
WideBitShift	Shift a 64-bit integer
WideShift	Shift a 64-bit integer with rounding
WideSquareRoot	Calculate the square root of a 64-bit integer
WideSubtract	Subtract two 64-bit integers
WideMultiply	Multiply two 32-bit integers
WideDivide	Divide a 32-bit integer into a 64-bit integer (32-bit quotient)
WideWideDivide	Divide a 32-bit integer into a 64-bit integer (64-bit quotient)

680X0 QUICKDRAW GX

On 680x0 machines that have QuickDraw GX installed, all the wide routines for the PowerPC platform listed in Table 1 are available, with the exception of WideBitShift. The QuickDraw GX header file GXTypes.h defines the wide routine types and function prototypes in exactly the same way that the header file FixMath.h does for PowerPC machines.

In addition, QuickDraw GX on 680x0 machines has a routine that the PowerPC platform doesn't have: WideScale. This function returns the bit number of the highest-order nonzero bit in a 64-bit number. The Wide library implements this function on the PowerPC platform.

THE WIDE 64-BIT LIBRARY

The Wide 64-bit integer math library on this issue's CD provides all the wide routines that are available on PowerPC machines and on 680x0 machines with QuickDraw GX, plus a few extras. The extra routines, which are available on both the PowerPC and 680x0 platforms, are listed in Table 2.

Table 2. Extra routines in the Wide library

Routine	Operation
WideAssign32	Set a 64-bit integer to the value from a 32-bit integer
WideAdd32	Add a 32-bit integer to a 64-bit integer
WideSubtract32	Subtract a 32-bit integer from a 64-bit integer
WideToDecStr	Convert a 64-bit integer to a SANE **decimal** string
WideInit	Initialize the library (optional)

WideAssign32, WideAdd32, WideSubtract32. These routines are self-explanatory.

WideToDecStr. This routine converts a signed 64-bit integer to the SANE string type **decimal**, which is also defined by the PowerPC Numerics library. This string structure is a good intermediate format for final conversion to a string format of your choosing.

Since WideToDecStr calls the SANE library to generate the string, SANE must be linked with your 680x0 application. The SANE library is included with all the major development systems.

To convert the string returned by WideToDecStr to a Pascal string, call the SANE routine **dec2str**.

If you want to generate a localized number, take a look at the article "International Number Formatting" in *develop* Issue 16. You could call the LocalizeNumberString function from that article after converting the output of WideToDecStr to a Pascal string, or you could modify LocalizeNumberString to accept the output of WideToDecStr.•

WideInit. The library is self-initializing; the first time you call any wide routine, WideInit is also called. If the execution speed of your first runtime call to a wide routine is important, you have the option of calling WideInit during your application's startup to avoid that overhead.

The purpose of WideInit is to determine what processor is being used, or emulated; it calls Gestalt to make this determination. If your Macintosh has a 68020–68040 CPU (68020, 68030, or 68040), the library will use the 64-bit multiply and divide instructions available on that processor; otherwise, the library will have to call software subroutines for those operations. On 68000 machines, such as the Macintosh Plus and SE, the processor's multiply instruction is limited to 32 bits and the library has no choice but to use the slower algorithmic approach for multiplication and division.

SOURCE CODE ON A PLATTER

The library can be compiled on the 680x0 and PowerPC platforms using either the Metrowerks CodeWarrior or Symantec C development system. The library tests which development system is compiling it and, if it's not CodeWarrior or Symantec, the preprocessor displays an error message saying the library needs to be ported to your environment. This is necessary because there's some inline assembly language in the source file, as discussed later in this section, and different C compilers handle assembly language differently.

While the interface routines to our 64-bit library are the same on the PowerPC and 680x0 machines, when you compile the library a different subset of routines is linked in, depending on your environment:

- If you build the library for a 680x0 machine without QuickDraw GX headers, all the Wide library routines are defined.
- If you build the library for a 680x0 machine and include the QuickDraw GX header file GXTypes.h or GXMath.h before the Wide library's Wide.h header file, the extra routines and the WideBitShift routine are defined. The other wide routines are already available via the QuickDraw GX traps.
- When you compile for the PowerPC platform, only the five extra routines (WideAssign32, WideAdd32, WideSubtract32, WideToDecStr, and WideInit) are defined in the library. All the other wide routines already exist in the PowerPC Toolbox. Additionally, if GXTypes.h or GXMath.h isn't included, WideScale is defined.

Table 3 summarizes where the wide routines can be found on the different platforms.

Table 3. Where to find wide routines

Routine	680x0	680x0 with QuickDraw GX	PowerPC
WideInit	Wide.c	Wide.c	Wide.c
WideAdd	Wide.c	QuickDraw GX	PowerPC Toolbox
WideAdd32	Wide.c	Wide.c	Wide.c
WideAssign32	Wide.c	Wide.c	Wide.c
WideBitShift	Wide.c	Wide.c	PowerPC Toolbox
WideCompare	Wide.c	QuickDraw GX	PowerPC Toolbox
WideDivide	Wide.c	QuickDraw GX	PowerPC Toolbox
WideMultiply	Wide.c	QuickDraw GX	PowerPC Toolbox
WideNegate	Wide.c	QuickDraw GX	PowerPC Toolbox
WideScale	Wide.c	QuickDraw GX	Wide.c or QuickDraw GX
WideShift	Wide.c	QuickDraw GX	PowerPC Toolbox
WideSquareRoot	Wide.c	QuickDraw GX	PowerPC Toolbox
WideSubtract32	Wide.c	Wide.c	Wide.c
WideSubtract	Wide.c	QuickDraw GX	PowerPC Toolbox
WideToDecStr	Wide.c	Wide.c	Wide.c
WideWideDivide	Wide.c	QuickDraw GX	PowerPC Toolbox

Note that the Wide library decides at *compile* time which routines to use. When QuickDraw GX header files are not included, the Wide library routines are called. If your application needs to make a runtime decision about whether to use QuickDraw GX, you'll need to make some changes to the library. One solution is to rename the Wide library routines and remove the conditional compilation tests for QuickDraw GX from the source. Then at run time you can decide which version to call — the QuickDraw GX routines if they're available, or the internal Wide library routines if not.

UNIVERSAL HEADERS

The Wide library was compiled with version 2.1 of Apple's universal headers. The latest headers are available on this issue's CD. You should make sure you have a recent version of these headers, because the library uses the constant GENERATING68K. If the header file ConditionalMacros.h doesn't contain this constant, your version of the universal headers is too old.

680X0 ASSEMBLY LANGUAGE

Some of the routines in the library are written in assembly language to take advantage of the 64-bit multiply and divide instructions on 68020–68040 machines, because on these machines the C language will use only 32-bit multiply and divide instructions. On PowerPC machines, the Wide library doesn't need assembly language because the 64-bit multiply and divide routines are provided by the Toolbox.

The library's source file Wide.c contains both C and assembly language. It has been successfully compiled by Symantec C 7.0.4 and CodeWarrior 7. If you want to compile the library on any other development system, you may have to do a little work porting it. Most of the changes will be confined to the conditional compilation statements at the beginning of Wide.c where the differences in SANE types and inline assembly language are handled.

A CLOSER LOOK AT SOME WIDE ROUTINES

Now let's look at a couple of the more interesting routines in the Wide library to see how they work. See the source code on the CD for full implementations of all the routines.

WIDEMULTIPLY

WideMultiply (Listing 1) performs a 32-by-32-bit multiply and produces a 64-bit result. The first and second parameters are the two signed 32-bit integers to be multiplied together. The return value is a pointer to the 64-bit result that's also returned via the third parameter.

Listing 1. The multiply routine

```
wide *WideMultiply (
   long multiplicand,  /* in: first value to multiply */
   long multiplier,    /* in: second value to multiply */
   wide *target_ptr)   /* out: 64 bits to be assigned */
{
   /* Initialize Wide library if not already done. */
   if (!gWide_Initialized) WideInit();

   /* If the 64-bit multiply instruction is available... */
   if (gWide_64instr) {
      /* Execute the assembly-language instruction MULS.L */
      Wide_MulS64(multiplicand, multiplier, target_ptr);
   }
   else {
      /* Call the Toolbox to perform the multiply. */
      LongMul(multiplicand, multiplier, (Int64Bit *) target_ptr);
   }

   return (target_ptr);
}
```

WideMultiply first tests whether the library has been initialized yet; if not, it calls WideInit. Next the routine tests whether the 64-bit multiply instruction is available on the current CPU by examining the global variable gWide_64instr (which was set by the initialization routine WideInit). If the instruction is available, WideMultiply calls the assembly-language function Wide_MulS64 to take advantage of it (as described later); otherwise, WideMultiply calls the Toolbox routine LongMul to perform the multiplication, as would be the case on 68000 machines.

WIDESQUAREROOT

The WideSquareRoot function (Listing 2) takes a 64-bit unsigned number as input and returns a 32-bit unsigned result. All possible results can be expressed in 32 bits, so overflow isn't possible.

For this routine I decided to let the SANE library do the work of generating the square root. The routine converts the 64-bit input number to an 80-bit floating-point number and then calls the SANE library function **sqrt** to calculate the square root. Finally, WideSquareRoot converts the resulting 80-bit floating-point number back to a 64-bit integer and returns the low-order half of the result.

When a 64-bit integer is converted to an 80-bit floating-point number, no loss in precision occurs. An 80-bit floating-point number is made up of three parts — the sign (1 bit), the exponent (15 bits), and the fractional part (64 bits). As you can see, a 64-bit integer exactly fits in the fractional part.

Listing 2. The square root routine

```
unsigned long WideSquareRoot (
   const wide *source_ptr)   /* in: value to take the square root of */
{
   wide          work_integer;
   Extended_80   extended_80_number;

   /* Initialize Wide library if not already done. */
   if (!gWide_Initialized) WideInit();

   /* Convert "wide" number to "extended" format. */
   Wide_ToExtended(&extended_80_number, source_ptr);

   /* If compiling with CodeWarrior, the parameter to sqrt is a
      pointer instead of a value, as defined in PowerPC Numerics. */
#ifdef __MWERKS__
   Sqrt(&extended_80_number);
#else
   extended_80_number = sqrt(extended_80_number);
#endif

   /* Convert "extended" format to "wide" number. */
   Wide_FromExtended(&work_integer, &extended_80_number);

   /* OK to ignore work_integer.hi as it's always 0. */
   return (work_integer.lo);
}
```

Two differences between the CodeWarrior and Symantec development systems that show up in the Wide library's WideSquareRoot function are the 80-bit floating-point types and the parameters of the SANE library's square root function. Under CodeWarrior, the Wide library internal type Extended_80 is defined as the type **extended80**, and Sqrt returns the result to the same location as the input number. Under Symantec C, Extended_80 is defined as the type **extended**, and **sqrt** returns the result as a function return value.

INTERNAL ASSEMBLY-LANGUAGE ROUTINES

The Wide library uses internal assembly-language routines to execute 64-bit multiply and divide instructions on machines that support those instructions. In case you're interested, here are the details.

Symantec and CodeWarrior handle the **asm** keyword differently, so I used some preprocessor commands (#defines) to handle the differences between the two development systems. Near the beginning of the Wide.c source file there are four #defines that differ depending on which development system you're using, as shown in Table 4.

WIDE_MULS64

Wide_MulS64 (Listing 3) is an internal assembly-language routine that WideMultiply calls to execute the 64-bit multiply instruction on the 68020–68040 CPUs. It starts with ASM_FUNC_HEAD, as mentioned in Table 4. The three definitions at the

Table 4. Wide.c #defines

Name	Where used	Symantec	CodeWarrior
ASM_FUNC_HEAD	Just before the function definition	Not defined	**asm**
ASM_BEGIN	Just before the first assembly-language statement in the function	**asm {**	LINK A6, #0
ASM_END	Just after the last assembly-language statement in the function	}	UNLK A6
ASM_TAIL	Just after the function definition	Not defined	RTS

Listing 3. 64-bit multiply instruction

```
ASM_FUNC_HEAD static void Wide_MulS64 (
   long multiplicand,   /* in: first value to multiply */
   long multiplier,     /* in: second value to multiply */
   wide *out_ptr)       /* out: 64 bits to be assigned */
{
#define MULTIPLICAND    8
#define MULTIPLIER      12
#define OUT_PTR         16

ASM_BEGIN
      MOVE.L   MULTIPLICAND(A6),D0     //
      DC.W     0x4C2E,0x0C01,0x000C    // MULS.L multiplier(A6),D1-D0
      MOVE.L   OUT_PTR(A6),A0          //
      MOVE.L   D0,WIDE_LO(A0)          //
      MOVE.L   D1,WIDE_HI(A0)          //
ASM_END
ASM_FUNC_TAIL
}
```

start of the function (MULTIPLICAND, MULTIPLIER, and OUT_PTR) are the byte offsets to the parameters. Although in Symantec C it's possible to refer to function parameters by name via A6, this isn't possible in CodeWarrior. I had to give up accessing the parameters by name and use #defines instead.

To execute the 64-bit multiply instruction I had to define it with a DC.W directive that generates the desired object code. This was necessary because the Symantec C inline assembler supports only the 32-bit multiply instruction and won't recognize the 64-bit assembly opcode.

WIDE_DIVIDEU

If the 64-bit divide instruction isn't available, the library calls the internal assembly-language routine Wide_DivideU (Listing 4) to perform the division using an algorithm. The algorithm is basically a binary version of the paper and pencil method of doing long division that all of us learned in school. It's a loop that executes once for each bit in the size of the divisor, which is 32 in our case. The Wide_DivideU subroutine actually handles only unsigned division, but the library function that calls it will take care of converting the input parameters to positive values and, if required, converting the result to a negative value.

The top of the assembly-language loop starts at the **@divloop** label. For each loop, the algorithm shifts the quotient and the remainder left one bit position before trying to subtract the divisor from the remainder. If the subtraction can be done, the least-significant bit in **quotient.lo** is set; otherwise, the subtraction is undone by the add instruction near the **@div50** label. Then, if the divisor is greater than the loop bits that are accumulating in register D4, the least-significant bit in **quotient.hi** is set.

Listing 4. 64-bit unsigned division algorithm

```
ASM_FUNC_HEAD static void Wide_DivideU (
   wide *dividend_ptr,    /* in/out: 64 bits to be divided */
   long divisor,          /* in: value to divide by */
   long *remainder_ptr)   /* out: the remainder of the division */
{
#define DIVIDEND_PTR   8
#define DIVISOR        12
#define REMAINDER_PTR  16

ASM_BEGIN
      MOVEM.L  D2-D7,-(SP)          // save work registers
      CLR.L    D0                   //
      CLR.L    D1                   // D0-D1 is the quotient accumulator
      MOVE.L   DIVIDEND_PTR(A6),A0  //
      MOVE.L   WIDE_HI(A0),D2       //
      MOVE.L   WIDE_LO(A0),D3       // D2-D3 = remainder accumulator
      CLR.L    D4                   //
      MOVE.L   D2,D5                // D5 = copy of dividend.hi
      MOVE.L   DIVISOR(A6),D6       // D6 = copy of divisor

      MOVEQ.L  #31,D7               // FOR number of bits in divisor
@divloop:
      LSL.L    #1,D0                // shift quotient.hi accum left once
      LSL.L    #1,D1                // shift quotient.lo accum left once
      LSL.L    #1,D4                //
      LSL.L    #1,D3                //
      ROXL.L   #1,D2                // shift remainder accum left once
      SUB.L    D6,D2                // remainder -= divisor
      BCS      @div50               // If CS, remainder is negative
      BSET     #0,D1                // quotient.lo |= 1
      BRA.S    @div77               //
@div50:
      ADD.L    D6,D2                // remainder += divisor
@div77:
      BTST     D7,D5                //
      BEQ      @div90               // If EQ, bit not set in dividend.hi
      BSET     #0,D4                //
@div90:
      CMP.L    D6,D4                //
      BCS      @div99               // If CS, divisor < D4
      SUB.L    D6,D4                // D4 -= divisor
      BSET     #0,D0                // quotient.hi |= 1
@div99:
      DBF      D7,@divloop          // loop until D7 == -1
```

(continued on next page)

Listing 4. 64-bit unsigned division algorithm *(continued)*

```
      MOVE.L   DIVIDEND_PTR(A6),A0  // output the remainder
      MOVE.L   D0,WIDE_HI(A0)       //
      MOVE.L   D1,WIDE_LO(A0)       //
      MOVE.L   REMAINDER_PTR(A6),A0 // output the remainder
      MOVE.L   D2,(A0)              //
      MOVEM.L  (SP)+,D2-D7          // restore work registers
ASM_END
ASM_FUNC_TAIL
}
```

Notice that the first assembly-language statement in Wide_DivideU is a MOVEM.L instruction that saves on the stack all the registers that the division loop uses; the last instruction is a MOVEM.L instruction that restores these registers. Fortunately, this subroutine can place all its working variables in registers and avoid the stack for its loop, thus improving performance.

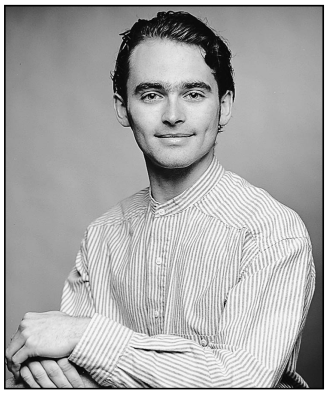
WORLDS APART

There you have it. Now 64-bit integer math can be handled with the same API on both the 680x0 and PowerPC platforms. Having the same function-level interface on these two very different processors makes life a lot easier for application programmers. Don't you wish all libraries had the same interface regardless of the CPU or system software version?

Thanks to our technical reviewers Dave Evans, Quinn "The Eskimo!", and Dave Radcliffe.

Special thanks to Dave Johnson for software testing.•

Learn how software components allow you to simplify the way you develop software.
Take Developer University's Creating OpenDoc Parts class to learn how to create parts using the OpenDoc Framework.
5 days, $1500.
May 20-24, June 24-28.
Call Developer University to register now at (408)974-4897 or e-mail devuniv@applelink.apple.com. Courses are offered in Cupertino, California.
DEVELOPER DU UNIVERSITY



DAVE EVANS

BALANCE OF POWER

Sleuthing Through Your Code

The night was well advanced, but the bright glow of fluorescent lamps misrepresented time. As I sat back in my comfortable chair, rubbing tired eyes, I wondered what the venerable but fictional Mr. Sherlock Holmes would offer me as advice. Perhaps because I was so weary from the long hours of debugging, I easily imagined Mr. Holmes sitting near me in a tweed suit smoking his pipe. Certainly he would address me as he once addressed his compatriot Dr. Watson, with a slightly condescending tone, and he would tell me that in my debugging I was missing the key iota of information.

At that moment, a solitary number seemed brighter on my monitor. Perhaps I have an overactive imagination, but it seemed as if MacsBug were magically illuminating that crucial, overlooked information. My computer was at interrupt level 2, yet it was waiting for a driver request to complete. How could I have missed the interrupt level earlier? It was no wonder that the computer froze. My software had most likely called the driver synchronously at exactly the wrong time. The voice of Mr. Holmes rang again in my ears. This time he quoted from that unfortunate story "A Case of Identity" when he said, "It has long been an axiom of mine that the little things are infinitely the most important."

Sir Arthur's famous detective was unsurpassed as an observer of detail. He believed that keen attention to all things — even the mundane — was the key to good detective work. In debugging software, I've found this advice is also true. Although many software bugs can be solved quite easily, the most challenging problems demand more attention. This is especially true of crashes or freezes in your software. To find the detail we need for those, we often have to go below source-level tools and get comfortable with lower-level aids.

In this column I'll take you through some low-level debugging techniques. I'll start with basic strategy and then discuss particular methods and examples. Although many details will be PowerPC-specific, much of the information here is useful on all Macintosh computers.

THE STRATEGY OF A SLEUTH

The experienced engineer starts with a basic strategy when faced with a troublesome software crash or freeze. The strategy is similar to Mr. Holmes's approach to solving difficult crimes. Using the scientific method, he starts by collecting key information and details. When he has finished researching, he begins to analyze the information and eliminates hypothesis after hypothesis. Once close to a solution, he seeks out more detail to narrow his suspects to a single culprit. Similarly, your strategy for debugging software should start with careful observation and research. Then you should hypothesize, test your theories, and collect more detail. This narrowing approach will draw you closer to the pernicious coding error in your software.

It's tempting when faced with a difficult crash to experiment instead of researching it first. But beware! Don't just reimplement your code with new approaches until it stops crashing. Though some may cynically suggest that that's the Macintosh way to program, don't be lulled into this strategy. I've found that it usually produces unstable code and ultimately takes longer than researching the original problem.

In researching a crash or a freeze, the private bug detective should first ask these few basic questions:

1. What kind of crash or freeze is this?
2. What code did the computer stop in?
3. How did I get to that code?

For these, you'll need a low-level debugger (such as MacsBug). Let's look at each one in turn.

GET YOUR BEARINGS

The first step is to determine the kind of problem you've got. For crashes there are a number of possible problems, including the all-too-familiar illegal instruction and bus errors. Note that PowerPC exception handlers don't currently distinguish between these or other types. In MacsBug the correct type will be reported, but your debugger may instead describe all crashes as general spurious interrupts or type 11 errors.

DAVE EVANS still works at Apple in the Mac OS System Software group. He always enjoyed Sherlock Holmes stories while he was growing up, and he was excited to learn that most of the stories are no longer protected under copyright and are easily accessible on the Internet (see the 221B Baker Street Web page at http://www.contrib.andrew.cmu.edu/u/mset/holmes.html).•

If your crash is from an illegal instruction error, it's possible that the processor jumped to an invalid address or the intended code moved in memory. In this case you'll notice (in a disassembly where execution stopped) that most instructions are invalid or nonsense. This can also occur if the emulator tries to emulate PowerPC code, or if the processor tries to execute 680x0 code as PowerPC code. Try disassembling memory as both PowerPC code (using **ipp pc**) and 680x0 code (using **ip pc**).

If your crash is from a bus error, the most likely cause is an invalid address in some register. Disassemble memory where execution stopped and examine the instructions. If there are instructions that dereference registers, inspect those registers for addresses that aren't in a valid range. If you're debugging 680x0 code on a Power Macintosh, you'll need to look at all the instructions near the crash, because the 680x0 emulator won't tell you exactly which instruction caused the error.

Researching a freeze requires a different approach. If the freeze prevents you from using any debugging tools, you must isolate the offending code by watching the computer execute up to the freeze. Setting breakpoints, tracing, and stopping execution at known locations will bring you closer. This approach is slow but will lead you to the code that caused the error or to the state that prompted it. If the computer is frozen but you can still use debugging tools, it's very possible that you're in an infinite loop.

THE LAYOUT OF THE CRIME SCENE

Sherlock Holmes sometimes astonished readers by deducing crimes just from hearing second-hand details. He was also known, however, to walk the back alleys of London and gumshoe the scene of a crime when necessary. Learning the layout of the crime scene was crucial for a number of his deductions. When staring at your newly crashed software, do you recognize the code that your debugger is displaying? Disassemble memory near the location of the crash and snoop around for clues. Check for the following to determine how your computer came to this final resting place:

- If you're using MacsBug, use the **wh pc** command to check where the code is.
- Display memory and disassemble from the beginning of the code's block of memory.
- Does the code nearby reference strings or Gestalt selectors?
- Look for text symbols and strings in the code.

If you've crashed in PowerPC code, most low-level debuggers will give great information about where you are. This is because most PowerPC code is registered and linked using the Code Fragment Manager, which these debuggers can access for hints. For example, if you use the **wh pc** command in MacsBug, after crashing in PowerPC code you'll see something like this:

```
Address 000BAE34 is in the System heap
   at 00002800 at NQDColor2Index+00018
The address is in a CFM fragment "NQD"

It is 0001AD28 bytes into this heap block:
    Start    Length      Tag  Mstr Ptr Lock
 • 000A00F0 0003DB00+04   R   00002AC4   L
```

Here we see that the computer crashed at a location 24 bytes from the beginning of the NQDColor2Index routine. This routine is in the NQD (or Native QuickDraw) code fragment. Since this address is close to the beginning of the routine, we can disassemble from its start and examine the six instructions that executed before the crash for more clues:

```
Disassembling PowerPC code from bae00
  NQDColor2Index
    +00000 000BAE00   li      r5,0x0000
    +00004 000BAE04   lwz     r4,TheGDevice(r0)
    +00008 000BAE08   sth     r5,QDErr(r0)
    +0000C 000BAE0C   stw     r31,-0x0004(SP)
    +00010 000BAE10   lwz     r5,0x0000(r4)
    +00014 000BAE14   addi    r31,r3,0x0000
    +00018 000BAE18  *lwz     r3,0x000C(r5)
```

A bus error at NQDColor2Index+00018 would occur if register R5 contained an invalid address. Look at the register display to validate that hypothesis. Notice in the code that R5 is a dereference of R4, which comes from the low-memory global TheGDevice. Here we crashed because TheGDevice had become invalid, so now your investigation turns toward that global.

A freeze will typically occur because of a double page fault or exception or because of an infinite loop. Synchronous driver calls will also freeze if called when the interrupt level is above 0. A double fault or exception is common only if you're writing driver software. Your computer can handle only one page fault or exception at a time. A double fault or exception occurs when software that services a fault subsequently causes a second fault. For example, disk drivers are sometimes called by the Virtual Memory Manager to help service page faults; therefore, if you develop a disk driver you must take care not to cause page faults since you may be asked to service one as well.

A good way to detect infinite loops is to trace for a few instructions using your debugger. If you notice the same set of instructions being repetitively executed, you could be in an infinite loop. Look at branch instructions for clues to why the loop isn't completing. A special case of these loops is the vSyncWait routine. It looks like this:

```
MOVE.W      $0010(A0),D0
BGT.S       *-6
```

This tight loop is waiting for the two-byte value located 16 bytes from register A0 to become 0 or negative. This is a standard sequence to wait for a driver request to complete. The driver request is described in an IOParam record pointed to by register A0. When the driver is done servicing the request, it will interrupt the loop and modify the ioResult field 16 bytes into that record. It will then return from the interrupt, and the loop will complete normally. A freeze in this loop means the driver isn't servicing the request. If you typed **dm a0 iopb** in MacsBug, you might see something like this:

```
 Displaying IOParamBlockRec at 000003A4
  000003A4  qLink             NIL
  000003A8  qType             0002
  000003AA  ioTrap            A003
  000003AC  ioCmdAddr         NIL
  000003B0  ioCompletion      NIL
  000003B4  ioResult          0001
  000003B6  ioNamePtr         NIL
  000003BA  ioVRefNum         0008
  000003BC  ioRefNum          FFDF
  000003BE  ioVersNum         #0
  000003BF  ioPermssn         #23
  000003C0  ioMisc            NIL
  000003C4  ioBuffer          01C7E2B0
  000003C8  ioReqCount        00010000
  000003CC  ioActCount        00010000
  000003D0  ioPosMode         0001
  000003D2  ioPosOffset       1B84AA00
```

Take note of the ioTrap and ioRefNum fields. In this case, ioTrap is $A003, which is the synchronous Read trap. Using the **drvr** dcmd in MacsBug, you'll find that the driver with refNum $FFDF is .ASYC00, which is the SCSI driver. This hang, then, occurs during a synchronous Read call to the SCSI driver. Perhaps I should next check the current interrupt level.

HOW DID WE GET THERE?

After a long, ponderous silence, while sharply focused on the current enigma, Holmes might startle you by saying, "Let us reconstruct, Watson." Then he would describe the probable series of events that preceded that particular criminal act. If the reconstruction wasn't adequate to identify a perpetrator, at least it would review the crucial discoveries so far. It would show Holmes's appreciable progress toward a solution. Similarly, while in the midst of a difficult debugging task, you should reconstruct the turn of events to gain extremely helpful information.

Figuring out what happened, once the computer is stopped cold in a crash or a freeze, isn't easy. In effect, you're looking for footsteps in the sand that are often obscured or covered with other false marks. For this task, the technique we most often use is the stack crawl.

Procedural programming on the Macintosh uses a stack. For each procedure call, the stack is added to, and vital clues such as return addresses and stack frame pointers are left for us to find. In PowerPC code, the link register adds to our clues and is guaranteed to point back to the penultimate procedure of interest. Your low-level debugger will certainly have a stack crawl tool to use as well.

In MacsBug, the **sc** and **sc7** commands are your basic stack-crawling aids. Start your search with the **sc** command, which looks for stack frames. Frames are structures found on the stack containing both the return address and a pointer to the previous frame. In PowerPC code the frames also contain a standard area to preserve basic registers. Fortunately, frames are required in PowerPC code and follow a standard format. Most 680x0 compilers will generate stack frames as well, although much of the 680x0 system software was written in assembly language without frames. If during your crash you have a valid stack frame address in register A6 or R1, the **sc** command will show you a history of which code execution preceded your software's demise. Listing 1 shows a basic **sc** command's result.

In this example the first two links are in a CODE resource from file number $0F6E. Use the MacsBug **file** command to determine which file they were loaded from. It's likely that they're from the current application, and the return addresses displayed in the Caller column (01C139CA and 01C132EA) are most likely in the application's binary. The return addresses listed are crucial to your sleuthing. They not only point out where execution would have returned to but, more important, they show which instructions were recently executed: the ones just before the return address. Those addresses are your footprints in the sand. They are clues in your reconstruction, and they hint to the turn of events that led to the crash or freeze.

Note the third and fourth lines in Listing 1, which show return addresses in an 'scod' resource. Those 'scod' resources implement the Process Manager. It's possible that the application binary, probably at the instruction just before address 1C132EA, made a call to the Process Manager.

Listing 1. Display from the **sc** command

```
 Calling chain using A6/R1 links
  Back chain  ISA  Caller
  01C8A0AC    68K  01C139CA  'CODE 0001 0F6E Main'+03A1A
  01C8A0A0    68K  01C132EA  'CODE 0001 0F6E Main'+0333A
  01C89F4A    68K  00058748  'scod BFB1 011C'+01A38
  01C89E6A    68K  00064090  'scod BFB1 011C'+0D380
  01C89E40    68K  408787FC  CHECKUPDATESEARCH+0003E
  01C89E16    68K  40878426  __GETSUBWINDOWS+000D6
```

Listing 2. Display from the **sc7** command

```
Return addresses on the stack
  Stack Addr  Frame Addr   ISA   Caller
   01C8A0B0                68K   01C16D62 'CODE 0001 0F6E Main'+06DB2c
   01C8A0A4    01C8A0A0    68K   01C139CA 'CODE 0001 0F6E Main'+03A1A         *
   01C8A094                68K   40849116 UNLOADSEG+00046
   01C8A06A    01C8A066    68K   409CFFFC DISPTABLE+8D0BC
   01C8A018                68K   4087EAF0 GETRESOURCE+000B2
   01C8A00E                68K   408806F6
   01C8A008                PPC   00094BE8 EmToNatEndMoveParams+00014
   01C89FF8                68K   0011ACDA
   01C89FE0                68K   4087ECFE VRMGRSTDENTRY+000B0
   01C89FDC                68K   4087ECFE VRMGRSTDENTRY+000B0
   01C89FD8                68K   0011A5B4
   01C89F4E    01C89F4A    68K   01C132EA 'CODE 0001 0F6E Main'+0333A         *
   01C89F4A                68K   01C8A09E
   01C89F22    01C89F1E    68K   00058748 'scod BFB1 011C'+01A38              *
   01C89F1E                68K   01C89F48
   01C89EDE    01C89EDA    68K   00163E30
   01C89EDA                68K   01C89F1C
   01C89E62                68K   01C8AFBE
   01C89E44    01C89E40    68K   00064090 'scod BFB1 011C'+0D380              *
   01C89E1A    01C89E16    68K   408787FC CHECKUPDATESEARCH+0003E             *
   01C89DF4    01C89DF0    68K   40878426 __GETSUBWINDOWS+000D6               *
   01C89DE2                68K   4087876E CALCANCESTORRGNS+0002A
   01C89DDE                68K   001191E6
```

The fifth and sixth lines of the display show return addresses in the Macintosh ROM. The symbols are shown because I've installed a ROM map file in my MacsBug Preferences folder. You should use the provided ROM map file for your computer, because it will often give you better stack crawl information. You can also deduce that these return addresses are in the ROM from the addresses themselves. Most Macintosh ROMs begin at memory address $40800000. PCI-based Macintosh ROMs currently begin at $FFC00000, and PowerPC processor–based PowerBook ROMs at $40000000. You can determine the beginning address of your ROM by looking at the ROMBase low-memory global. In MacsBug, for example, type **dl ROMBase** to display the beginning ROM address.

The **sc7** command in MacsBug gives you less precise information. In cases when you don't have stack frames, you can ask your debugger to display all possible return addresses on the stack. Your debugger will intelligently guess which values on the stack are possible return addresses, but most of the information displayed will be extraneous. You must pick through this information for clues — an arduous task. The stack frame–based crawl is neat and tidy, whereas the same situation would produce the **sc7** display shown in Listing 2. I've added an asterisk (*) on each line that's also in the **sc** command's display.

In this example, there were a number of values on the stack that might have been valid return addresses. The six we saw in the **sc** command's display are there. Many of the other lines will not be relevant return addresses, because many procedures reserve space on the stack but don't always use it or initialize it. There will often be old return addresses in that unused part of the stack. These old return addresses are like very faint footprints in the sand — from some previous execution — and they may tell you what occurred much earlier in time. More often, though, they'll just be distracting and irrelevant to your search.

Be very wary of an **sc7** command when tracing through PowerPC code. PowerPC code typically has large stack frames, at least 56 bytes for each procedure, and the code often doesn't use all those bytes. This will cause many old return addresses to stay in the unused parts of the stack frame, and those old addresses will appear in your **sc7** command's display.

Sometimes you'll notice that the **sc** and **sc7** commands fail to work. In MacsBug, you may see the error

```
Bad stack: stack pointer must be even and
   <= stack base
```

There's more than one stack that the system uses, but the stack base that MacsBug refers to in this error is the application stack's base or top address. The **sc** and **sc7** commands first check to see if the A6, A7, and R1 registers point to locations below the application stack's base. If they don't, MacsBug returns this error. The executing code may be using a different stack, however. Many parts of the Mac OS system software use separate stacks. To force MacsBug to execute a stack crawl anyway, specify the register to use and the amount of memory to search through. For example, the MacsBug commands **sc7 a7 4000** and **sc a6 4000** will execute a stack crawl even if the A6 and A7 registers point above the application stack's base.

System stacks vary in size from about 8000 bytes up to 48000 bytes. There's no easy way to determine the base of a system stack that's in use. If you don't get interesting clues from 16384 bytes ($4000 in hex), vary the number of bytes you specify and compare your results.

ELEMENTARY, OF COURSE

Don't be pacified by source-level debuggers. Lower-level tools give you a much better understanding of the Mac OS and your code. These tools also give you the ability to research the most complicated problems. Strive to be a software sleuth, and you'll gain some truly useful expertise.

Thanks to Geoff Chatterton, Doug Clarke, Michael Dautermann, and Tim Maroney for reviewing this column.•

Want to Show off your cool code?

YOUR NAME HERE

Do you have code that solves a problem other Macintosh developers might be having? Why not show it off by writing about it in *develop*?

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

So don't just sit on those great ideas; feel the thrill of seeing them published in *develop*! To receive our Author's Guidelines, editorial schedule, and information about our incentive program, please send a message to develop@apple.com or AppleLink DEVELOP, or write Caroline Rose, Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.

Q *Can I assume that the value of the ColorSync CMWorldRef parameter returned by the NCWNewColorWorld routine isn't NULL if the routine was successful? I'd like to determine whether a color world exists before trying to use it.*

A Yes, you can assume that if no error is returned, the CMWorldRef parameter will be valid.

Macintosh Q & A

Q *Can ColorSync 2.0 profiles be embedded in EPS images?*

A Yes, you can embed ColorSync profiles into EPS, as well as into PICT and TIFF formats. For more information on how to do this, see the Macintosh Technical Q&A "Embedding ICC Profiles" (CS 06). There are also details on how to embed profiles in PICTs, along with sample code, in *Advanced Color Imaging on the Mac OS*, page 4-34.

Q *How can I determine whether SCSI Manager 4.3 has been loaded by an extension at startup time?*

A Drivers or applications that intend to call the asynchronous SCSI Manager must check that it's present by checking for the presence of the _SCSIAtomic trap (0xA089).

While this is sufficient by itself for applications, driver writers must keep in mind that if the asynchronous SCSI Manager is loaded as a system extension, the driver may be running before the _SCSIAtomic trap is installed (this is true for the startup device driver), so simply checking for the existence of the trap when your driver starts up isn't sufficient. The extension loading process needs to be completed before you make the check.

One way to do this is to use the accRun mechanism. If you set the dNeedTime flag in your driver, it will get an accRun call at SystemTask time. You can test for _SCSIAtomic then, set a flag indicating that you've discovered it, and then optionally reset the dNeedTime flag so that your driver doesn't get called after you've completed your discovery process. (Note that this technique is only suitable for old-style drivers, type 'DRVR'. New PCI-compatible drivers, type 'ndrv', can't use the accRun mechanism.)

This method isn't foolproof, however. It's possible for your driver to get an accRun call before all the extensions are loaded, such as when a dialog is presented from an extension that calls, for instance, ModalDialog (which will eventually dispatch SystemTask). This results in the driver receiving an accRun call even though there may be more extensions to follow.

So, in addition to simple determination of the presence of _SCSIAtomic, you need a way to test that the Finder (or some other application process) has been launched. You can do this by checking the length of CurApName. CurApName has a -1 length at startup and becomes positive when a process (such as the Finder) gets launched.

An alternative is to queue up a Notification Manager task without specifying an icon or sound, but only a response procedure. The procedure will get called after the extensions finish loading. You can then check for the _SCSIAtomic trap.

Q *I'm writing a QuickDraw GX printer driver that supports SCSI and PrinterShare (server) connection types. I can connect multiple printers to one Macintosh on the SCSI bus, and I've seen that I can have active print jobs printing on all of them simultaneously. My question is: Do I have to be concerned about reentrancy when coding my message overrides?*

A There are a few issues you'll need to keep in mind. One is that each copy of your driver must store any data it needs in its own data space. You can do this by using the GetMessageHandlerInstanceContext and SetMessageHandlerInstanceContext functions. If there is common global data that all copies of your driver will need to access, you can use the functions SetMessageHandlerClassContext and GetMessageHandlerClassContext. These are documented in Chapter 6 of *Inside Macintosh: QuickDraw GX Environment and Utilities*.

For each instance of your driver, you'll also need to watch out for insufficient memory. You shouldn't need to add much code if you're already checking for error conditions when attempting to allocate memory within your driver, but if there are places where you're not checking to make sure that the allocation was actually successful, you'll need to add code (it's a good idea to always check anyway).

You'll also need to confirm that you don't have multiple instances of your driver trying to write to the same printer at the same time. There are any number of ways you can confirm this, including using a shared (ClassContext) data block with a semaphore to mark whether an instance of your driver was in the middle of a GXWriteDTPData call. Each instance could then first check that semaphore before attempting to read or write data from the desktop printer. Be sure to include file locking while your driver is reading or writing other files.

Finally, if you're writing a PostScript driver, be aware that the PostScript font downloading code is not reentrant.

In general, you should use these techniques to write any QuickDraw GX printer driver, whether or not you expect it to need reentrancy.

Q *I want to program with Open Transport. What libraries should I link with?*

A Here are descriptions of the libraries that come with Open Transport, and why you might need to link with them.

Let's look first at linking Open Transport with PowerPC code. The basic libraries to link with are OpenTransportLib and OpenTransportAppPPC.o. If you need AppleTalk services, also link with OpenTptAppleTalkLib and OpenTptATalkPPC.o. If you need Internet services, also link with OpenTptInternetLib and OpenTptInetPPC.o.

The OpenTransportUtilLib and OpenTptUtilsAppPPC.o libraries may provide a smaller footprint if your application deals only with ports. Once you've got it working, try replacing OpenTransportLib and OpenTransportAppPPC.o with these and see if your application still links.

The OpenTransportExtnPPC.o and OpenTptUtilsExtnPPC.o libraries replace OpenTransportAppPPC.o and OpenTptUtilsAppPPC.o if you're writing a standalone code resource, CFM fragment, or ASLM shared library.

Note that if your code is meant to run on machines with and without Open Transport (that is, you revert to Classic AppleTalk or MacTCP if Open Transport isn't available), you should make sure to weak-link with the libraries ending in "Lib." Otherwise the system will refuse to launch your application when Open Transport isn't installed.

Now let's take a look at linking Open Transport with 680x0 code. The basic libraries to link with are OpenTransport.o and OpenTransportApp.o. If you need AppleTalk services, also link with OpenTptATalk.o. If you need Internet services, also link with OpenTptInet.o.

The OpenTptUtils.o library may provide a smaller footprint if your application deals only with ports. Once you've got it working, try replacing OpenTransport.o with this and see if it still links.

The OpenTransportExtn.o library replaces the OpenTransportApp.o if you're writing a standalone code resource or ASLM shared library.

In addition, the following libraries are identical to their similarly named counterparts except that they're suitable for linking with MPW model-near clients: OpenTransport.n.o, OpenTransportApp.n.o, OpenTptATalk.n.o, OpenTptInet.n.o, OpenTptUtils.n.o, and OpenTransportExtn.n.o.

Q *How do I use Open Transport with CFM-68K?*

A The short answer is that you don't, at the moment.

To get a better answer to this you must first specify what you want to do: use CFM-68K to build Open Transport modules (plug-ins like device drivers and protocols), or call the Open Transport API from a CFM-68K application or library (an OpenDoc part, for example).

Building Open Transport modules with CFM-68K is not currently supported nor is it likely to be supported. The Apple Shared Library Manager is your only alternative for building 680x0 modules.

Support for calling Open Transport from a CFM-68K application or library is likely to be incorporated in the next major release of Open Transport. This is in line with Apple's shared library strategy, as outlined in the DLL Statement of Direction document (ftp://seeding.apple.com//ess/public/aslm/DLL_directions), which indicates a move away from ASLM in general.

Q *I wanted an extension to load last in the initialization or startup sequence, so I put the backquote character (`) as the first character in its name. But when I did this, it actually loaded near the beginning. What happened?*

A If you view the contents of any folder by name, you'll see that items whose names begin with a backquote appear last (or nearly last) in the list. To sort the list, the Finder calls PACK 6, which uses the international sorting routines; these sorting routines order words beginning with a backquote near the end of the list.

During extension loading, however, PBHGetFInfo is called, and it in turn calls RelString. RelString is called with case insensitivity and diacritical sensitivity turned on. Unfortunately, due to a bug that HFS relies on, backquote sorts between *a* and *b*. This means that extensions beginning with a backquote load after extensions beginning with *a* (or *A*) and before those beginning with *b* (or *B*).

There are no plans to fix this problem, because of the need to maintain compatibility with old HFS volumes (which were created before this bug was discovered and which use this sorting order).

Interestingly, UpperString converts a backquote to an *a*, but leaves all other nonletters unchanged.

If you want to make sure that your extension is loaded last in the initialization process, use the tilde (~) as the first character of its name.

Q *When I'm printing and I call PrClosePage at the end of a page, I get a paramErr (-50) error. What does that mean?*

A There are two ways to get this error. The most common is to pass a bad graphics port to PrClosePage. If you don't pass back the port you got from PrOpenPage, you will (rightfully) get an error.

The second way to get paramErr from PrClosePage is more esoteric. If you've hidden the menu bar before printing, and you leave it hidden, some drivers will report a paramErr error when PrClosePage is called. What's happening is that somewhere deep in the Printing Manager, one of the Printing Manager routines is calling a QuickDraw function with the menu bar's rectangle as the parameter. This QuickDraw function sees the empty rectangle (because you've hidden the menu bar) and sets QDError to paramErr. The driver checks QDError when it's done printing, sees the error, and sets PrError to the error. Note that this happens only on 680x0 machines, not on PowerPC machines, which is ironic since 680x0 QuickDraw does much less error checking, and seldom sets QDError.

Q *Are there any tools to help me debug my QuickDraw 3D project?*

A During your development process, you should use the debugging version of QuickDraw 3D. To install it, just place the three extension files in the Extensions folder (in the System Folder) and restart your computer. The debugging version has a larger footprint and lower performance, but it has more extensive error checking. Using the debugging version will help you find problems with your code; it posts notices in addition to the errors and warnings posted by the optimized version. You'll also want to check out the 3Debug application; it graphically displays QuickDraw 3D memory usage, which can be very helpful. 3Debug and the debugging version of QuickDraw 3D are provided with the QuickDraw 3D release.

Q *I have a sample program compiled in MPW with Symantec's SC compiler. When I try to link the sample program, I get the following linker error:*

```
### Link: Error: Undefined entry, name: (Error 28) "qd"
Referenced from: main in file: :Obj 68K:FIFDECO.c.o
```

qd *is the QuickDraw global variable. If I declare the global, as in*

```
QDGlobals qd;
```

the error goes away. This is confusing, because the global is supposed to be declared for PowerPC code, but should automatically be declared for 680x0 files. In fact, this same code compiles and links correctly with Symantec C++ v7.0 IDE, as well as Metrowerks CodeWarrior. Is there some new library I need to include to get the 680x0 global declared? Or has some subtle change been made to the header files?

A Recently there has in fact been a change: The MPW libraries for the classic Macintosh runtime architecture now require that the QuickDraw global **qd** be defined in the global space of your code, the same as in the MPW libraries for the other Macintosh runtime architectures (namely, the PowerPC and CFM-68K runtime architectures). If you're working in the MPW environment, a simple definition such as the following is all that will be necessary:

```
QDGlobals qd;
```

If you're working in multiple environments (say, MPW, Metrowerks, and Symantec) use a preprocessor conditional such as this:

```
#if GENERATINGCFM
   QDGlobals qd;  // Required for all CFM environments
#else
#if !defined(SYMANTEC_C) && !defined(SYMANTEC_CPLUS)
   #define __MPW_ONLY__
#endif
#if defined(__SC__) && defined(__MPW_ONLY__)
   QDGlobals qd;  // Required for SC in MPW compilations
#endif
   #undef __MPW_ONLY__
#endif
```

For more details on the use of QDGlobals and **qd**, see Technote 1016, "Where Has my **qd** gone? And How Do I Use **qd** and QDGlobals Correctly?"

Q *I'm confused about the implementation of the AutoStart feature that was announced with the QuickTime 2.1 release. Can you tell me how to create a CD with this feature that will work cross-platform?*

A The AutoStart feature first documented in the release notes for QuickTime 2.1 has been available since the release of QuickTime 2.0. This means that any CD title you release today with an AutoStart application or document will work with most users' current installation of software, since QuickTime 2.0 has been available for well over a year. (Of course, there are other benefits to using QuickTime 2.1 over 2.0, but that's beside the point.)

The AutoStart feature is available only on HFS volumes, because it relies on information located in block 0 of an HFS disk or partition. The first two bytes in the sector of block 0 should be 0 or LK, although this realistically should be limited to 0 since LK designates an HFS boot volume. The name of the AutoStart file is stored in the area allocated for the Clipboard name. This area begins 106 bytes into the sector of block 0, with the first four bytes at that offset containing the hex value 0x006A7068. This value indicates that an AutoStart filename follows. After this 4-byte tag, 12 bytes remain, starting at offset 110. In these 12 bytes, the name of the AutoStart file is stored as a Pascal string, giving you up to 11 characters to identify the file. The file must reside in the root directory of the HFS volume or partition.

You may designate either an application or a document as the AutoStart file. If you choose an application, it may be visible or invisible in the root directory of the volume. However, document files must be visible. Additionally, you may select an alias file as the AutoStart file, but it too must be visible in the root directory of the volume or partition. If the AutoStart file is a document or an alias to a document, QuickTime will ask the Finder to launch the document as if it had been double-clicked from the Finder. If the creating application isn't available, the Finder will issue its normal warnings or use Macintosh Easy Open if available.

The real goal of the AutoStart feature in QuickTime is for users to be immediately engaged, upon insertion of a CD-ROM product, in an experience of the developer's choice, whether it's jumping right into a multimedia program or reading an important "ReadMe" document. Because there's no way for the user to bypass the launch of the AutoStart file (except for disabling QuickTime at startup), you, the developer, must determine what user experience you want to capture and decide whether or not the AutoStart feature makes sense for your project.

Creation of the AutoStart block 0 information is dependent on the CD-ROM mastering software that you use. Most of today's CD-ROM mastering software is capable of writing the AutoStart information. To be certain, however, it's best if you check with the developer of your CD-ROM mastering software if you aren't sure of its capabilities.

Note that the AutoStart feature wasn't implemented in QuickTime for Windows. This is primarily because new CD-ROM drivers are needed; the current drivers don't know when a new CD is inserted. When producing a cross-platform CD title, you'll need to create a hybrid disk that has an HFS partition if you want to use the AutoStart feature. You can create an HFS/ISO 9660 hybrid disk with your Macintosh project on the HFS partition and your cross-platform files on the ISO 9660 partition for use by both the Macintosh and PC main programs. Of course this means that you wouldn't have any AutoStart features on the PC platform unless you implemented a PC solution (like the Windows 95 auto-play feature) on the ISO 9660 partition.

Q *The kATAOfflineEvent and kATARemovedEvent documentation in the various Developer Notes for IDE-compatible Macintosh computers seems incomplete. Is it?*

A Yes, it is. Updated ATA Manager documentation is in progress. In the meantime, the documentation is supplemented here.

- kATAOnlineEvent (code 1) — This event notifies clients when an ATA or ATAPI device becomes available for use. The event occurs either when a new device is connected to the bus or when a previously unavailable device becomes available again (as in system wakeup when power is restored to the device).

  If the device has a registered driver, only that driver will be notified of the event; otherwise, each registered default driver will be notified until a driver responds favorably (that is, with a noErr response to the event). Note that for newly connected devices a driver loaded from the device is given priority.

  Drivers should keep track of whether the device coming online is a newly connected device or one that's currently offline (that is, connected but not unavailable). A device should be considered connected until a kATARemovedEvent event for the device occurs.

- kATAOfflineEvent (code 2) — This event notifies the registered driver of an ATA or ATAPI device that the device is now unavailable for use (offline). The device, however, is still connected to the bus and the offline state is assumed to be temporary. This event will occur at system sleep when power is removed.

  Currently, this event is generated only when the ATA Manager receives a PM_SUSPEND event (essentially the same as a Power Manager sleep demand event) from the PC Card Manager. Drivers receiving kATAOfflineEvent events most likely will want to maintain control of the device but deny access to the device from its clients. In addition, the driver should note that the device may need to be reconfigured when it comes online again (a kATAOnlineEvent event will be generated when this happens).

- kATARemovedEvent (code 3) — This event notifies the registered driver of an ATA or ATAPI device that the device has been removed. The removal may be either controlled (for example, a software eject command to the ATA Manager) or uncontrolled (or example, a forced removal by the user). Note that the device may have been in either an online or an offline state before the removal. If the state was online before the removal, a kATAOfflineEvent event is not generated, since the removal implies that an offline condition had to occur.

- kATAResetEvent (code 4) — This event notifies the registered driver of an ATA or ATAPI device that the device has been reset. The device may need to be reconfigured by the driver before it can be used again. This event was created for use with multiple devices per bus (ATA Master/Slave mode), since reset applies to all devices on the bus and not to a specific device. Apple currently doesn't implement multiple devices per bus with ATA, so this event isn't implemented. It's advised, however, that drivers support this event now to prevent problems later on when the event is implemented.

- kATAOfflineRequest (code 5) — This event is obsolete. It was defined for the early stages of the PC Card Manager which would echo the Power Manager sleep events to its clients. The ATA Manager would in turn echo the request to its clients. This event was like the sleep request event. The current PC Card Manager allows only for an event akin to a sleep demand event, which does not permit rejection by the client.

- kATAEjectRequest (code 6) — This event notifies the registered driver of an ATA or ATAPI device that a request has been made to eject the device. If the response to the request is 0, the device will be ejected and a subsequent kATARemovedEvent event will be generated when the ejection is successful. The kATAEjectRequest event serves as a protection mechanism to alert drivers of a pending ejection. Drivers will most likely want to reject the request unless they initiated the request, since ejection will remove the device from the bus.

Note also that the kATAResetEvent, kATAOfflineRequest, and kATAEjectRequest events are not currently implemented in the ATA Manager.

Q *How do I convert Macintosh Simplified Chinese encoding to the relevant GB standard?*

A The Macintosh encoding for Simplified Chinese is a shifted GB2312. To convert from GB2312 to Macintosh encoding, just add 0x8080 to each character. To convert from the Macintosh encoding to GB2312, subtract 0x8080 from each character. For example, an ideographic comma (Unicode code point 0x3001) is 0x2122 in GB, and 0xA1A2 on the Macintosh.

The only subranges of characters you need to worry about are the Roman characters. Below is some code that illustrates how to do the conversion.

```
// Returns true if the character needed conversion, or false if it was a
// one-byte character (meaning that only the first byte was processed).
// (That is, a false return means the character was a Roman character.)
boolean MacToGB2312(unsigned char first, unsigned char second,
                        unsigned short *output)
{
   if (first < 0x81) {
      *output = first;
      return false;
   } else {
      unsigned short temp;
      temp = (first - 0x80) << 8;
      temp += (second - 0x80);
      *output = temp;
      return true;
   }
}

// This will always convert, so we don't need to get the bytes separately
// nor do we need to return a Boolean saying whether we converted.
void GB2312ToMac(unsigned short input, unsigned short *output)
{
   *output = input + 0x8080;
}
```

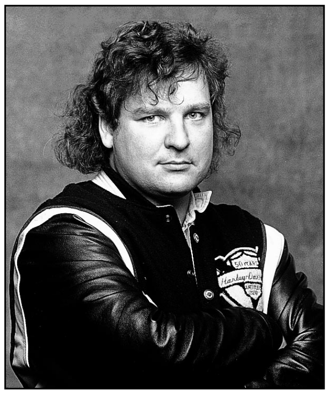
As you can see from the code, you need to modify both bytes of a two-byte character. This is done so that it's obvious whether a character is part of a two-byte character or is a one-byte Roman character.

For more information, see *Understanding Japanese Information Processing* by Ken Lunde. See also Ken Lunde's Web page at http://jasper.ora.com/lunde/; while it contains mostly information about Japanese text processing and standards, there are pointers to more information about Chinese and Korean information processing as well.

Q *I know that homonyms are words that are pronounced alike (and are often spelled the same) but have different meanings, and I know that synonyms are words that have the same or nearly the same meanings. But there are also words that are spelled alike but are pronounced differently and have different meanings. For instance, "row" (rhymes with "go") meaning things arranged in a line, and "row" (rhymes with "cow") meaning a fight. What do you call words like these?*

A Heteronyms.

These answers are supplied by the technical gurus in Apple's Developer Support Center. For more answers, see the Macintosh Technical Q&As on this issue's CD or on the World Wide Web at http://dev.info.apple.com/techqa/Main.html. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)•



JIM MENSCH

THE VETERAN NEOPHYTE

Manual Labor

As a 14-year Apple veteran, why would I be writing a "neophyte" column? It's true that I've written system software for many Apple computers and I've been in every tech support capacity that Apple has ever dreamed of. I'm writing this because I'm in fact still wet behind the ears; I learn new things every day here at Apple. Most of what I learn involves problem solving and debugging. I believe that creative problem solving is the one trait that separates a great programmer from an average programmer. Great programmers must hone their problem-solving skills all the time, whether at a computer or not. In fact, sometimes examining how day-to-day problems are solved can help us develop proper coding and debugging skills at work. Here I'd like to relate a little fable to illustrate this point (the names have been changed to protect my automotive pride).

A FABLE

Bob and Stu were working on a primo 1963 Dodge Dart that they had just bought. It was a classic: push-button transmission, cherry upholstery, straight body. A classic car with a classic problem: it had been run dry of oil and the engine needed to be rebuilt. Flush with confidence (having just completed an engine rebuilding course at the local community college), Bob said, "No problem, man. Let's rebuild it ourselves; how hard can it be?" That, I suppose, was the first mistake.

So, Bob and Stu set out to rebuild the engine, using all new parts. A scant two months later, the ten-hour job was finished. Finally they were ready to install the engine. Struggling to get the 1000-pound engine into the car and onto two very small bolts that hold it at an angle too obtuse to allow it to drop in straight, they were cussing and cutting themselves constantly.

After a while it dawned on them that things were not going as planned. In a moment of brilliance, Stu said, "Hey, what does the manual say about installing this thing?" The manual! Every good car mechanic has a manual and follows it. Why didn't I — er, Bob — think of that? Reading the manual, they found that the torque converter (a big heavy round thing with teeth on it) was supposed to be mounted not to the engine but to the transmission!

They pulled out the engine and struggled for an hour to get the converter mounted to the transmission. After this the engine went in relatively easily (with the help of Thom the helpful Brit). Having connected all the hoses, belts, doodads, and whatnots, our intrepid pair looked at each other with giddy anticipation. "Could it be that we're ready to start this thing?" they wondered. So they tried to start it — and they failed to start it. Just as with every major programming project, they had put in countless hours, and when the time came to fire the baby up, nothing happened.

It was time to debug this problem. "Hmm. Seems like a compression problem. Do we have compression?" asked Stu. "Yup, it's low, about 60%, but I guess that's because the engine hasn't been broken in yet," replied Bob. Were the plugs firing? Ground one side, turn out the lights, look for the blue spark: of course they were firing. Was the gas getting to the carburetor? Not yet, so they siphoned some up and tried again. Still no go.

So they started looking at the esoteric stuff. Was the timing chain on correctly? Well, we have compression, and it seems that we shouldn't if the timing is wrong. Three hours later, after Bob browbeat Stu into agreeing that the timing must be correct, Stu browbeat Bob into testing it anyway, since while Bob talked a good game it was always possible that he was wrong. They partly disassembled the engine and watched the little valve bits go up and down, and sure enough the timing was right. A victory for Bob, hollow as it may have been.

They continued to argue about what the problem could be, and finally decided to let it rest a while. This went on for weeks, until one day Bob remembered something that his teacher told him in class: a tablespoon of oil in every cylinder will get the seals sealing so that an engine could start. Could it be that easy? Could it be that the first thing that they had looked at — the compression — was in fact what was preventing the engine from starting? Bob put a little oil in each spark plug hole and

JIM MENSCH (mensch@applelink.apple.com, AppleLink MENSCH) has spent the last 14 years as a wage slave at Apple. Before that he did real work that involved cleaning and lifting and toting stuff and working with tools. While his mother is his real inspiration in life, he looks to the relaxed masses for guidance. An avid book collector and cook, he has absolutely no time for computers when he's not at work. His personal motto is "Eat more beets."•

the mighty engine roared to life! Our heroes stood dumbfounded at first, then quietly patted themselves on the back for such a fine job. Months after they had started their odyssey, they finally got the beast running.

YOU'RE NEVER TOO SMART TO READ THE MANUAL

The first thing to notice is that reading the manual was not the first step Bob and Stu took toward solving their problem. Neglecting to read the manual cost them hours of avoidable frustration and rework. Like *Inside Macintosh*, automotive manuals contain many hidden gems that are there for the asking. For instance, the shop manual didn't explicitly say, "Don't be a moron; the torque converter stays attached to the transmission!" but it did say, "Step 9. Remove torque flex plate screws, leaving converter attached to transmission."

Inside Macintosh contains many such tidbits waiting to be found. For instance, I was recently asked by a developer why a particular call to close a window wasn't causing the window behind it to redraw properly (it was leaving a desktop-patterned hole behind). Examining the problem a little further, we found that a resource that was needed had been purged and wasn't being reloaded. The code was smart enough not to crash when the purged resource was discovered, but it didn't seem to be able to reload the thing. As it turns out, the developer was using a rather strange strategy for manipulating the ResLoad attribute of the Resource Manager. He was turning it off when he wanted it off but not turning it back on again right away; instead he would turn it back on when he needed it on. I pointed out to the him that this was the problem, and he said, "I've been programming the Mac for almost 10 years and I've never read anywhere that the Window Manager assumes ResLoad is TRUE!"

While he's right about this on the surface, if we look we find that *Inside Macintosh* warns in the SetResLoad description that "If you call SetResLoad with the load parameter set to FALSE, be sure to . . . set [it] to TRUE as soon as possible. Other parts of system software that call the Resource Manager expect this value to be TRUE." (This has been in there so long that Caroline Rose wrote the first draft of it!) Since the WDEF is system software, it assumed that ResLoad would in fact be TRUE. After I pointed this out, the developer decided it was time to break out those manuals that had been collecting dust for so many years, and revisit some of the documentation he thought he had remembered.

YOUR FIRST INSTINCT IS USUALLY THE BEST

Another lesson to learn is that when problems arise, don't spend hours plodding through esoteric logic. First think, "What's the most obvious cause of this problem?" You might recall that Stu was right when he suspected a compression problem. By discarding the obvious without first examining it fully, we risk costing ourselves days of work only to find out that we were right all along. As a seasoned programmer, you'll learn that you can get a feel for why a problem exists. You may not have any ready logic to explain why an event occurred, but you might have a feeling anyway. Go with that feeling. The obvious things are the easiest to check (but don't make it too easy and stare past the real trouble). They're also usually the quickest to fix. Looking there first can save time, effort, aggravation, and lots of cussing. In my 14 years of problem solving for Apple, I've found that the simple, obvious solution is right 90% of the time.

CONFIRM YOUR LOGIC WITH REAL EXAMPLES

Why did Bob and Stu retest the timing, anyway? Experienced troubleshooters, they realized that simply arguing about a point may lead to a conclusion, but any conclusion that *can* be tested *should* be. Logic dictates that the cam shaft can be 180° out of phase (for every rotation of the cam shaft, the crankshaft rotates twice and thus will be in the same position at 0° cam rotation as 180° cam rotation), but you should check it anyway if you're stuck. It's easy to get your logical conclusions backwards and get your crankshaft 180° out of phase. On a car this might result in outright failure, while on a computer it can mean more insidious things.

I recently needed to use the Power Manager to control screen dimming, drive spindown, and CPU sleep. All of the calls had a "get" function that returned a Boolean, TRUE if the feature was on or FALSE if it was off. Strangely enough, two of the "set" calls required a value of TRUE to enable the feature but one required TRUE to *disable* the feature. I spent hours looking at the complex logic of IF statements before I simply watched it all go by in MacsBug and noticed that I was sending a TRUE value to a call named Disable. Going back through my logic again brought the error right out.

BACK TO WORK

If you've learned anything from the above tale, good. Remember, manuals contain quite a lot of information and they're a good place to start. Also remember that this isn't rocket (or automotive) science. Think simple thoughts; don't create extra work for yourself.

If you have any questions about why writing system software is like nailing your head to a board, or why your car grinds at high speeds, drop Jim a line.•

Thanks to Meg Bailey, Cameron Birse, Brian Hamlin, Mark Harlan, Bo3b Johnson, and Dave Johnson for reviewing this column.•

Newton Q & A: Ask the Llama

Q *I have a slip in my application that edits part of my application preferences. I use GetAppPrefs to get the preferences frame, and then set a pointer to a subframe in my slip:*

```
myAppPref.viewSetupFormScript := func()
begin
   local prefs := GetAppPrefs(kAppSymbol, kDefaultPrefFrame);
   self.target := prefs.defaultNames;
   inherited:?viewSetupFormScript();
end
```

The user makes the change and I use EntryChangeXmit, but sometimes I lose the change. Any hints?

A It looks as if you're encountering an interaction between soup entries and garbage collection. In your viewSetupFormScript, you use GetAppPrefs to load the soup entry corresponding to your application preferences into the NewtonScript heap. Then you set a target slot in your preferences slip to the defaultNames slot in that preferences frame. I assume that some time later, probably in the viewQuitScript, you reload your preferences frame (with GetAppPrefs again) and call EntryChangeXmit on the frame returned by that call.

The problem occurs because you use a local variable to point to your preferences entry. Once the viewSetupFormScript is completed, this local goes away, so the preferences entry is subject to garbage collection. This may seem unintuitive since soups are where you store persistent data. However, there's a difference between the data that comprises an entry in a soup and an actual entry in the NewtonScript heap. When you request a soup entry, the data from the soup is swapped into the heap so that you can access it as a frame. The entry on the heap is a copy of the data in the soup, not the real data. Changes to that heap copy aren't written to the soup until you call EntryChangeXmit.

After your call to GetAppPrefs, the heap looks like Figure 1. Then your viewSetupFormScript returns and the **prefs** local goes away, so your heap looks like Figure 2.

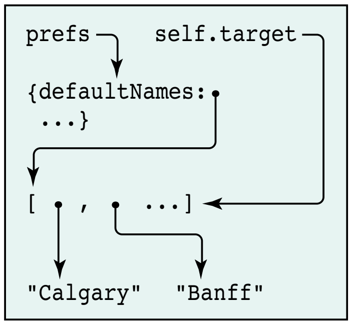


Figure 1. Heap after GetAppPrefs

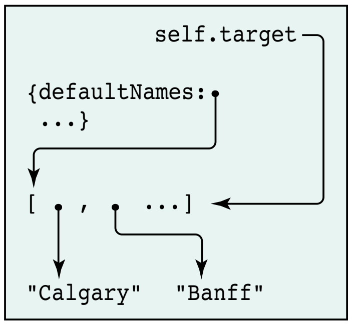


Figure 2. Heap after viewSetupFormScript

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-related questions to dr.llama@applelink.apple.com (AppleLink DR.LLAMA). The first time we use a question from you, we'll send you a T-shirt.•

Notice in Figure 2 how there is nothing referencing the preferences entry, but there is something referencing the defaultNames subobject. As far as the system is concerned, the preferences entry frame is now available for garbage collection. The next time the preferences entry is loaded in, an entire new copy of that entry is made, including the defaultNames subobject. So **self.target** points to a valid NewtonScript array that's different from the new copy of your preferences entry.

This explains why the information doesn't get updated, but not why this doesn't happen every time. It doesn't happen every time because the soup system will cache the frame representation of a requested entry. When you request an entry, the first thing the soup system does is check for a cached entry. If it exists, it's used, in which case the defaultNames subobject is the same one that **self.target** is referencing — that is, no new copy of the preferences entry is loaded into the heap.

So, what happens is that once the user finishes editing the entry, you call GetAppPrefs, which may return the cached preferences entry. If garbage collection has occurred, your target slot will point to the edited version of the defaultNames structure, but not to the defaultNames slot value from the new preferences frame. Figure 3 shows the heap after garbage collection has occurred. After your call to GetAppPrefs, you get the situation shown in Figure 4. Your local **prefs** variable points to a new heap copy of the preferences entry, but your target slot points to the old defaultNames value. The EntryChangeXmit call will affect the new copy of the soup entry, leaving it apparently unchanged.

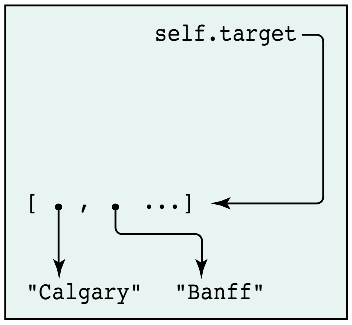


Figure 3. Heap after garbage collection

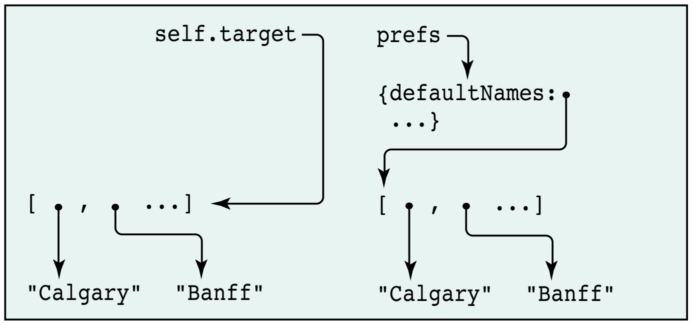


Figure 4. Heap after GetAppPrefs

There are two ways to fix this: you could put the edited defaultNames structure into the preferences frame before calling EntryChangeXmit, or you could hold a reference to the application preferences in your slip (or in your base view) for the duration of the edit. The first way is more memory efficient.

The lesson is that keeping around references to objects inside soup entries is a dangerous practice. The safe thing to do is read in your entry, do the modifications, save the entry, and set the reference to nil.

Q *I have an application that may print or fax many pages of information. I need to draw a lot of the content of those pages. I know that in 1.x viewDrawScripts, faxing needed to be fast. How about in 2.0? Are there better ways to go?*

A The main thing missing in Newton 1.x OS is a method that gets called before the fax connection is made. In Newton 2.0 OS, the formatInitScript method of your print format will be called before the connection is made. You can use this script to do time-consuming drawing and cache the results for later use.

An extension of this technique is to render all your pages into a Virtual Binary Object and then access the appropriate place in that object during printing or faxing. The advantage of this is that you save heap space, since a VBO is paged in to a system heap (not the NewtonScript heap). For some devices this is the only way to print large numbers of pages.

Q *The setup application uses some nifty embedded keyboards. I checked the beta version of the Newton Programmer's Guide for a prototype, but there doesn't appear to be one. Is this an oversight? How can I make these keyboards?*

A There are several, as yet undocumented, ROM prototypes for embedded keyboards. They will appear in the final *Newton Programmer's Guide*, but for now they're in the Newton Toolkit platform file:

- protoAlphaKeys — alphanumeric keypad
- protoNumericKeys — numeric keypad
- protoTouchtonePad — minimal phonepad
- protoPhonePad — phonepad plus punctuation and arrow keys

All of these embedded keyboards will send input to the current key view. All you have to do is draw one in your layout and make sure the target view is the current key view. See the Newton DTS Q&A document on the Newton Developer CD for instructions on how to use an afterScript to set the proto of a view.

Q *I'm porting my code from Newton 1.x OS to Newton 2.0 OS. When I build my project using Newton Toolkit 1.6 and the 2.0 platform file, I get an error telling me that k<insertNameHere>Func is undefined. What's the problem?*

A The chances are that your 1.x code is using one of the platform file functions that either have been incorporated into ROM or are obsolete. However, developers may want to write code that works on both Newton 2.0 and 1.x devices. To enable this, we provide the old functions but we mark them as *deprecated*, which means they shouldn't be used in Newton 2.0–savvy applications, but can be used for compatibility reasons.

For example, in 1.x platform files there's a kRegisterCardSoupFunc function; in the 2.0 platform file, this is called kRegisterCardSoupDeprecatedFunc since there's a new and better way to register soups in Newton 2.0 OS. See the platform file release notes for a list of deprecated functions, protos, and so on.

Q *How are reals represented in the package format? The data field for 12.345 is represented as 0x4028B0A3D70A3D71, for example.*

A NewtonScript uses the Apple SANE **double** format (basically the IEEE format) for floating-point numbers. These are implemented as 8-byte binary objects of class **real** and contain a sign bit, 11 bits of biased exponent, and 52 bits of fraction.

0x4028B0A3D70A3D71 is the binary data (8 bytes) of the SANE representation of 12.345. It's the same data that's used to hold the number on the Newton itself.

Q *I would like to add a separator line followed by some new application-specific actions. I proceeded to register a frame with the title as the symbol pickSeparator. It worked, except that the separator was selectable. All I'm trying to achieve is an eye-pleasing separation between the system actions and my actions. I also tried returning in the GetTitle routine a frame with*

```
{item: 'pickSeparator, pickable: nil}
```

but that resulted in a blank entry. Is there a way to do what I'm trying to do — that is, to have a pickSeparator that isn't selectable in the action button?

A In the routeScripts array, you can use a nil value instead of a frame. That will add another pickSeparator at that position in the routeScripts. Note that the system will fill in the separator between the items that are routing transports (like Print) and the items that are actions (such as Delete). If you need to add this separator dynamically, you can provide your own GetRouteScripts method that dynamically returns the routeScripts frame.

That said, please check the latest *Newton 2.0 User Interface Guidelines* to make sure that you're putting a separator in a valid spot.

Q *I'm profiling my application to see why it takes so long to open. However, of the time it takes to open, only a small percentage is spent in my code. I'm measuring from the start of the viewSetupFormScript to the end of the viewSetupDoneScript in my base view.*

A There are a few things you can do. The first is to make sure you're profiling system functions to see if that's where the time is going. It may be that you're doing things in your startup process that would be better done at a later time.

You may also be running into low-memory conditions. Run the HeapShow utility that comes with Newton Toolkit and look at the frames heap and free system space (handles and pointers). You can do this in combination with NS Debug Tools to step through your code and track memory usage. Note that the Newton Toolkit inspector will use a fair bit of system space, so you may want to get a baseline memory usage without the inspector connected.

Q *I'm having two problems with a protoPicker view. First, when I open a protoPicker view (whose vFloating flag I haven't turned off) it's obscured by a textButton in the main view. I can't figure out why it doesn't float over this plain vanilla textButton.*

Also, I can't select some of the items in the picker. The inaccessible items appear last in the list, from the portion of the picker view that extends beyond the picker's parent view or the slip's main view.

The only unusual thing I can see here is that the protoPicker view is not a sibling of the textButton. The view hierarchy looks like this:

```
slipMainView
   clusterView
      protoPicker
   textButton
```

Can you help?

A It looks like the protoPicker is being opened as a child of the clusterView. This means that the active (tappable) area of the protoPicker will be clipped to the bounds of the clusterView. It also sounds like the clipping viewFlag of the clusterView isn't set. That allows the protoPicker to be drawn outside of its parent, so you may think it's clickable even when it isn't.

There are three possible solutions:

- Resize the protoPicker so that it's no larger than the clusterView.
- Make the protoPicker a child of a view higher up in the hierarchy (for example, the slipMainView).
- If your protoPicker is larger than the application base view, use BuildContext to attach it to the root view.

Q *How do I reset a protoTextList so that when I change the listItems and redisplay, the display starts at the first item again? Right now if I've scrolled the text list and then I reset it, the top item is wrong.*

A The documentation mentions a SetupList method that you call when you initialize the view. However, this is not enough if you're changing the listItems after you've opened the textList. Since the current implementation of textList scrolls by offsetting the origin, you also need to reset the origin.

Here's a method that you can add to your own protoTextList that will add a text item and redraw the list correctly:

```
myProtoTextList.AddItem := func()
begin
   // make sure listItems is an array
   if NOT listItems then
      listItems := [];

   local newItem := GetRandomWord(5, 10);
   // insert in sorted order for strings
   BInsert(listItems, newItem, '|Str<|, nil, nil);

   // redisplay based on new data
   // this will reset the list to the top item
   :SetOrigin(0, 0);
   :SetupList();
   :RedoChildren();
end;
```

Q *Can you write a funny Q&A ?*

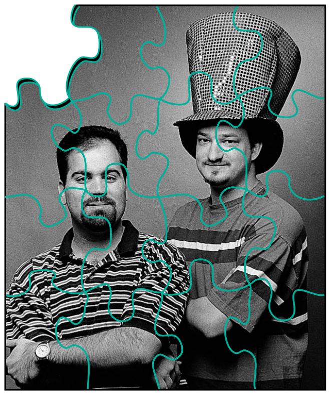
A Yes.

Thanks to jXopher Bell, Henry Cate, Bob Ebert, Mike Engber, David Fedor, Ryan Robertson, Jim Schram, Maurice Sharp, and Bruce Thompson for these answers. Special thanks to Bob Ebert for the answer on the format of reals.•

If you need more answers, check out http://dev.info.apple.com/newton on the World Wide Web or look at Newton Developer Info on AppleLink.•

New World Order

See if you can solve this programming puzzle, presented in the form of a dialog between Cameron Esfahani (cam) and Alex Rosenberg. The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.


CAMERON ESFAHANI AND ALEX ROSENBERG

Alex	Hey cam, Marathon crashed in a weird manner when I tried to play it under an early version of Mac OS 8.
cam	Working hard, eh? I suppose you'll tell me this was compatibility testing.
Alex	Yeah, well, it makes a good demo.
cam	Hey, wait a minute! If it's an early version of Mac OS 8, Puzzle Page readers won't ever find this bug, and they'll write nasty letters to the editor about it.
Alex	Well, they should know that they'll learn some valuable debugging techniques that they can apply to their own programming conundrums. Don't they read the intro to the Puzzle Page?
cam	I guess not. Anyway, back to your problem. So, in what way does it fail?
Alex	Just after launch, the machine freezes. This happens every time; it's 100% reproducible. I can't seem to get into MacsBug.
cam	Philistine! We use the one true debugger, the Macintosh Debugger for PowerPC. Mac OS 8 debugging is generally done from a second machine over a serial cable. You're probably frozen because the program has crashed and the debugger has halted the machine, waiting to start a debugging session.
Alex	What? I thought I gave that up with my Lisa. This is a Macintosh, after all.

CAMERON ESFAHANI (cameron_esfahani @powertalk.apple.com, AppleLink DIRTY) SWM, 24, 5' 7", 180 pounds, brown hair, brown eyes. Apple engineer. Loves movies and music. Plays golf and tennis, rollerblades, ice-skates, and is learning to ski again. Enjoys life but has a serious side. Likes cooking, reading, shopping. Once a dog guy, now a definite cat man. Believes the American musical of the '50s and '60s to be the second greatest invention of the twentieth century. Favorites include *West Side Story, The Sound of Music, Music Man,* and *Singin' in the Rain.*•

cam Kernel-based operating systems are typically developed with two-machine debuggers. Besides, think of the wonderful third-party opportunity!

100 Alex Um, yeah. Anyway, you've hooked up the serial cable and are running the debugger on the second machine. After watching a progress bar for a while, you see a dialog that says "Access Fault."

cam An access fault is caused by an attempt to access an illegal address. The PC is at the instruction that caused the fault.

Alex There's a Show PC command in the debugger's Extras menu. It puts me at 0x626FDE50.

cam Right. We need to isolate whether this fault occurred in application code or in the system. Choose Show Fragment Info from the Views menu and type that address into it.

Alex I can't type anything; the machine is crashed. Oh, I get it: I have to keep switching my head back and forth between machines like a spectator at a tennis match. What fun. So, how long does this barber pole thingy spin for, anyway? Hey look, the Fragment Info window highlighted the Marathon code fragment. The PC is in Marathon's code.

cam What does the code around the PC look like?

95 Alex It looks like this:

```
   626FDE34   mflr     r0
   626FDE38   stw      r0,0x0008(SP)
   626FDE3C   stwu     SP,-0x0038(SP)
   626FDE40   lwz      r4,0x0000(r3)
   626FDE44   lha      r3,0x0000(r4)
   626FDE48   bl       _eGetDCtlEntry
   626FDE4C   lwz      RTOC,0x0014(SP)
*  626FDE50   lwz      r12,0x0000(r3)
   626FDE54   lbz      r3,0x0028(r12)
   626FDE58   extsb    r3,r3
   626FDE5C   lwz      r0,0x0040(SP)
   626FDE60   addic    SP,SP,56
   626FDE64   mtlr     r0
   626FDE68   blr
```

R3 is the return value from the function call to _eGetDCtlEntry and apparently contained a bad address.

cam Choose Show Registers from the Views menu. This will show all the registers of the current process.

90 Alex It looks like the return value for _eGetDCtlEntry was 0. The **lwz** instruction is dereferencing R3 and putting the result in R12.

cam If you select the R12 register and choose Show Memory from the Views menu, you can see the memory at that address.

ALEX ROSENBERG (alexr@bungie.com) Alex's left brain works on everything from communications software to the latest 3D graphics tricks. His right brain is constantly thinking up interesting T-shirts that Apple's Marketing folks don't approve of. While working as a member of the Mac OS 8 "Ministry of Information," he experimented with optimization for PowerPC, worked closely with Apple's compiler team, and contributed to IBM's *The PowerPC Compiler Writer's Guide.* Now one of the minions at Bungie Software, Alex recently decided that eating is overrated.•

85 Alex That entire area of memory is full of 0xEEEEEEEE's.

cam That's unmapped memory. Is _eGetDCtlEntry the internal name of the routine GetDCtlEntry?

Alex Yes, the debugger is able to pick up that name by using a "trace-back table," which is the PowerPC equivalent of MacsBug symbols. I guess the next step would be to figure out why GetDCtlEntry is returning nil. What is it supposed to be doing?

cam According to *Inside Macintosh: Devices*, GetDCtlEntry returns the device control entry for the device specified by the value passed in refNum. If we look at the rest of the code in this function, right before we call GetDCtlEntry, we seem to be getting the refNum from the first 16 bits of some "handle" (or some other kind of pointer to a pointer), which is getting passed into this function.

80 Alex All right, but we're going to have to restart. Any information passed into this function has been lost because we're after the call to GetDCtlEntry.

cam To restart we'll need to remember the offset into the Marathon fragment where the fault will occur, because the Marathon fragment could be loaded in a different address range.

75 Alex The offset can be calculated by subtracting the faulting address from the beginning address for the fragment, which was shown in the Fragment Info window. For this address, the offset is 0x4832C.

cam Right, but we'd like to get control a little before the actual crash. The offset to the beginning of that function is 0x48310. Restart the system, and hold down the Control key when you relaunch Marathon. On a debugging system, this will break into the debugger after it has completely loaded the application but just before it begins to execute it.

Alex All right. The machine seems to have stopped at that point. The new start of the Marathon code fragment is 0x6337D6A0. Adding in the offset of 0x48310, we get an address of 0x633C59B0.

cam Bring up the Show Instructions window and enter 0x633C59B0 as the address. It will be exactly the same code as what we saw before. Set a breakpoint at the first instruction in this function — the **mflr** instruction — and run.

Alex We've reached that breakpoint.

cam Do a stack crawl and see who called us. Head over to the ever useful Views menu; there's a Show Stack Crawl command.

70 Alex All right. Apparently the caller is address 0x633988E0.

cam OK, let's step through this function and see what they end up passing to GetDCtlEntry for the refNum.

65 Alex It looks like they're passing in 0 for the refNum.

cam Well, there's your problem: 0 is not a valid refNum. It seems that they're getting an invalid refNum from some part of the system and passing that to GetDCtlEntry. GetDCtlEntry is returning nil and we're crashing by dereferencing nil.

Alex Uh, that's great, but I still can't play Marathon.

cam Where does the caller of this function, 0x633988E0, get the "handle" from?

60 Alex I'll bring up an instruction window at that address:

```
63398778   mflr      r0
...
633987B0   lwz       r24,-0x0218(RTOC)
...
633988D8   lwz       r3,0x0000(r24)
633988DC   addic     r3,r3,0x001C
633988E0   bl        $+0x2D340                ; 0x633C5C20
633988E4   nop
633988E8   stw       r3,0x0000(r30)
```

R3 seems to be loaded from a global. Let's figure out where this global gets initialized. The "handle" lives inside a structure that's pointed to by the global at -0x218(RTOC). This pointer has the "handle" stored at offset 0x1C within it.

cam We could try to track down where the field at offset 0x1C gets initialized. A pointer wouldn't move around, so we wouldn't have to worry about relocation. We can use the Data Breakpoint window feature of the Macintosh Debugger. The PowerPC 601 has a special register that allows you to stop execution whenever a specified address is read or written to. It's like hardware support for our old friend step spy.

Alex Sounds like a plan. So, I bring up the Data Breakpoint window and will break whenever someone writes to that address.

cam Of course, you realize that just as the code fragment could be loaded in a different place each time it's launched, the RTOC could have a different value as well.

Alex Good point. I'll be sure to use the new RTOC value when I restart Marathon.

cam What happens after we set up the data breakpoint?

55 Alex It's kind of strange. We seem to be stopping a lot, but people aren't writing to offset 0x1C in this structure; they seem to be writing 32 bits to offset 0x1A and overwriting 0x1C.

cam I don't understand. The routine that called the crashing routine was passing in a value at offset 0x1C.

Alex Apparently we calculated something wrong.

cam I don't know where we could have gone wrong. Wait a second. Look at address 0x633988E0; it says we're branching to address 0x633C5C20, but the routine we're crashing in is at 0x633C59B0.

50 Alex Well, maybe it's just a call to a different code fragment, a "cross-TOC" call, and it has to use some indirection to get to the crashing function.

cam No, it can't be a cross-TOC call, for two reasons: first, there's no TOC reload after the function call, and second, the routine we crashed in is in the Marathon fragment. You saw the Fragment Info results.

Alex I'll buy that. Now let's set our breakpoint just before this function call to the unknown address. We can step through that code.

cam All right. After we hit the breakpoint, step into that function.

45 Alex Holy cow! It's some totally different piece of code.

cam Look through the instruction disassembly of this new routine. Is there anywhere in there where they call the crashing function?

```
   633C5C98   bl         0x6337E150
   633C5C9C   lwz        RTOC,0x0014(SP)
   633C5CA0   ori        r31,r3,0x0000
   ...
   633C5CDC   ori        r3,r31,0x0000
*  633C5CE0   bl         0x633C59B0
```

40 Alex Yeah. At address 0x633C5CE0, they call our crashing function with a parameter obtained from R31. Working further back in the instruction disassembly, we see that a function call is made and the result of that is put in R31. This occurs at 0x633C5C98. It calls a routine at address 0x6337E150.

cam And looking at that, it appears to be a cross-TOC call. I restart Marathon and step into the routine at 0x6337E150.

Alex It appears to be cross-TOC glue:

```
   6337E150   lwz        r12,-0x0A90(RTOC)
   6337E154   stw        RTOC,0x0014(SP)
   6337E158   lwz        r0,0x0000(r12)
   6337E15C   lwz        RTOC,0x0004(r12)
   6337E160   mtctr      r0
   6337E164   bctr
```

cam So apparently Marathon is calling another library to get this mystical "handle."

35 Alex Yep. Whenever you're going to go from one library context to another, you need to save and restore the TOC. That's one of the things this glue code does. As you can see, R12 is loaded from -0x0A90(RTOC). R12 will contain a pointer to a transition vector, which contains an address of a routine and a new TOC value. The transition vector is imported from the library we're linking against.

cam So we should be able to plop the transition vector address into the Fragment Info window and figure out which library that comes from, right?

30 Alex Good idea. Dumping the address at -0x0A90(RTOC) we get the following:

```
0200CC0C: 01FC9D98 01FC9DA4 01FC9DB0 01FC9DBC
0200CC1C: 01FC9DE0 01FC9DEC 01FC9D80 01FC9D74
...
```

cam I use the Fragment Info window to find out which fragment contains the address 0x01FC9D98.

25 Alex It seems to live in the QuickDraw data section, which makes sense, since a transition vector is data.

cam Aha! QuickDraw! That figures. And you wondered why they call it KON & BAL's Puzzle Page. I use the Show Exports button in the Fragment Info window to list all of the exports of the QuickDraw library.

Alex I was wondering when we were going to use that button. You end up getting a long list of all the routines exported by QuickDraw, sorted alphabetically.

cam But I have an address I want to match. If you click on the Address column title, that list will get resorted by address. I search through the list for address 0x01FC9D98.

20 Alex That address is the address of the GetDeviceList transition vector.

cam That makes sense. This “handle” we’ve been worrying about has a refNum in the first 16 bits of the structure, and a GDHandle has the refNum of the associated driver stored in the first field in the structure. I bring up a memory window and examine the device list stored in low memory to see if the gdRefNum is 0.

Alex It is. Who’s responsible for initializing the GDevice record?

cam NewGDevice and InitGDevice. NewGDevice will pass the refNum to InitGDevice. Let’s disassemble the code for NewGDevice.

15 Alex Apparently it does a NewHandleClear to allocate the GDHandle, and never initializes the refNum.

cam Whoops. Ah, the joys of pre-alpha software. Well, it should be reasonably easy to get one of the QuickDraw engineers to fix this bug. I install a fixed version of the QuickDraw shared library. We should be rockin’ now!

Alex Not so fast! When I restart Marathon, I crash. If I do a stack crawl and examine the code, I seem to be crashing in exactly the same place. GetDCtlEntry still seems to return nil.

cam Just another day at the salt mines. OK, it’s time to step through GetDCtlEntry. I put a breakpoint just before we call it.

Alex The refNum from the GDHandle is -51.

cam That looks like a valid refNum. Step into GetDCtlEntry.

Alex We first go through the cross-TOC glue and eventually get into GetDCtlEntry.

cam What does GetDCtlEntry look like?

10 Alex It seems fine, but when you step through the routine and actually fetch the DCtlHandle from the unit table, it ends up being nil.

```
...
626FC79C   cntlzw     r3,r3
626FC7A0   srwi       r3,r3,5
...
```

cam That obviously shouldn’t happen. There’s a driver entry in the table and the refNum seems valid. What is the code at 0x626FC79C doing?

5 Alex It’s performing a logical NOT operation on R3. Groovy, huh?

cam But the bitwise NOT of the refNum should be used as the index into the unit table, not the logical NOT!

Alex So, you claim that GetDCtlEntry is looking at the wrong place in the unit table to get the DCtlHandle. Are you sure?

cam Let’s go to the source. What does *Inside Macintosh: Devices* say?

Alex It says, “The device reference number is the one’s complement (logical NOT) of the unit number.” But the logical NOT isn’t the one’s complement; the bitwise NOT is.

cam Um, OK, what does *Inside Macintosh* Volume II say?

Alex It claims that the unit number is “equal to -1 * (refNum + 1).”

cam And that’s a bitwise NOT. So it seems that *Inside Macintosh: Devices* is wrong. Weird, wacky stuff. But I still don’t understand why the stack crawl we did earlier pointed us to the wrong place.

Alex We did the stack crawl when we had just entered the crashing function, even before it executed the **mflr** instruction. The debugger, when it does a stack crawl, is going to look for stack frames to see where the callers are. Since we hadn’t allocated a stack frame in the crashing function yet, we were still using the caller’s stack frame. So that was the stack frame the debugger started from. If we had stepped a few more instructions in and allocated our stack frame, the debugger would have figured it out. It would be interesting to see if the debugger could actually detect the case of a nonexistent stack frame and use the link register to work back to the caller.

cam That also explains why the data breakpoint stuff didn’t work. We thought that the data structure we were watching held the GDHandle. It didn’t; it contained something totally unrelated, which was passed into the function that called the crashing routine.

Alex So, because NewGDevice didn’t initialize the gdRefNum and GetDCtlEntry was returning the wrong entry from the unit table, I don’t get to teach the Pfhor about large caliber, high-velocity rounds.

cam Nasty.

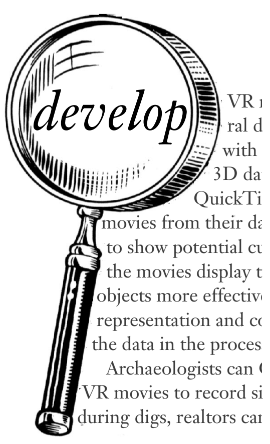
Alex Yeah.

SCORING

Your performance compares to these memorable screen roles:

80–100	Chuck Heston in *El Cid*
55–75	Anne Parillaud in *La Femme Nikita*
30–50	Chow Yun-Fat in *Hard Boiled*
5–25	Richard Roundtree in *Shaft* ●

Thanks to Tom Dowdy, Wayne Meretsky, Mike Neil, Tom Saulpaugh, KON (Konstantin Othmer), and BAL (Bruce Leak) for reviewing this column.●



Looking to complete the set?

If you're looking for a complete *develop* collection, full-color, bound copies are available for $13 per issue, including shipping and handling. (Back issues are also on the *develop Bookmark* CD and the *Developer CD Series* Reference Library edition, as well as on AppleLink and the Internet.) For more information about how to order printed back issues (and where to find them online), see the inside front cover of this issue. *Supplies are limited. Please allow 4 to 6 weeks for delivery.*

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7; Debugging Declaration ROMs

Issue 2 C++ (Objects; Style Guide); Object Pascal; Memory Manager; MacApp; Object-Based Design

Issue 3 ISO 9660 and High Sierra; Accessing CD Audio Tracks; Comm Toolbox; 8•24 GC Card; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Palette Manager; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; TextBox You've Always Wanted; Sound; Terminal Manager; Debugging Drivers

Issue 10 Apple Event Objects; Enhancements for the LaserWriter Font Utility; GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 Asynchronous Routines; QuickTime and Components; Debugging; Color Printing; DeviceLoop

Issue 14 Localizable Applications; 3-D Rotation; QuickTime (Video Digitizing; Making Better Movies)

Issue 15 QuickDraw GX; Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Proto Templates on the Newton; Standalone Code on PowerPC; Debugging on PowerPC; Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Issue 19 OpenDoc Part Handlers; PowerPC Memory Usage; Designing for the Power Macintosh; QuickDraw GX (Printing; Bitmaps); Inheritance in Scripts

Issue 20 AOCE; Make Your Own Sound Components; Scripting the Finder; NetWare on PowerPC

Issue 21 OpenDoc Graphics; Dylan; Designing a Scripting Implementation; Object-Oriented Hierarchical Lists; Introducing PowerPC Assembly Language

Issue 22 QuickDraw 3D; Copland; PCI Device Drivers; Custom Color Search Procedures; The OpenDoc User Experience; Futures

Issue 23 QuickTime Music Architecture; QuickDraw 3D Geometries; Internet Config; Multipane Dialogs; Document Synchronization; ColorSync 2.0

Issue 24 Speeding Up **whose** Clause Resolution; OpenDoc Storage; Sound; Alert Guidelines; Printing Images Faster With Data Compression; The New Device Drivers and Memory

Issue 25 Generating QuickTime VR Movies From QuickDraw 3D; Flicker-Free Drawing With QuickDraw GX; NURB Curves; C++ Exceptions in C; Localized Strings for the Newton

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to develop@apple.com or to:

Caroline Rose
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
Internet: crose@apple.com
AppleLink: CROSE
Fax: (408)974-9423

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
Internet: dkj@apple.com
AppleLink: JOHNSON.DK
CompuServe: 75300,715
Fax: (408)974-9423

Please direct all subscription-related queries to Apple Developer Catalog, P.O. Box 319, Buffalo, NY 14207-0319 or to order.adc@applelink.apple.com (AppleLink ORDER.ADC). You can also call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, or (716)871-6555 elsewhere.

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. The Apple Developer Catalog and Apple Developer University are open to anyone who wants access to development tools and instruction. Additional information and services are available through Apple's Developer Programs.

The Apple Developer Catalog offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. This complimentary catalog features hundreds of Apple and third-party development products and offers convenient payment and shipping options, including site licensing.

Apple Developer University (DU) provides courses to get you started programming on Apple platforms and Mac OS–compatible hardware, as well as advanced, in-depth training on new technologies such as QuickTime VR, QuickDraw 3D, OpenDoc, Apple Guide, and Newton. In addition to classroom training, self-paced courses and tutorials are available through the *Apple Developer Catalog*.

The Macintosh Developer Program provides members with ongoing Macintosh-related technical information and services. It includes:

- The monthly Apple Developer Mailing, which includes the *Developer CD Series*.
- Macintosh technology seeding.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Newton Developer Program provides ongoing Newton-related technical information and services. It includes:

- The monthly Newton Developer Mailing.
- The quarterly Newton Developer CD.
- Newton development class discounts.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Apple Multimedia Program (AMP) provides resources to keep multimedia developers up-to-date on Apple's offerings for authoring and playback. It includes:

- The quarterly Apple Multimedia Information Mailing.
- Access to a special members-only area on the AMP Web site (http://www.amp.apple.com).
- Invitations to special events and participation in Apple events such as trade shows.
- Seeding opportunities.
- The Interactive Music Track, an extension of the AMP designed specifically for musicians, music industry members, and interactive music developers.

Apple Developer Catalog To order a product 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. You can also send e-mail to order.adc@applelink.apple.com (AppleLink ORDER.ADC), or write *Apple Developer Catalog*, P.O. Box 319, Buffalo, NY 14207-0319. The catalog is on the Web at http://www.devcatalog.apple.com.

Apple Developer University Course descriptions and schedules can be found on the Web at http://dev.info.apple.com and in the Developer Services area on AppleLink. You can also call (408)974-4897, fax (408)974-0544, send e-mail to devuniv@applelink.apple.com (AppleLink DEVUNIV), or write Developer University, Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs These programs vary on a country-by-country basis. For more information on any of Apple's developer support programs worldwide, call (408)974-4897, fax (408)974-7683, send e-mail to devsupport@applelink.apple.com (AppleLink DEVSUPPORT), or write Developer Support, Apple Computer, Inc., 1 Infinite Loop, M/S 303-2T, Cupertino, CA 95014.