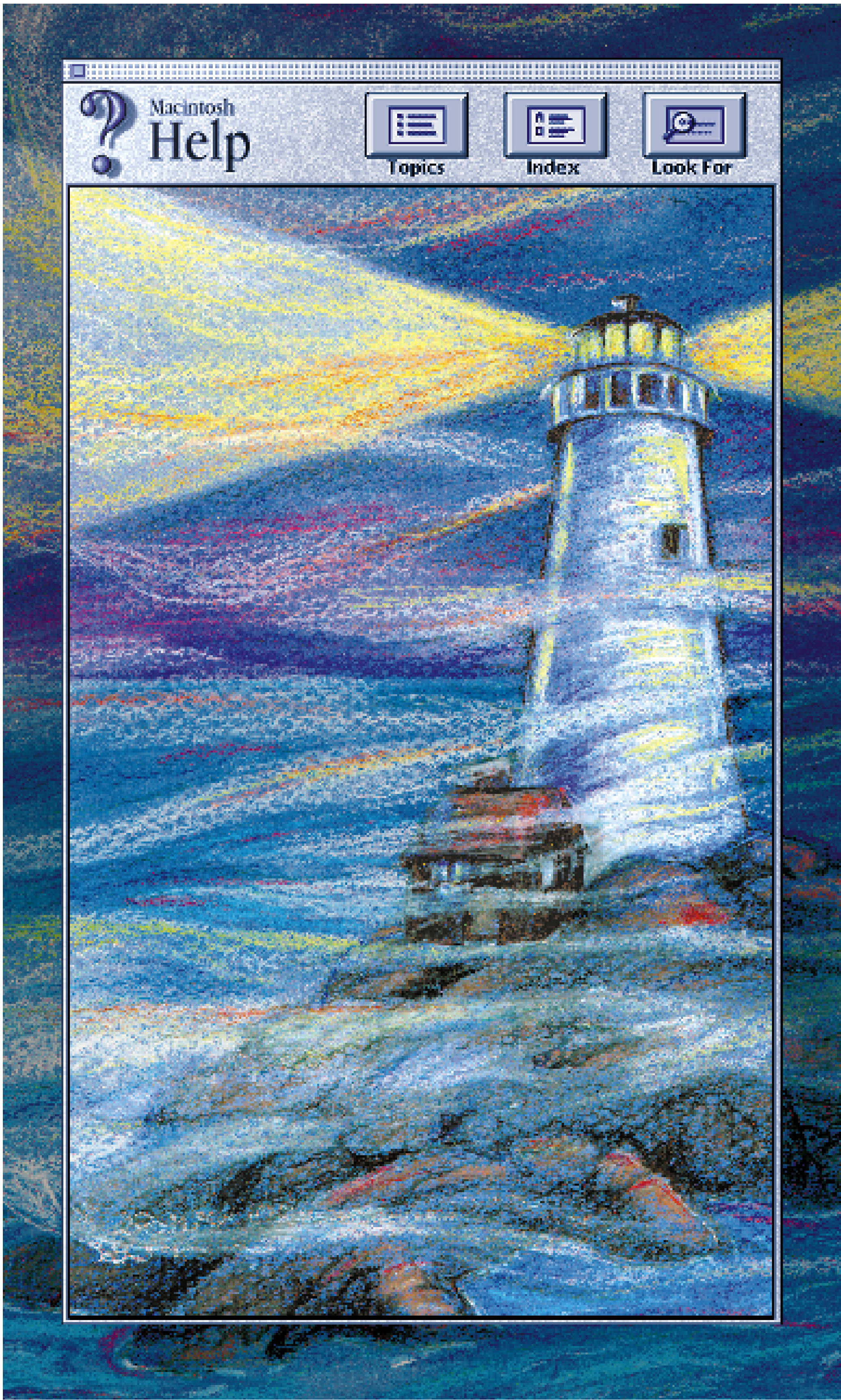
QuickTime Movie Playback • QuickDraw GX Debugging Version • Dogcow History, Part 2

develop

The Apple Technical Journal

Issue 18 June 1994



$10.00

develop

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*
Managing Editor *Cynthia Jasper*
Technical Buckstopper *Dave Johnson*
Our Boss *Greg Joswiak*
His Boss *Dennis Matthews*
Review Board *Pete ("Luke") Alexander, Dave Radcliffe, Jim Reekes, Bryan K. ("Beaker") Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*
Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Joe Williams*
Indexer *Marc Savage*

ART & PRODUCTION

Production/Art Director *Diane Wilcox*
Technical Illustration *John Ryan*
Formatting *Forbes Mill Press*
Film Services *Aptos Post, Inc.*
Prepress Production *PrePress Assembly*
Printing *Wolfer Printing Company, Inc.*
Photography *Sharon Beals, Cynthia Jasper, Matt Melmon, Diane Wilcox*
Cover Illustration *Hal Rucker and Corinne Okada of Rucker Huggins Design*

ISSN #1047-0735. © 1994 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, APDA, AppleLink, the dogcow logo, HyperCard, HyperTalk, LaserWriter, Lisa, LocalTalk, Mac, MacApp, Macintosh, Macintosh Quadra, MacTCP, Moof, MPW, and Newton are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, AppleScript, Balloon Help, develop, Finder, MacinTalk, Macintosh Duo, NewtonMail, Power Macintosh, PowerTalk, QuickDraw, QuickTime, and Sound Manager are trademarks of Apple Computer, Inc. Adobe is a trademark of Adobe Systems Incorporated, which may be registered in certain jurisdictions. PowerPC is a trademark of International Business Machines Corporation, used under license therefrom. NuBus is a trademark of Texas Instruments. All other trademarks are the property of their respective owners.

Printed on recycled paper

THINGS TO KNOW

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

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

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

Macintosh Technical Notes. Where references to Macintosh Technical Notes in *develop* are followed by something in parentheses like "(Memory 13)," this indicates the category and number of the Note on this issue's CD.

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

CONTACTING US

Feedback. Send editorial suggestions or comments to Caroline Rose at AppleLink CROSE, Internet crose@applelink.apple.com, or fax (408)253-8521. Send technical questions about *develop* to Dave Johnson at AppleLink JOHNSON.DK, Internet dkj@apple.com, CompuServe 75300,715, or fax (408)253-8521. Or write to Caroline or Dave at Apple Computer, Inc., 20525 Mariani Avenue, M/S 303-4DP, Cupertino, CA 95014.

Article submissions. Ask for our Author's Guidelines and a submission form at AppleLink DEVELOP, Internet develop@applelink.apple.com, or fax (408)253-8521. Or write to Caroline Rose at the above address.

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

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

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

Issue 18 June 1994

ARTICLES

COLUMNS

EDITOR'S NOTE

CAROLINE ROSE

One thing about magazine publishing: the wheel keeps turning no matter what. There can be no slips because of problems that crop up or any extra work that needs to be done; the "to press" date is set way in advance. So for this issue, having to deal with redesigning our layout and having our technical editor out on jury duty for months, I've been going crazy. The smiling face you see in this photo is not what I've looked like for most of the last three months!

But enough about me. Let's talk about the redesign. As you've no doubt already noticed, we've changed *develop*'s cover to make it look more like those you'll find on newsstands. We've added an eye-catching strip across the top and moved the article titles from the right side of the cover to the left, where they would peek out if the issues were stacked from left to right (imagine offset overlapping windows).

We've changed the layout in the inside to fit a bit more on each page. Issues of *develop* will be slimmer than before, but you'll still be getting the same amount of content. We couldn't resist tweaking a few other things while we were at it. Specifically, "listing boxes" now help keep code together in one tidy place. But we still put code close to where it's discussed, and we still place code in the body of the article if that makes more sense.

Another change in this issue is a direct result of the aforementioned jury duty: Dave Johnson's popular Veteran Neophyte column is missing. But don't worry; it will return once justice has been served. Meanwhile we do have two new columns: Balance of Power, with PowerPC™-related tidbits from Dave Evans, and Newton Q & A, which lets you interrogate a llama and receive a T-shirt if he uses your question.

As always, we'd like your feedback. There will be more changes to come, and we really want to know what works for you and what doesn't. So please, don't just gripe (or sing our praises) among yourselves; drop us a line and give us your $.02! All information about who to contact for what is now located in one handy place, on the inside front cover (along with other irresistible tweaks).

By letting us know how we can improve, you help us win awards like the one that we learned of while working on this issue: an Award of Distinction, and Best of Category (Monthly or Quarterly Magazines), in the Society for Technical Communication's 1993 Northern California Technical Publications Competition. So thanks, and please keep it coming!

CRose

Caroline Rose
Editor

CAROLINE ROSE (AppleLink CROSE) was part of a huge East-to-West Coast migration that took place one heady summer many years ago. She stumbled upon a technical writing job and took to it and to California like a programmer to Mountain Dew. She's been a programmer herself (at Tymshare) and even a manager (at NeXT), but marks her favorite work years as those spent writing and editing at Apple — first *Inside Macintosh* and later *develop*. To keep herself sane outside of work, Caroline has little do with computers and has been known to stop making sense. She does, however, keep her wits honed for Scrabble.•

LETTERS

NUMBER FORMATS FOLLOWUP

Regarding the article “International Number Formatting” in *develop* Issue 16: I’ve written a *free* ResEdit editor (“FMAT Editor”) that enables you to easily create Script Manager canonical formats and store them as resources. I hope you’ll find it useful.

— Michael Hecht

The author of the article looked at your editor, and he thought it was so useful that he submitted it to the CD. Thanks!

— Caroline Rose

NEW FLOATING WDEF ON CD

I’m the author of a freeware floating window WDEF known as the Infinity Windoid. It was one of the first such WDEFs to implement the System 7–style coloring of windows, tinge colors and all. It includes support for a title bar down the left side of a window, System 6’s coloring of windows, multiple monitors via DeviceLoop, and checking the available colors in 8-bit mode to determine whether the title bar should be color.

I recently finished up version 2.5.1 of the WDEF, which adds the ability to have a grow box, a title string in the title bar, and a slightly different style of title bar.

Please consider including this WDEF with source code on your CD. An appropriate place for it would be with Dean Yu’s floating windows code (Issue 15). Dean has stated that his WDEF is not highly robust, and providing my WDEF with the source would make a lot of sense. This would also help me meet my goal of making it available to as many Macintosh developers as possible.

— Troy Gaul

Thanks for bringing your WDEF to our attention. We’ve included it with Issue 15’s floating windows code on this issue’s CD. Dean has also tweaked the code again since his last updates. Check it out!

— Caroline Rose

GREAT MAG, BUT LOSE LEDGE

My vote for your View From the Ledge column is: lose it! Dilbert (the comic strip by Scott Adams) does a much better job at commenting on the political and social structures of corporate America.

Thanks for the fine publication. *develop* is the only periodical that doesn’t end up in the trash after I read it. Keep up the good work.

— Dave Lamkins

The editor thanks you for your kind words about develop. *As the author of View From the Ledge, I’ve been volunteered to address your remarks about that column.*

Scott Adams is unquestionably a genius. He has a certain wit and control of the English language that hasn’t really been equaled since Shakespeare, and an artistic flair that I’m certain makes all Florentines weep. There’s simply no way I can compete with this caliber of raw talent.

No doubt Mr. Adams gets invitations to dine with the President. I’ve never met any of the Presidents of Apple, let alone of the United States — although John Sculley did pull up next to me at a stoplight a few years

DON’T FORGET TO WRITE!
We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., 20525 Mariani Avenue, M/S 303-4DP, Cupertino, CA 95014 (AppleLink CROSE or JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

ago, and I used to have a Jean-Louis Gassée mask hanging in my cubicle.

Mr. Adams has the advantage of dealing with a world where the characters are two-dimensional, all of their worst traits are brought to the forefront, and the situations these characters get caught up in border on the absurd. None of those things can be said about the real office environment.

Having said all that, we're putting View From the Ledge on ice after this issue, and the thaw comes only if there's a chinook in the form of a strong reader response.

— Tao Jones

MISSING CD? A ROYAL PAIN

First, I'd like to say that I think *develop* is excellent. I await each issue with frenzied anticipation and find each one absorbing, informative, well written, and funny.

I'd also like to comment on the Bookmark CD. I can't, however, because we don't get it. I keep reading articles referring to "this issue's CD" and it's infuriating! We get our regular monthly Developer CD with *Apple Directions* but no *develop* CD. Is this because we're in England? Don't you like us any more? It's the royal family's fault, isn't it? They've made us look stupid. They're nothing to do with us, honest. We'd just like our CD.

—Richard Gibson

Some international developers receive develop *without the Bookmark CD as part of a mailing that includes the Developer CD and Apple Directions. Anyone who gets the Developer CD doesn't need the Bookmark CD (see "This issue's CD" on the inside front cover).*

We still like our British readers, so from our standpoint you should have no gripe with the royal family.

Thanks for writing!

—Caroline Rose

Do you yearn for the adulation of your colleagues?

YOUR NAME HERE

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

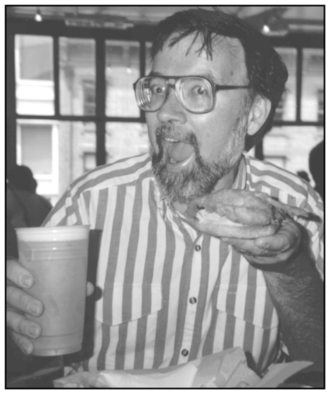
If you're a lot better at writing code than writing articles, don't worry. An editor will work with you. The result will be something you'll be proud to show your colleagues (and your Mom).

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

For Author's Guidelines, editorial schedule, and information on our incentive program, send a message to DEVELOP on AppleLink, develop@applelink.apple.com on the Internet, or Caroline Rose, Apple Computer, Inc., 20525 Mariani Avenue, M/S 303-4DP, Cupertino, CA 95014.

Giving Users Help With Apple Guide

A positive user experience means getting the task done with a minimum of hassle. But as applications engage in the features race, their complexity increases as well, leading to increased user frustration. One solution is to add a powerful help system that can guide the user through a task. The new Apple Guide system makes this chore easier than you might think.



JOHN POWERS

Someone has decided that your application needs help. Maybe it's a little heavy on the features, and Marketing is afraid that the user will be overwhelmed. Perhaps your user studies have shown that users can really benefit from some coaching when learning your interface. Or the finance experts have figured out that you can save a lot of money on printing and user support costs by putting help on the screen rather than in printed manuals. And yesterday the CEO declared that your product must win a gold medal in customer satisfaction.

It *can* be done.

This article tells you how to add Apple Guide, a new kind of help system, to your application with a pain and suffering level that you can control, ranging from almost nothing to only a tingle.

The article doesn't tell you what help your application should provide; that's up to your software designer, technical writer, instructional designer, interface designer, and whoever else likes to stir the design pot. Once you've decided what help you want and the technical writer has written it using the Apple Guide authoring tool, you integrate it. That's where this article — and this issue's CD, which contains a sample program with Apple Guide integrated — comes in.

WHAT IS APPLE GUIDE?

Apple Guide is a new kind of help system that acts as an interactive, task-oriented guide and will be available to all applications systemwide as of System 7.5. Apple Guide is based on more than two years of research on people's need for help while using their computers. Several key findings of this research underlie the Apple Guide design:

JOHN POWERS (AppleLink JOHNPOWERS), MWM, Apple employee, seeks intense relationship with software. Enjoys object-oriented design, craftsmanship, classical music, long walks, quiet moments, laser discs with popcorn, travel, engaging novels, and kinky polymorphism. Software must be willing to share time with extended family. Documentation a must.•

The Apple Guide Developer's Kit will soon be available from APDA (if it isn't already). It contains everything you need to author databases and integrate Apple Guide. Included in this kit is an authoring tool that compiles help content written with a standard word processor into an Apple Guide database file. The kit also contains sample help content files and documentation.•

- Users are task-oriented.
- Users seek help when they're frustrated doing a task.
- Help should be an interactive guide to accomplishing a task, not a static document.

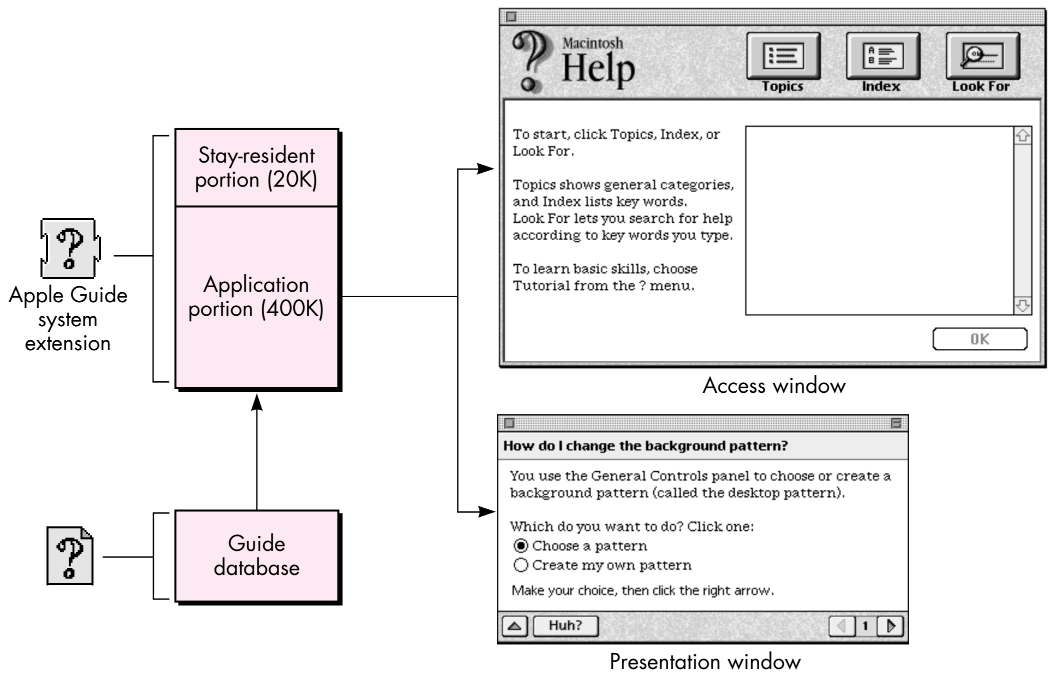
The paradigm is simple:

1. The user has a task to accomplish.
2. The user becomes frustrated when an obstacle prevents completion of the task.
3. The user asks for help.

Traditional help requires users to go to a printed or on-screen document to identify the problem in the context of the documentation. This in itself can become a frustration. Users would rather have someone (or something) guide them through the obstacle. That's what Apple Guide does: it acts as an interactive, task-oriented guide.

When users ask for help, the first thing they see is the *access window*, which provides three ways of selecting the help topic. Once the user selects the help topic, the access window is replaced by the *presentation window*, which presents the help topic as a series of panels containing text, graphics, QuickTime movies, and controls. (Note that in an application in which Apple Guide has been fully integrated, the help topic can be preselected for the user based on context — more on this later.) To experience Apple Guide for yourself, try out the application called MoGuide on this issue's CD.

Apple Guide is implemented as a system extension (see "What Systemwide Help Means" for some ramifications of this) that uses a guide database file to drive its interaction with the user, as depicted in Figure 1. The database file, written using the Apple Guide authoring tool, contains multimedia help content and instructions on how to interact with the user. The delivery engine is in the system extension, which



To start, click Topics, Index, or Look For.

Topics shows general categories, and Index lists key words. Look For lets you search for help according to key words you type.

To learn basic skills, choose Tutorial from the ? menu.

How do I change the background pattern?

You use the General Controls panel to choose or create a background pattern (called the desktop pattern).

Which do you want to do? Click one:

● Choose a pattern
○ Create my own pattern

Make your choice, then click the right arrow.

Figure 1. How Apple Guide works

WHAT SYSTEMWIDE HELP MEANS

Because it's implemented as a system extension, Apple Guide is available to all applications systemwide. It's designed to help users within and across application boundaries. Apple Guide can guide a user through multiple applications including, but not limited to, the application that invoked it.

Apple Guide's systemwide nature has ramifications for everything you do with it. While it enables delivering an unprecedented level of help to users, it also presents certain programming challenges. Consider the following characteristics of systemwide help and what they mean to you as a developer.

Help works across multiple applications. Your application's guide database can guide a user through multiple applications, not just your own. You can provide help for tasks that involve system-level operations and the use of other applications.

One ramification of this is that if the user switches from your application to another one after starting up Apple Guide and without quitting your application or Apple Guide, Apple Guide will still be present and showing your guide database. This may be desirable, depending on what help you're providing. If not, the user can quit Apple Guide from the help window. Don't force Apple Guide to quit when your application goes to the background.

Help is always available to the user. Your application can always invoke Apple Guide, even if it's in use by another application. That application's guide database will be closed and yours will be opened and shown instead.

This works the other way around as well — if the user invokes Apple Guide from the Help menu within another application while yours is open in the background, your guide database will be closed and the other application's will be opened and shown instead. Moreover, the other application can still make Apple Guide quit even if you closed its guide database to show your own.

The bottom line for you is that you can't assume that your guide database is always open, even if the user invoked it from within your application. A call for dealing with this is described later, under "Is Your Guide Database Open?"

The user is always in control. Users can quit Apple Guide at any time, even if they've switched from the application that started it up. Once again, this means that you can't assume your guide database is always open, even if the user invoked it from within your application.

contains two components: a stay-resident portion and a launch-as-needed application portion. With a RAM footprint of less than 20K, the stay-resident portion installs patches at startup time, manages the Help menu (the one labeled with a question mark in a balloon), and starts up Apple Guide from the Help menu. The application portion is loaded and run when help is being delivered; with a RAM footprint of 400K, it's launched as a faceless background application in its own heap. The Apple Guide system extension and several guide databases will be provided as part of System 7.5.

TO INTEGRATE OR NOT TO INTEGRATE?

You can put Apple Guide to work for you whether or not you decide to integrate it into your application. However, as in life, the level of pain you choose determines the amount of gain.

The easiest way to add Apple Guide to an application is to place an Apple Guide database file in the same folder as the application. No coding changes are necessary; the integration is automatic. The Apple Guide extension adds the database to the application's Help menu and launches the database when the user chooses it.

Here's what Apple Guide delivers without any changes to your application:

- help initiated from the Help menu
- interactive delivery of multimedia help content

- the ability to mark static interface objects, or dynamic interface objects identified using the Object Support Library (OSL), to draw the user's attention to them
- elementary to intermediate context sensitivity

This is a lot of help in itself. But if you want more control over Apple Guide, you'll need to make code changes in your application. Here's what those changes can add:

- information about where the user is in the help system
- more direct control over what help the user receives and responsiveness to the user's context
- help initiated from any control — including buttons, lists, and special keys — and from application menus
- guide databases with your own creator, type, and document icons
- the ability to use databases located in a folder separate from the application

If you want the gain and you're ready for the (slight) pain, read on.

To communicate with Apple Guide from your application, you use standard, trap-based function calls. The function calls enable your application to get information about Apple Guide, start up Apple Guide, respond to Apple Guide, modify help content, and quit Apple Guide. I'll explain how to do each of these in the sections that follow. You might want to examine the source code for the sample program MoGuide on this issue's CD to see the function calls used in context. Then you can try integrating Apple Guide into your own application.

GETTING INFORMATION ABOUT APPLE GUIDE

Your application can find out a number of different things about Apple Guide: whether it's installed, what its status is, whether an Apple Guide database is available, the number and characteristics of guide databases available, and whether your own guide database is open.

IS APPLE GUIDE INSTALLED?

Before you do anything with the Apple Guide extension, you need to determine whether it's installed. The following code shows how:

```
long  result=0;
OSErr err = Gestalt(gestaltHelpMgrAttr, &result);
if (err==noErr && (result & (1 << gestaltAppleGuidePresent)))
   ;  // Apple Guide is available.
else
   ;  // Apple Guide is not available.
```

WHAT'S THE STATUS OF APPLE GUIDE?

Once you've determined that Apple Guide is installed, you can get more information about its state — for example, is it active (displaying a help window)? If it is, which help window is being displayed? The latter information can be saved and used to restart users at the point where they left the help system.

The AGGetStatus function returns whether Apple Guide is active (kAGIsActive), sleeping (kAGIsSleeping), or not running (kAGIsNotRunning). In the active state, help is being provided (a guide database is open). In the sleeping state, the Apple Guide background application is loaded and running, but no help is being provided (no guide database is open). In the not-running state, the application portion of Apple Guide isn't loaded or running.

If Apple Guide is active, you can further determine whether the access window or the presentation window is showing. For example:

```
if (AGGetActiveWindowKind()==kAGAccessWindow)
   ;  // Apple Guide access window is showing.
else if (AGGetActiveWindowKind()==kAGPresentationWindow)
   ;  // Apple Guide presentation (topic) window is showing.
else
   ;  // No window is showing; Apple Guide is sleeping or not running.
```

Be sure you've determined that Apple Guide is installed before invoking AGGetStatus or any other Apple Guide function. Otherwise, if Apple Guide isn't present, you'll get an unimplemented trap error.

IS AN APPLE GUIDE DATABASE AVAILABLE?

You can find out whether an Apple Guide database is available to the current (frontmost) application, and furthermore, whether a specific type of guide database is available. (There are two file types and five database types, classified according to the type of help information they contain; see "About Apple Guide Database Files" for a rundown of the types.)

The AGGetAvailableDBTypes function returns bits set to correspond to the various types of guide databases available to (in the same folder as) the frontmost application.

```
enum AGDBTypeBit    // To test against AGGetAvailableDBTypes
{
   kAGDBTypeBitAny =         0x00000001,
   kAGDBTypeBitHelp =        0x00000002,
   kAGDBTypeBitTutorial =    0x00000004,
   kAGDBTypeBitShortcuts =   0x00000008,
   kAGDBTypeBitAbout =       0x00000010,
   kAGDBTypeBitOther =       0x00000080
};
```

The kAGDBTypeBitAny bit will be true if the application has any Apple Guide database file available. Thus, the following statement will test whether any guide database is available:

```
if (AGGetAvailableDBTypes() & kAGDBTypeBitAny)
   ;  // Some kind of database is available
```

You can simlarly use the other constants to test the availability of specific types of Apple Guide databases.

HOW MANY AND WHICH GUIDE DATABASES ARE AVAILABLE?

The tests described in the preceding section apply only to guide database files or their aliases in the same folder as the application. You can also find files outside the application's folder, by using the AGFile library that's included on this issue's CD. This library doesn't use or require the Apple Guide system extension and offers access to information about Apple Guide database files.

ABOUT APPLE GUIDE DATABASE FILES

Apple Guide database files provide the content of the help that your application gives users. These files can be of the following two types:

- kAGFileMain: Main or normal type of Apple Guide database file
- kAGFileMixin: Mix-in database file, which modifies a main database file with updates and additions at run time

Files of both types have a creator of kAGCreator, which is 'reno', and are further subdivided by database type according to the type of help information they contain. The database type is stored as part of the database content and determines where the file is represented in the Help menu, as indicated in Figure 2.

Apple Guide supports multiple guide database files but only one of each type, except for kAGFileDBTypeOther. You can have multiple kAGFileDBTypeOther files; they're added at the end of the Help menu, ordered by filename.

All Apple Guide database files located in the same folder as your application will be represented in the Help menu. (Note that aliases will work as well as database files themselves, but that folders nested within your application's folder will not be searched.)

If you don't want a guide database represented in the Help menu, you should separate the guide database from the application. In this case, you must provide a mechanism in your application to start up Apple Guide with that database, since the user can't do it from the Help menu. The section "Starting Up Apple Guide" tells all about this.

Databases can also have your creator and type, allowing you to use your own document icons. Databases identified this way won't show up on the Help menu and must be opened from your application.

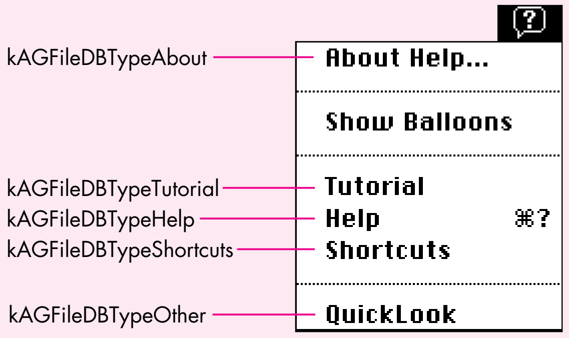


Figure 2. Menu items identified by database type

With the AGFile library, you can count and find guide databases of specific file and database types. Once you've found a database, you can get information about it. For example, the following code counts the number of main guide database files (those that aren't mix-in files) of type kAGFileDBTypeHelp in a specified volume and directory:

```
AGFileCountType   guideFileCount;
AGFileDBType      databaseType=kAGFileDBTypeHelp;
Boolean           wantMixin=false;

guideFileCount = AGFileGetDBCount(vRefNum, dirID, databaseType, wantMixin);
```

You can get the FSSpec for a particular type of Apple Guide database file in a specified volume and directory as follows:

```
AGFileCountType   guideFileCount;
AGFileDBType      databaseType=kAGFileDBTypeOther;
Boolean           wantMixin=false;
short             dbIndex=1;
FSSpec            dbSpec;

err = AGFileGetIndDB(vRefNum, dirID, databaseType, wantMixin, dbIndex,
      &dbSpec);
```

The AGFileGetIndDB call will return noErr and the FSSpec for the database file if it's present. Increment dbIndex to find additional guide databases of the same type.

Listing 1 shows how to accumulate a list of FSSpecs for all the guide databases in a specified volume and directory.

Listing 1. Accumulating FSSpecs for guide databases

```
AGFileCountType   guideFileCount;
AGFileDBType      databaseType=kAGFileDBTypeAny;
Boolean           wantMixin=false;
FSSpec            dbSpec;
Handle            hFileList;

// See how many guide files are available.
guideFileCount = AGFileGetDBCount(vRefNum, dirID, databaseType,
   wantMixin);
if (guideFileCount>0) {
   // Create a new list of file FSSpecs.
   hFileList = NewHandle(guideFileCount * sizeof(FSSpec));
   if (hFileList!=nil) {
      // Get each file and add to list.
      for (short i=1; i<=guideFileCount; i++) {
         if (AGFileGetIndDB(vRefNum, dirID, databaseType,
               wantMixin, i, &dbSpec)==noErr) {
            ((FSSpec*)(*hFileList))[i-1] = dbSpec;
         }  // if (...
      }  // for (short...
   }  // if (hFileList...
}  // if (guideFileCount...
```

Once you have the FSSpec for a database file, you can ask for more information. For example, the following code can be used to get the version and menu item name for a database:

```
AGFileMajorRevType   majorRev;
AGFileMinorRevType   minorRev;
Str255               menuName;

AGFileGetDBVersion(&fileSpec, &majorRev, &minorRev);
AGFileGetDBMenuName(&fileSpec, menuName);
```

IS YOUR GUIDE DATABASE OPEN?

As mentioned earlier in "What Systemwide Help Means," you can't assume your guide database is always open, even if the user invoked it from within your application. To cope with this fact of life, your application should call the AGIsDatabaseOpen function when it's switched from background to foreground, passing in the database's reference number, to see if its guide database is still open.

```
if (AGIsDatabaseOpen(myAGRefNum))
   ;  // The database with myAGRefNum is open.
else
   ;  // The database with myAGRefNum isn't open; someone else closed it.
```

If AGIsDatabaseOpen is true, you can also assume that Apple Guide is active. If AGIsDatabaseOpen is false and you haven't explicitly closed your database, another application or the user has closed it.

STARTING UP APPLE GUIDE

You can start up Apple Guide with either the first available guide database or one you specify. You can also control the initial view the user sees, choosing from seven different options. In this initial view you can present the user with a list of topics related to the context, or you can skip the initial view and instead take the user directly to a help topic.

DEFAULT STARTUP

The default startup, the quickest and easiest way to start up Apple Guide, opens the first available guide database. The database must be in the same folder as your application (the Finder is an exception — its guide databases are in the Extensions folder). The mechanism is similar to what happens when the user selects an item from the Help menu.

The default startup is accomplished by calling the AGOpen function with the kAGDefault flag:

```
err = AGOpen(kAGDefault, 0, nil, &myAGRefNum);
```

A reference for the database that was opened is returned in myAGRefNum. You must use this reference when you close the database.

Apple Guide starts up with the frontmost application's guide database and the startup conditions specified in the database. If more than one Apple Guide database is available for the application, the first database of type kAGFileDBTypeHelp is used. If no database is present, the error code kAGErrDatabaseNotAvailable is returned and Apple Guide doesn't start up.

DATABASE OTHER THAN THE DEFAULT

An application can also open a specific guide database when it starts up Apple Guide, by specifying the FSSpec for the database in the AGOpen call:

```
err = AGOpen(&myGuideDatabaseFSSpec, 0, nil, &myAGRefNum);
```

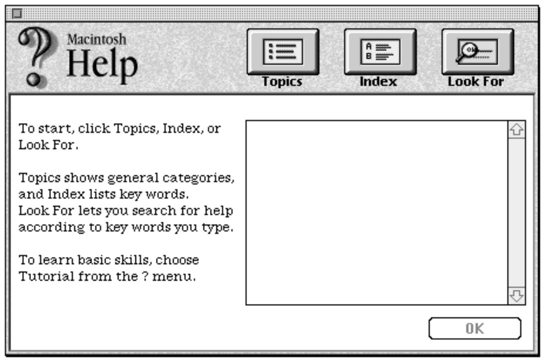
See the earlier section "How Many and Which Guide Databases Are Available?" for a suggested way to get a database FSSpec.

VIEW OTHER THAN THE DEFAULT

Apple Guide offers seven startup view options. This is normally controlled by the guide database; however, these defaults can be overridden with a startup function parameter. For example:

```
err = AGOpenWithView(&myGuideDatabaseFSSpec, 0, nil, kAGViewIndex,
      &myAGRefNum);
```

Six of the possible startup views are listed below. These views, which are access windows, are shown in Figure 3. Startup view number 7 is the topic or presentation window view, which is started with a different function call (described below in "Going Directly to a Topic").

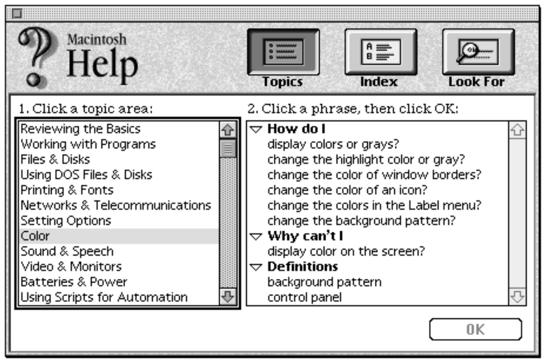


To start, click Topics, Index, or Look For.

Topics shows general categories, and Index lists key words. Look For lets you search for help according to key words you type.

To learn basic skills, choose Tutorial from the ? menu.

View 1: Full-window howdy



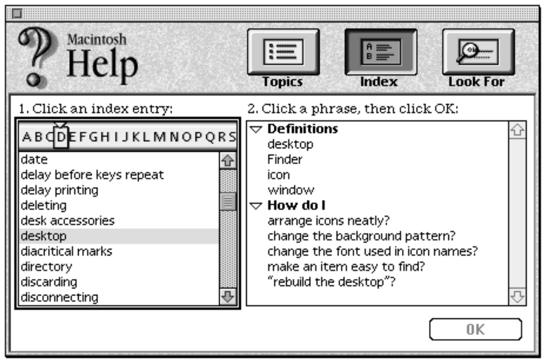
1. Click a topic area:

Reviewing the Basics
Working with Programs
Files & Disks
Using DOS Files & Disks
Printing & Fonts
Networks & Telecommunications
Setting Options
Color
Sound & Speech
Video & Monitors
Batteries & Power
Using Scripts for Automation

2. Click a phrase, then click OK:

How do I
display colors or grays?
change the highlight color or gray?
change the color of window borders?
change the color of an icon?
change the colors in the Label menu?
change the background pattern?
Why can't I
display color on the screen?
Definitions
background pattern
control panel

View 2: Full-window topic areas



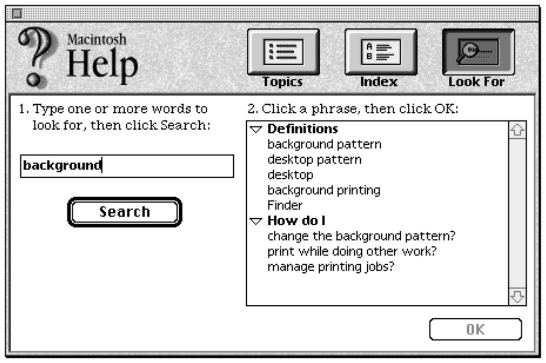
1. Click an index entry:

date
delay before keys repeat
delay printing
deleting
desk accessories
desktop
diacritical marks
directory
discarding
disconnecting

2. Click a phrase, then click OK:

Definitions
desktop
Finder
icon
window
How do I
arrange icons neatly?
change the background pattern?
change the font used in icon names?
make an item easy to find?
"rebuild the desktop"?

View 3: Full-window index terms



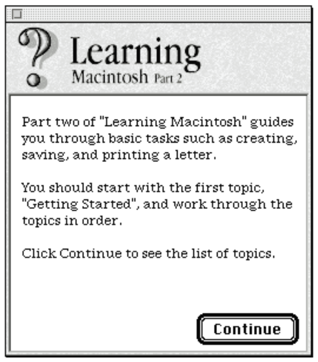
1. Type one or more words to look for, then click Search:

background

2. Click a phrase, then click OK:

Definitions
background pattern
desktop pattern
desktop
background printing
Finder
How do I
change the background pattern?
print while doing other work?
manage printing jobs?

View 4: Full-window look-for

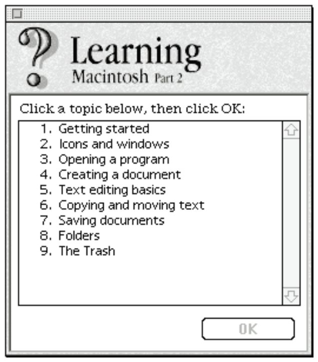


Part two of "Learning Macintosh" guides you through basic tasks such as creating, saving, and printing a letter.

You should start with the first topic, "Getting Started", and work through the topics in order.

Click Continue to see the list of topics.

View 5: Single-list howdy



Click a topic below, then click OK:

1. Getting started
2. Icons and windows
3. Opening a program
4. Creating a document
5. Text editing basics
6. Copying and moving text
7. Saving documents
8. Folders
9. The Trash

View 6: Single-list topics

Figure 3. Six startup view options

```
enum
{
   kAGViewFullHowdy =      1,    // Full-window howdy
   kAGViewTopicAreas =     2,    // Full-window topic areas
   kAGViewIndex =          3,    // Full-window index terms
   kAGViewLookFor =        4,    // Full-window look-for (search)
   kAGViewSingleHowdy =    5,    // Single-list howdy
   kAGViewSingleTopics =   6     // Single-list topics
};
```

Since almost all guide databases contain multiple topic areas, they usually start up with the kAGViewFullHowdy view. If you specify a kAGViewSingleHowdy or kAGViewSingleTopics view for these kinds of guide databases, only the topics for the first topic area will be shown.

PRIMING A SEARCH

An application can start up Apple Guide (or restart it if it's already running) in the look-for view with a list of context-sensitive topics that match a search string (a Str255). To initiate a context-sensitive search, use the AGOpenWithSearch function and include a look-for search string as follows:

```
err = AGOpenWithSearch(&myGuideDatabaseFSSpec, 0, nil, mySearchString,
      &myAGRefNum);
```

GOING DIRECTLY TO A TOPIC

Sometimes you may want to initiate help when the user clicks a Help button in a dialog or Option-clicks an interface object. In these cases, you have a good idea of the context and can bypass the Apple Guide access window, going directly to the presentation window with the selected help topic. Here's how:

```
err = AGOpenWithTopic(&myGuideDatabaseFSSpec, 0, nil, myTopicId,
      &myAGRefNum);
```

The presentation window will appear with the first panel of the topic.

The Apple Guide authoring tool usually handles topics by name and quietly assigns a topic ID number when the database is compiled. However, you can override this automatic assignment by specifying your own topic ID. See the authoring tool documentation for information on how to assign your own topic ID numbers.

RESPONDING TO APPLE GUIDE

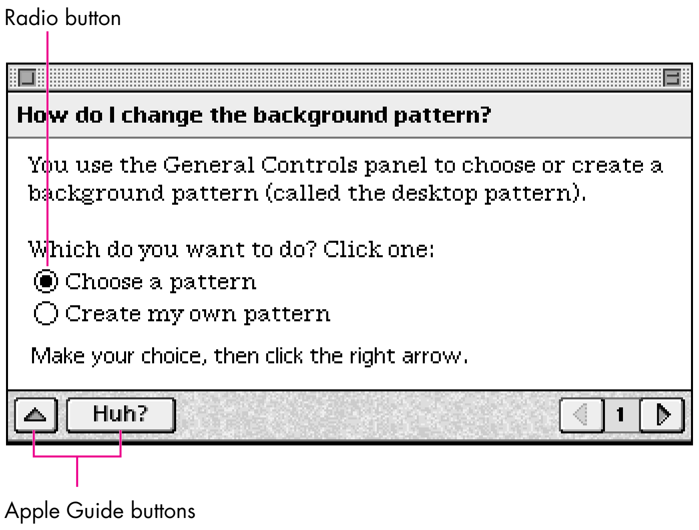
The author of an Apple Guide database file can initiate a number of different kinds of events to which your application may need to respond. These include attaching Apple events to controls in help topics, drawing coach marks on dynamic interface objects, sending Apple events when certain panels appear, and using context checks to determine whether a presentation panel should be shown or skipped.

RECEIVING THE AUTHOR'S EVENTS

A help author can attach an Apple event to any control in an Apple Guide presentation window. The controls can be Macintosh buttons, checkboxes, radio buttons, or Apple Guide buttons (the last two types of controls are illustrated in Figure 4). The author's action specification for these controls will contain

- the target for the action (usually an application signature)
- the Apple event class
- the Apple event ID
- the optional key
- optional data for the optional key

The action might be directed to Apple Guide — for instance, to start up another presentation window, to return to the access window, or to go to another panel in the same topic. The action can also be directed to your application; in that case, the target is your application signature, and the Apple event class and event ID are ones for which you've installed a handler. The content of the optional key/data field can be anything; how it's used is up to the application developer. Apple Guide extracts the



How do I change the background pattern?

You use the General Controls panel to choose or create a background pattern (called the desktop pattern).

Which do you want to do? Click one:

(● Choose a pattern) ○ Create my own pattern

Make your choice, then click the right arrow.

Figure 4. Controls in a presentation window

optional data field, calculates its size in bytes, calls AEPutParamPtr with a descriptor type of typeChar, and sends the Apple event.

To receive the action, install an Apple event handler for the class and event ID. Your handler can get the optional data parameter using your key.

There is no reply to this event. If you want to reply, see the section "Responding to Context Checks."

PROVIDING OBJECT LOCATION FOR COACH MARKS

Coach marks are circles, X's, underlines, and arrows that can be drawn on interface objects to direct the user's attention to them, as illustrated in Figure 5. Using such marks has been shown in tests to be much more effective than putting drawings of the objects into the help content (users tend to click on the drawing, not the real thing). The Apple Guide authoring tool provides a fairly complete set of instructions to mark many common interface objects, such as the following:

- menu titles and items
- windows and any coordinates relative to a window
- window elements such as close boxes, size boxes, titles, and scroll bars
- items that have a help balloon "hot rectangle"
- dialog items

These are all relatively static elements with locations that can be found by Apple Guide's marking facility. Some interface objects, on the other hand, are dynamically located at locations known only to the application. If you've identified a dynamic interface object using the Object Support Library (OSL), you can use Apple Guide's built-in OSL coach-marking facility to locate and mark the object. See the documentation for the Apple Guide authoring tool for details.

If you want to mark dynamic interface objects and you're not using the OSL, Apple Guide provides another method for you to use. In this method a handler installed by Apple Guide calls your application asking for the rectangle of a named object. To use

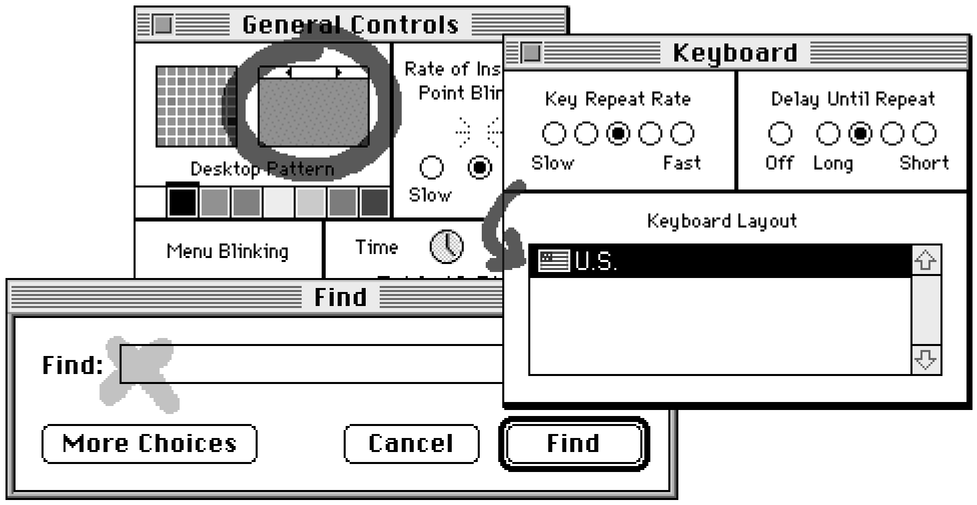


Figure 5. Coach marks

this method, your application must install a coach handler that takes the object name and replies with the object rectangle in global coordinates.

Here's an example. First, you install the coach handler:

```
AGCoachRefNum myAGCoachRefNum;
OSErr err = AGInstallCoachHandler(MyCoachReplyProc, myRefCon,
                                    &myAGCoachRefNum);
```

The prototype for a CoachReplyProc is as follows:

```
typedef pascal OSErr (*CoachReplyProcPtr) (Rect* pRect, Ptr name, long refCon);
```

Apple Guide will invoke your CoachReplyProc with the name (a null-terminated C-string) and myRefCon from the AGInstallCoachHandler call. Set the value of pRect to the object rectangle in global coordinates. The following is an example of a CoachReplyProc. Apple Guide will draw the coach mark around, under, or through the rectangle provided in the reply.

```
pascal OSErr MyCoachReplyProc(Rect* pRect, Ptr name, long refCon)
{
   // Look up the name and return the rect in global coordinates.
   OSErr err = MyRectForName(pRect, name);
   return err;
}
```

Coach handlers persist across the opening and closing of guide databases. However, you must remove the coach handler when your application quits. Use the AGCoachRefNum that was returned to you in the AGInstallCoachHandler call.

```
OSErr err = AGRemoveCoachHandler(&myAGCoachRefNum);
```

RESPONDING TO THE USER'S HELP FOCUS

You can get Apple Guide to notify your application whenever a particular panel is being displayed (in other words, whenever it's the user's help focus) by taking advantage of Apple Guide's authorable actions. To do this, use the authoring tool to attach an Apple event of your choosing to an “On Panel Show” or “On Panel Hide” action. When the panel is shown or goes away, the Apple event is sent to your application, which can then respond appropriately. For example, it might be useful to attach an Apple event to an “On Panel Show” action for the last panel in a series of instructions, to signal that the user has reached the end. See the authoring tool documentation for information on how to attach an Apple event to a panel action.

RESPONDING TO CONTEXT CHECKS

Apple Guide can use context checks to get information about the user’s environment and, based on that, to decide which panels to show in a presentation window. This enables the Apple Guide author to tailor the help to the user’s current context. It gives an individualized feel to the instruction and avoids requiring the user to navigate through irrelevant information, things that users appreciate a lot. When you integrate Apple Guide, the user’s context will usually be your application. The help author will want to use information from your application to determine what help the user needs. You’ll need to respond to these context checks by installing an Apple event handler, processing the context check, and returning true or false.

Here’s how it works: An author attaches conditions to selected panels. These conditions can be based on context checks or on user controls in other panels. While the user is reading the content of a presentation window, Apple Guide looks ahead to see which panel to show next. It evaluates the conditions, including context checks, attached to each panel. When Apple Guide evaluates a context check, it invokes the Apple event handler for the context check. The results of the evaluation based on the context check determine whether the panel should be shown or skipped.

Context checks are usually encapsulated as Apple Guide “external modules” and attached to a database. An external module is a code resource that executes in response to the query from Apple Guide. Creating and using external modules is outside the scope of this article, but see the TContext::ReplyToContext function in the sample code for an example of how to handle context checks from within your application.

Apple Guide handles context-check queries similarly to coach-mark queries. You install a context-check handler and Apple Guide calls it. First, the installation of the handler:

```
AGContextRefNum myAGContextRefNum;
OSErr err = AGInstallContextHandler(MyContextReplyProc, eventID, myRefCon,
                                    &myAGContextRefNum);
```

The prototype for a ContextReplyProc is as follows:

```
typedef pascal OSErr (*ContextReplyProcPtr)(Ptr pInputData,
                    Size inputDataSize, AEEventID eventId,
                    Ptr *ppOutputData, Size *pOutputDataSize,
                    long refCon);
```

When Apple Guide needs to do a context check, it invokes your callback with the context-check inputData and eventId. It also passes the refCon that you provided in the AGInstallContextHandler call. The ContextReplyProc should use NewPtr to create a storage area for a short, set *ppOutputData to the value of the pointer, and set *pOutputDataSize to sizeof(short). Always return a pointer to a short. Apple Guide will dispose of the pointer.

You’re probably wondering why we didn’t pass a Boolean instead of *ppOutputData and *pOutputDataSize. The reason is that the context reply is a special case of a more general reply mechanism, which I'm not going to go into here. Just take my word for it — it has to be this way.

Context handlers persist across the opening and closing of help databases. However, you must remove the context handler when your application quits. Use the AGContextRefNum that was returned to you in the AGInstallContextHandler call.

```
OSErr err = AGRemoveContextHandler(&myAGContextRefNum);
```

If you just want to field Apple events that don't require a reply, see the earlier section "Receiving the Author's Events."

MODIFYING HELP CONTENT

So far, we've talked only about main guide database files, the standalone files that support all or part of an application. You can modify the content of a main guide database by mixing in a mix-in guide database.

When a field upgrade involves new features and you don't want to release a revised guide database, you can use Apple Guide's mix-in files to add help content to the main guide database file. The mix-in files don't appear in the Help menu. When Apple Guide starts up, each mix-in file is merged with a main guide database file of the corresponding DBType. Multiple mix-in files can be mixed in.

The process is automatic, but the guide database author can control it by using Gestalt selectors. Each guide database file contains three Gestalt selectors. The selectors must evaluate to true in order for a main database to appear in the Help menu or a mix-in database to be mixed in. The Gestalt selectors in a given file are combined using Boolean OR logic. Empty Gestalt selectors are ignored. If all Gestalt selectors are empty, the file is unconditionally accepted.

There's a really cool way for the programmer to handle this. If the first Gestalt selector is the literal 'QLfy', Apple Guide will call a code resource of type 'QLfy' in the database file. If the code resource returns true, the database file will be mixed in; if it returns false, it won't be. The prototype for the code resource is as follows:

```
short(*QualifyFuncType)(void);
```

The AGFile library described earlier provides accessor functions that you can use to get the selectors and their values from any database. Only the authoring tool can change the selectors, however.

QUITTING APPLE GUIDE

When you've finished using a guide database, use the AGClose function with a pointer to the AGRefNum for the database.

```
err = AGClose(&myAGRefNum);
```

This will close the database and leave Apple Guide running in the background. When you've finished with Apple Guide, use the AGQuit function.

```
err = AGQuit();
```

This will make Apple Guide quit. If you attempt to call AGQuit without first closing the database, you'll get a kAGErrDatabaseOpen error.

If you start Apple Guide from your application, you should make it quit. Otherwise, the Apple Guide background application and help window will hang around without your application and consume system resources. This persistence is a result of the fact that Apple Guide is available systemwide. Even if the user quits Apple Guide by clicking the Cancel button or the close box, you must still balance every AGOpen function with an AGClose.

STEPS TOWARD INTEGRATION

Now you know what's involved in integrating Apple Guide into your application. If you're starting with a new product, you can build the use of Apple Guide into its design from the beginning; this can provide you with a competitive edge. If you're revising an existing product, you can add Apple Guide as a key feature. In either case, the instructional designer or technical writer should work closely with the software engineers to include an interactive task-oriented guide as an integral part of the design. Here's a suggested plan for doing that:

1. The instructional designer prepares a specification for how the application, Apple Guide, and the paper documentation will work together. The specification should include a description of the information that the guide database needs from the application and vice versa — for example, how help is started, Apple events, context checks, and coach-mark locations.
2. The software engineers use the instructional designer's specification to add the necessary Apple Guide features to the application.
3. The instructional designer authors the database.
4. The use of the guide database is included in the application's test suite.

PUTTING IT ALL TOGETHER

It's up to you and your database author to assemble Apple Guide's features into useful and coherent help for your user. This section gives some suggestions for making Apple Guide as helpful as it can be.

MULTIPLE DATABASE FILES

You should provide different guide database files for different needs. Apple Guide database possibilities include the following:

- an Apple Guide "agent" that guides the user through a task
- a tutorial to provide basic instruction
- command reference
- application shortcuts
- databases that match the user's level of experience
- a "here's what's new in this release" database instead of a read-me file (Marketing will love it!)

HELP EVERYWHERE YOU NEED IT

Your user should have access to help under all conditions. Here are some possibilities:

- from the Help menu
- from a help button in your tool palette

- from a help button in your modeless alerts and dialogs
- from a help hot key, enabling the user to click an object and bring up a relevant help topic

CONTEXT SENSITIVITY

You, too, can use the buzzword "context sensitive." Use the Apple Guide features to provide "smart" help — help that's tailored to the user's current context:

- Start up Apple Guide in the look-for view with a list of context-sensitive topics.
- Bypass the access window and go directly to the presentation window when the user initiates help from a control.
- Respond to the guide database author's context checks.
- Selectively mix in database content.

INTERACTIVITY

The built-in features of Apple Guide provide a high degree of interactivity. You can provide even more with integration:

- Identify dynamic interface objects with coach marks.
- Have Apple Guide notify you when the user is at a particular help panel (the help focus) so that your application can respond appropriately.
- Have your help database author add user controls that send Apple events to your application. Use them to allow the user to interact with you.

NEXT STOP, GOLDEN MASTER

This article has covered the basics of how your application can integrate Apple Guide. There's more in the complete API, but you have enough here to get started. With or without integration, the help system is there to serve your user. It shouldn't be compensation for bad interface design, nor should you add it just to save money on manuals or user support. Help is there to make your user more productive with your product, leading to a positive user experience, lots of recommendations, and sales, sales, sales.

It *can* be done.

Thanks to our technical reviewers Peter Commons, Winston Hendrickson, Josh Jacobs, Dave Lucky, Dave Lyons, and Eric Soldan.•

PETER HODDIE

SOMEWHERE IN QUICKTIME

Basic Movie Playback Support

Adding basic QuickTime movie playback support to most applications is simple, often just one day's work. Developers who want to do this turn first to *Inside Macintosh: QuickTime*, where it says to use a movie controller component. In *Inside Macintosh: QuickTime Components*, you find some elementary movie controller code samples, followed by a large reference section. This is usually enough to get started, but there are a few common problems. This column addresses some of them, with special attention to compatibility with future QuickTime releases. It assumes you're familiar with basic QuickTime and movie controller concepts.

OPENMOVIEFILE

All QuickTime movie files contain a movie resource, usually stored in the file's resource fork, and the actual movie data, stored in the file's data fork. To support cross-platform QuickTime movies, QuickTime's Movie Toolbox also allows the movie resource to be stored in the data fork along with the movie's data. (To learn how this is done, see John Wang's Somewhere in QuickTime column in *develop* Issue 17.) The usual sequence of calls to load a QuickTime movie from a file is: OpenMovieFile, NewMovieFromFile, CloseMovieFile.

In the common case of the movie resource stored in the resource fork, OpenMovieFile returns a file reference to the resource fork of the movie file, NewMovieFromFile loads the movie resource from that resource fork and creates a QuickTime movie, and CloseMovieFile closes the resource fork.

But if the movie was created on a computer running Microsoft Windows and QuickTime for Windows (using Adobe™ Premiere, for example), the file won't have a resource fork. Still, you can use the exact same sequence of calls. When OpenMovieFile is called, the file reference returned refers to the data fork; NewMovieFromFile loads the movie from the data fork, and CloseMovieFile closes the data fork.

Some developers don't use OpenMovieFile; they use FSpOpenResFile instead. While this works fine with movies made specifically for the Macintosh, it fails miserably otherwise. There's a sample movie with no resource fork, QuickBuck, on this issue's CD, so you can test this situation with your applications.

If you need to know whether OpenMovieFile opened the resource fork or the data fork, you can examine the file reference it returns, as follows:

```
pascal Boolean IsDataFork(short fileReference)
{
   FCBPBRec    anFCB;
   Str63       fName;

   anFCB.ioVRefNum = 0;
   anFCB.ioRefNum = fileReference;
   anFCB.ioFCBIndx = 0;
   fName[0] = 0;
   anFCB.ioNamePtr = (StringPtr)fName;

   if (PBGetFCBInfoSync(&anFCB) != noErr)
      return false;

   return (anFCB.ioFCBFlags & 0x0200) == 0;
}
```

NEWMOVIECONTROLLER

When you need a user interface for playing a movie, you should use NewMovieController to create a movie controller appropriate for use with that movie.

A common mistake is to instead use the Component Manager routine FindNextComponent or OpenDefaultComponent to locate a movie controller. This finds the first movie controller in the system's list of registered components. QuickTime has always contained only one movie controller, so this worked fine. However, future versions of QuickTime will almost certainly include other movie controllers, so the first one isn't necessarily the most appropriate one.

To help track down those offending applications that don't use NewMovieController, there's a system

PETER HODDIE writes code to introduce hard-to-find bugs into QuickTime. In his spare time he writes code to introduce even harder-to-find bugs into QuickTime.•

extension on this issue's CD which contains a different movie controller. You'll also find a movie, Other Controller Movie, that *should* invoke the sample movie controller. If any other movie invokes the sample movie controller, or if Other Controller Movie invokes the standard movie controller, the application you're testing isn't using NewMovieController. This will cause undesirable results in the not-so-distant future.

UPDATE EVENTS

If you use a movie controller in the recommended way (that is, you allow all events to be filtered through MCIsPlayerEvent), it updates all areas of the window covered by the movie and the movie's controls. Usually that's all a window contains, so all update events are completely handled by the movie controller. This works so well that some developers actually forget to support update events at all.

Unfortunately, it's not always so simple. QuickTime movies aren't always rectangular. If the movie is round and the window is rectangular (as in Figure 1), there are areas in the window that are not covered by the movie or the movie controls. Any update events in these areas are the responsibility of the application.

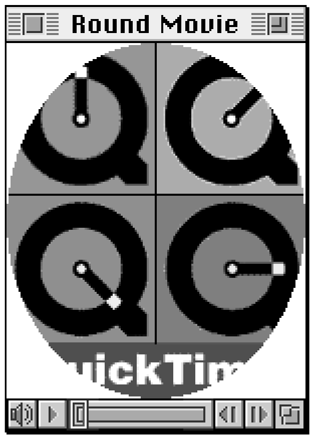


Figure 1. A nonrectangular movie

For applications using MCIsPlayerEvent, handling update events is easy:

```
BeginUpdate(theWindow);
EraseRect(&theWindow->portRect);
EndUpdate(theWindow);
```

This sample code erases all areas of the window besides the movie and its controls. Normally, erasing the portRect of the window would erase the entire window, but MCIsPlayerEvent sets the update region to just the areas it didn't already handle.

If you don't handle update events, things are even worse than you might think. The window won't be updated correctly, but more important, the operating system will keep generating new update events. Update events have a higher priority than idle events, so the system will never generate idle events — the movie will receive no time to play.

A sample round movie is provided on this issue's CD so that you can test your handling of update events.

KEYSTROKES

The standard movie controller provides for extensive keyboard control from the user but ignores keystrokes by default. They can be enabled with a single line of code:

```
MCDoAction(mc, mcActionSetKeysEnabled,
           (void *)true);
```

You might want to enable keystrokes only under certain circumstances. For example, a word processor might allow the movie controller to receive keystrokes only when a movie is selected. You can use the mcActionSetKeysEnabled action to enable and disable keystrokes as necessary.

MOUSE CLICKS

All applications that use the standard movie controller pass mouse clicks on to the controller. But not all applications pass mouse clicks made on the movie itself. Failure to pass such clicks will cause problems with any future movie controllers that allow the user to interact directly with the contents of the movie. For example, a movie controller might allow the user to pan around the image by dragging on the movie; if mouse clicks aren't passed through, using either MCClick or MCIsPlayerEvent, this feature won't work.

MOVIE CONTROLLER HELP

The standard Apple movie controller is simple enough for most people to understand immediately, but it supports help balloons anyway (future movie controllers might be less obvious). If Balloon Help is turned on, the standard movie controller automatically displays help for its various controls, as well as for the QuickTime movie itself. You don't have to do anything at all for this to work.

A problem can arise if your application puts up its own help balloons. Since QuickTime movies are often embedded in a larger document, the help balloons may conflict. The result is that the movie controller's help balloon alternates with the application's help balloon.

(Use Balloon Help with the Scrapbook desk accessory included with QuickTime to see what this looks like.)

The preferred solution is to stop the application from displaying a help balloon when the cursor is over a QuickTime movie or movie control. It's easy to tell whether a given point in a window intersects the movie:

```
Boolean PointInMovieController(MovieController
      mc, WindowPtr w, Point where)
{
   RgnHandle   rgn;
   Boolean     result = false;

   rgn = MCGetWindowRgn(mc, w);
   if (rgn != nil) {
      result = PtInRgn(where, rgn);
      DisposeRgn(rgn);
   }
   return result;
}
```

A second solution is to stop the movie controller from displaying its help balloons — necessary if you want to display your own help for QuickTime movies. To do this, install an action filter on the movie controller. Every action that occurs in the movie controller (play, step, update, key down, and so on) is passed through a single filter function. Through this filter, an application can gain access to all activity that occurs in the movie controller.

The MegaMovies application on this issue's CD provides a window that displays events that pass through the action filter. The action of interest is mcActionShowBalloon. When this action is sent, QuickTime is about to put up a new help balloon. One of the parameters passed to the action filter is a pointer to a Boolean. The filter can set this Boolean to false to tell the movie controller not to show a balloon. The following code fragments show how to install a simple action filter to prevent the movie controller from displaying help balloons.

```
pascal Boolean noBalloonsActionFilter
            (MovieController mc, short action,
             void *params, long refCon)
{
   if (action == mcActionShowBalloon)
      *(Boolean *)params = false;
   return false;
}

. . .
MCSetActionFilterWithRefCon(mc,
      &noBalloonsActionFilter, 0);
```

CURSOR SHAPE

Many applications change the shape of the cursor depending on what it's currently over. The standard movie controller never changes the cursor, but other movie controllers might want to. Unfortunately, many applications need to control the cursor themselves — when a movie controller changes the cursor, these applications change it back immediately.

A simple solution is for applications to change the cursor only when it's first placed over a movie. (To determine whether a point is over the movie, use PointInMovieController.) After that, let the movie controller control the cursor until it exits the area over the movie. To give the movie controller the opportunity to change the cursor's shape, you must call either MCIsPlayerEvent or MCIdle frequently while the cursor is over the movie, even if the movie is stopped. The sample movie controller on this issue's CD changes the cursor when it's over the movie, providing an easy way to debug such a scheme.

WINDOW ALIGNMENT

A simple way to improve a QuickTime movie's playback performance is to ensure that the movie is at a good location on the screen. The exact definition of a "good location" varies, based on the screen depth and the processor. A typical good location is one where the first pixel of each scan line begins on a long-word boundary. This allows the decompressors to write data in the most efficient way. On slower machines, proper placement can provide the necessary performance improvement to deliver smooth playback.

Fortunately, applications don't have to understand the details of how to find a good location. QuickTime's Image Compression Manager provides routines to position a window at these locations. When you create a window, you can use AlignWindow to move it to a good location before making it visible. If a window is to be moved, AlignScreenRect will modify the chosen location to make it a good location. When the user drags a window, call DragAlignedWindow instead of DragWindow to place the window in a good location. Examples of these calls are shown below.

```
WindowPtr   w;
Movie       m;
Rect        r;

// Code to create a properly aligned window.
w = GetNewCWindow(128, nil, (WindowPtr)-1);
m = getMovie();
GetMovieBox(m, &r);
AlignWindow(w, false, &r, nil);
```

```
// Code to drag a window with a movie in it and
// keep the window aligned properly.
GetMovieBox(m, &r);
DragAlignedWindow(w, theEvent.where, nil, &r, nil);
```

These alignment routines were added in QuickTime 1.5, so make sure that QuickTime 1.5 or later is installed before you call them.

MOVIE CONTROLLER EDITING

The standard movie controller supports the editing commands Undo, Cut, Copy, Paste, and Clear, but this functionality is turned off by default. To turn it on, call MCEnableEditing as follows:

```
MCEnableEditing(mc, true);
```

You can then use movie controller routines to implement editing:

```
Movie    m = nil;

switch (editMenuSelection) {
   case menuUndo:    MCUndo(mc);
                     break;
   case menuCut:     m = MCCut(mc);
                     break;
   case menuCopy:    m = MCCopy(mc);
                     break;
   case menuPaste:   MCPaste(mc, nil);
                     break;
   case menuClear:   MCClear(mc);
                     break;
}
if (m != nil) {
   PutMovieOnScrap(m, 0);
   DisposeMovie(m);
   }
```

Now you have to enable and disable the various menu items. You could call MCGetControllerInfo, which returns a long word of flags indicating, among other things, which Edit menu items should be enabled.

With QuickTime 1.5, there's an easier way: call MCSetUpEditMenu, and the movie controller will enable and disable the items in the Edit menu for you.

```
MCSetUpEditMenu(mc, theEvent.modifiers,
                  editMenuHandle);
```

This routine will even change the menu contents if appropriate. For example, Undo becomes Undo Paste if the last movie controller action was Paste; after Undo Paste is chosen, it becomes Redo Paste. What's more, if the user holds down modifier keys when pulling down a menu, other commands change as well. For example, holding down the Option key changes Paste to Add and Clear to Trim. (See Figure 2.)

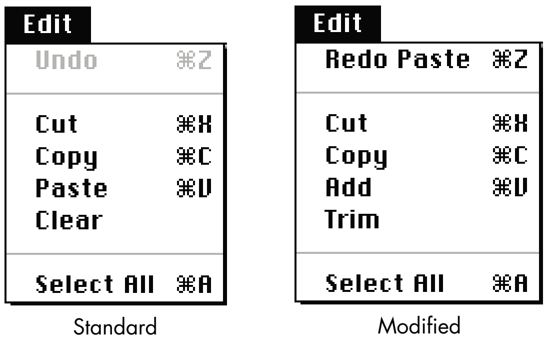


Figure 2. Standard and modified Edit menus

MCSetUpEditMenu assumes the Edit menu is arranged in the standard way. If yours is nonstandard, you'll need to use MCGetMenuString to obtain the appropriate text for each standard Edit command, and then enable and disable the menu items according to the information from MCGetControllerInfo.

JUST DO IT

It's so easy to add movie playback support that it's often well worth the effort. As long as you keep these few simple things in mind, you shouldn't have any problems, even with future versions of QuickTime.

Thanks to Jim Batson, Ken Doyle, and John Wang for reviewing this column.•

Programming for Flexibility: The Open Scripting Architecture

Users — and developers — waited a long time for the Macintosh Operating System to support the ability to attach and run scripts as a way of customizing applications. The Open Scripting Architecture (OSA) that's part of AppleScript finally provides the necessary services. Now you can realize massive gains in flexibility by using embedded scripts and can pass similar gains along to the user by making your application OSA savvy.

PAUL G. SMITH

Thanks to Apple's Open Scripting Architecture (OSA), an application can now be as flexible as a set of Lego building blocks. Defining the program's high-level behavior using scripts instead of traditional program code makes possible an unprecedented amount of flexibility, a cause for celebration particularly among in-house developers and developers of custom software. Want to make a change in the way your software works? It's simple to modify the scripts that define the behavior of the objects involved. Want to make a customized solution, using the program's components as building blocks? Easy: just write some new scripts. Want to construct an OpenDoc part from your program? You're already partway there.

Varying degrees of OSA support are open to your application. The OSA gives you the ability to do the following:

- execute scripts previously created with the AppleScript Script Editor
- store compiled scripts and other script values in your program and data files
- directly compile and execute scripts
- decompile existing scripts, for editing
- use embedded scripts to automate your program's handling of Apple events
- enable users to customize and extend your program's capabilities by attaching scripts to objects in the application's domain

PAUL G. SMITH (AppleLink SMITH.PG) is a developer and consultant specializing in intelligent agent software, computer-based communications, and object-oriented programming techniques, currently in his eleventh year of developing for the Macintosh. He divides his time between the offices of Full Moon Software Inc. near Cupertino, California, and his native England, where he runs a small European consultancy called commstalk hq and lives with his wife, Steph, and his cat, Mack. Steph leads a more glamorous life than Paul, having been a dancer in big productions in the London West End and now making glorious hats as a fashion milliner; Mack leads an easier life than Paul and takes a lot more naps.•

Further, the OSA makes it possible for all customizable applications to present a common set of scripting languages and dialects for users to choose from.

This article orients you to the OSA by outlining an OSA-savvy programming structure and then describing techniques you can use to import scripts from the Script Editor, run a script in your application, attach scripts to objects, compile and decompile scripts, route Apple events to scripts, and handle user-interface events. The programming structure and these techniques are demonstrated in the source code for the sample program SimpliFace on this issue's CD. The AppleScript Software Development Toolkit, available from APDA, contains the essential tools for OSA development.

STRUCTURING THE OSA-SAVVY PROGRAM

You need to do some preliminary setup work in your application before you can take full advantage of the services offered by the OSA and make use of scripts. Depending on how your application is already structured, this can mean anything from a slight restructuring to a complete rewrite from the ground up. I'll describe the basic requirements for an OSA-savvy program here and then show you the structure of SimpliFace so that you can see how one looks.

THE BASIC REQUIREMENTS

The first requirement for an OSA-savvy program is that it comply with the Apple event object model. As you probably know, this model sets out a standard way of structuring a program so that it can be controlled from other programs and so that it's scriptable using standard terminology familiar to the user. This model is solution oriented (that's the crucial part) because it concentrates on *what* users do with the application, not on *how* they and the application do it. The articles "Apple Event Objects and You" in *develop* Issue 10 and "Better Apple Event Coding Through Objects" in *develop* Issue 12 provide useful information about the Apple event object model and how to support it in your application. The Apple Event Registry is the essential reference for standard Apple event classes and commands.

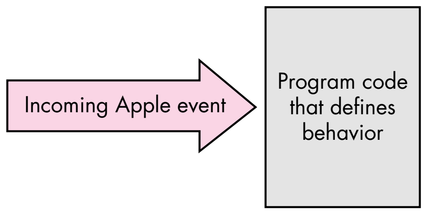
The second requirement (which isn't completely separable from the first) is that the application be fully factored — that is, that it separate the interface from the operations. In a factored program, the actions that result when users choose menu items, click buttons, and so on, generate a sequence of Apple events. When a user-initiated action is dispatched as an Apple event, or when an external program or script sends an Apple event, the program resolves which object the Apple event relates to. It then passes the Apple event to the appropriate handler for that object; this program code is responsible for the object's behavior.

When your application complies with the Apple event object model and is fully factored, and when it publishes its scripting terminology, it's possible to make it attachable — that is, to make it handle and store the data involved in the process of embedding or attaching a script. (I say "embedding a script" when I mean building one in at the program development stage, whereas I refer to "attaching a script" when I mean it's added or modified by the user.) And once your program enables scripts to be attached to objects such as windows, documents, and the application object itself, these scripts can customize the program's handling of object-model Apple events.

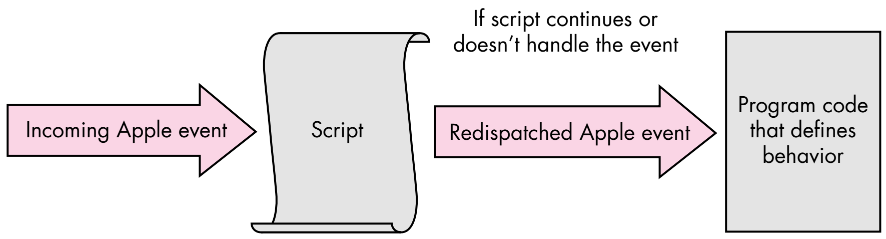
Scripts attached to program objects can affect the behavior of the program and its objects in two cases. In the first case, scripts attached to objects can modify the behavior of those objects when Apple events are resolved and handled. In the second case, scripts attached to user-interface objects like menus and buttons can define the sequence of Apple events that result from user-initiated actions. Both of these mechanisms may have a place in your application. Fortunately, your application's structure doesn't have to change much to allow scripts to customize behavior.

An attachable program can give a compiled script first crack at handling an incoming Apple event instead of passing the event first to the handler (the program code that defines the object's behavior). If the script handles the Apple event, the program code doesn't get called; if the script continues the Apple event (that is, passes the message to the script's parent object) or if it doesn't handle it, the program code gets called as usual, as illustrated in Figure 1. If necessary, the script can modify or add to the original parameters for the Apple event before passing it on to the program code.

For more on handling Apple events, see the description of command handlers on page 241 of the *AppleScript Language Guide.*•



In a program that's not attachable


If script continues or doesn't handle the event
Incoming Apple event
Script
Redispatched Apple event
Program code that defines behavior

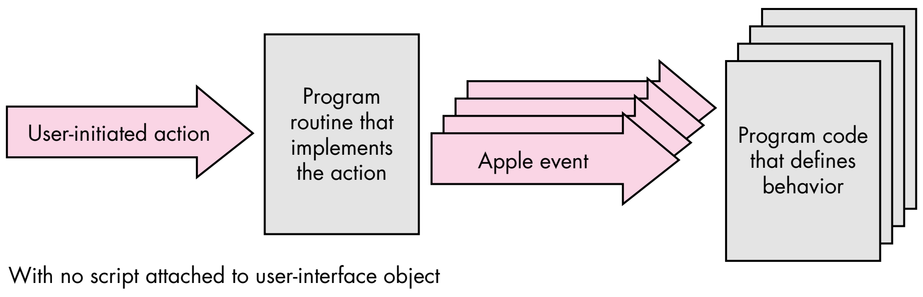
In an attachable program

Figure 1. Routing an Apple event

Thus, attaching scripts to objects can make the operation of your program a great deal more flexible. But you can go even further: instead of generating Apple events by making long-winded calls to the Apple Event Manager in response to user-initiated actions, you can attach scripts to user-interface objects. Selection of one of these objects then results in a script being called; the result of executing the script is that the appropriate Apple events are sent, as illustrated in Figure 2. In the first case, the primary reason the program makes the Apple event calls is so that the action is recordable; in the second case, the script makes the Apple event calls anyway, so that no extra work is required to make the action recordable, and thus the recordability comes for free. The overhead involved in this is minimal (and it may even reduce the bulk of program code); the increased flexibility is massive. It's not even necessary to make these embedded scripts user changeable — that's entirely up to you.

A SAMPLE PROGRAM: SIMPLIFACE

The sample program SimpliFace on this issue's CD demonstrates the principles just outlined. SimpliFace is a basic scriptable and attachable user-interface builder written in MPW C++. SimpliFace constructs scripted windows that can contain text labels (though not editable text) and buttons. It demonstrates many of the features of the OSA APIs, uses a lightweight C++ framework for Apple event object model compliance, suggests a novel approach to a fully factored application, and allows


User-initiated action
Program routine that implements the action
Apple event
Program code that defines behavior
With no script attached to user-interface object

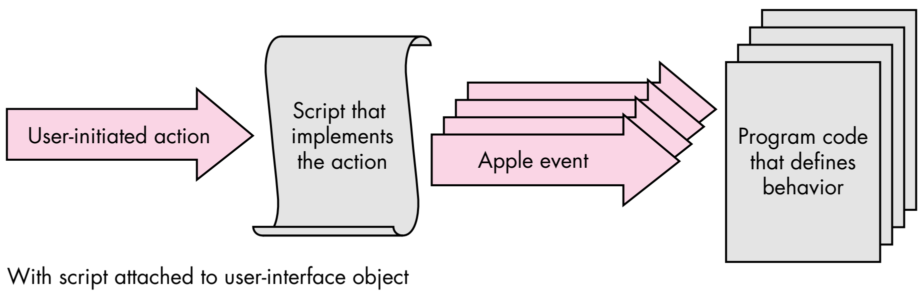

User-initiated action
Script that implements the action
Apple event
Program code that defines behavior
With script attached to user-interface object

Figure 2. Generating Apple events

scripts to be attached to all application objects. SimpliFace has little preprogrammed behavior; virtually everything is defined through scripts supplied by the user.

SimpliFace is built around a rough-and-ready C++ framework, inspired by the one used in the Apple Shared Library Manager's sample applications. I like to use lightweight C++ classes that don't depend on one another too much (thus aiding their reuse), so the program structure isn't as tightly integrated as that of, say, a MacApp program. In the spirit of other Apple sample applications, most of the error handling has been left for later.

Figure 3 illustrates the object containment hierarchy for SimpliFace at run time. There is one application object, which can contain zero or more window objects. Each window object can contain button objects and/or text label objects.

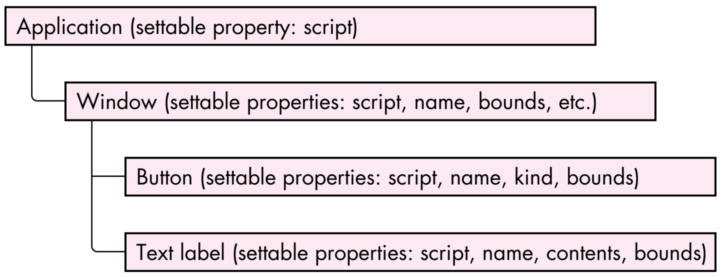

Application (settable property: script)
Window (settable properties: script, name, bounds, etc.)
Button (settable properties: script, name, kind, bounds)
Text label (settable properties: script, name, contents, bounds)

Figure 3. SimpliFace's object containment hierarchy

Figure 4 shows the SimpliFace class hierarchy. All application-domain scriptable objects derive from a TScriptableObject class (see the source file ScriptableObjects.h) that has an attached script and is able to assist with object resolution and Apple event handling. A TObjModelToken class (see ObjModelTokens.h) is defined to manage token resolution and Apple event dispatching; the interaction of these is managed from a set of static functions in the file ObjModelEvents.cp.

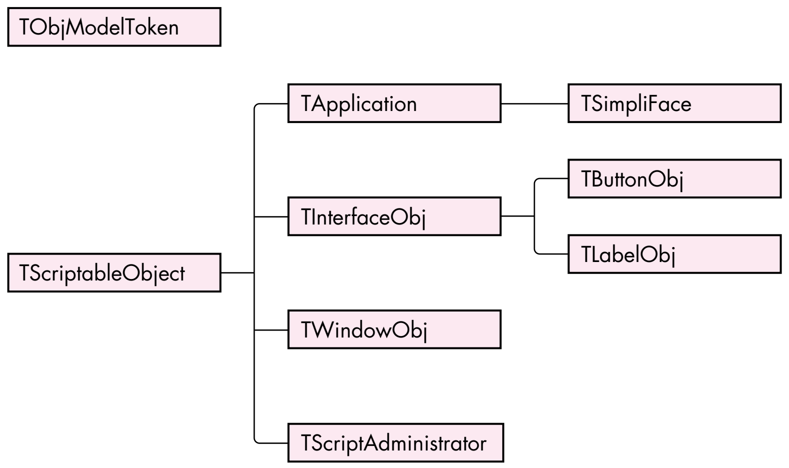


Figure 4. SimpliFace's class hierarchy

The application's behavior is defined in the files Application.cp and SimpliFace.cp (the latter contains the main program function). Outside the program's object containment hierarchy, a separate script administrator class, TScriptAdministrator, is defined. This class is responsible for fetching the script attached to objects and preparing it for execution, and serves to encapsulate the script-handling code. It's implemented in the file ScriptableObjects.cp.

The window, button, and text label objects are created by sending SimpliFace appropriate Apple events. When it receives an Apple event, SimpliFace resolves the object that the event is aimed at (the direct parameter of the event specifies the target object). SimpliFace then dispatches the Apple event (unless it's an Open Application, Get Data, or Set Data event) to the script of the target object. The work for this is handled in the file ObjModelEvents.cp. We'll look in greater detail at how SimpliFace handles an incoming Apple event in the section "Routing Apple Events to Scripts."

Two sample scripts written in the AppleScript language are supplied to demonstrate SimpliFace; you can run these using the Script Editor. One is the startup script that's run whenever SimpliFace is launched, and the other (called Test Simple Window) creates a window that contains two buttons and a text label. Here's the latter script:

```
make new window ¬
   with properties {name:"Tests", bounds:{60, 60, 350, 300}}
set the script of window "Tests" to winScript
open window "Tests"

make new button ¬
   with properties {name:"Quit", kind:standard, bounds:{10, 50, 80, 70}} ¬
   at end of window "Tests"
make new button ¬
   with properties {name:"Hello", kind:standard, bounds:{10, 10, 80, 30}} ¬
   at end of window "Tests"
make new text label ¬
   with properties {name:"Data entry", contents:"I'm a text label!", ¬
   bounds:{90, 80, 280, 130}} ¬
   at end of window "Tests"
```

Open the SimpliFace Dictionary, using the Script Editor, to see more details of the scripting interface.

DEBUGGING SIMPLIFACE

To help debug the Apple event handlers in SimpliFace, I implemented a simple debug transcript tool. A small amount of code (borrowed from MacApp) is used to intercept the MPW <stdio> library. The information thus captured is packaged and sent via a private Apple event to the Debug Transcript program, which you'll find along with the rest of SimpliFace on the CD.

The debugging mechanism is managed in the SimpliFace module DebugTrace. If you open the Debug Transcript program before you run SimpliFace, a lot of information about Apple events received and processed will appear in the transcript window. Apple events are decoded in the Apple event prehandler using the AEPrint routine, part of Jens Alfke's AEGizmos library. You can add to the information displayed simply by adding extra printf statements to the program code.

SimpliFace doesn't store window properties or object scripts on disk, so every time you launch it you need to set up the application script, and you must recreate all windows and window objects. This is facilitated by the automatic loading and execution of the startup script (called SimpliFace Startup) whenever SimpliFace is launched. To prevent this script from running, hold down the Control and Command keys while SimpliFace starts up. The mechanism used to run this script is described under "Running Scripts" later in this article.

TECHNIQUES FOR OSA-SAVVY PROGRAMS

Now that you have a general idea of how to structure an OSA-savvy program, we'll consider the specific techniques your program can use to take advantage of the OSA's services and enjoy the flexibility offered by using scripts. This section describes how to import scripts from the Script Editor, run scripts from within the application, attach scripts to objects, compile and decompile scripts, route Apple events to scripts, and handle user-interface objects. You won't necessarily need to implement all of these techniques in your program, but you should be aware of them so that you can decide how you want to implement scripting.

IMPORTING SCRIPTS FROM THE SCRIPT EDITOR

Unless you have a specialized requirement and want to write your own script editor, you'll get the best mileage by creating and compiling scripts with Apple's Script Editor and then importing these scripts into your application. You have some choices about how to approach importing scripts into your application:

- You can keep the scripts in Script Editor files and load them only when needed. This is the approach SimpliFace takes with its startup script: it looks in the same folder as the application for the file called SimpliFace Startup and loads and executes the script from that file.
- Extending this approach, your program can maintain a folder of script files and look there for named files. For the user, adding a new script file is as easy as dragging it to the folder in the Finder.
- Alternatively, your program can offer an import function that allows the user to select a script file from which a compiled script is to be imported. Your program can then store the compiled script object in any way it wishes, using its own data storage mechanisms.

Before we examine the technique you should use to import scripts, we need to quickly review how the Script Editor and the OSA store scripts. AppleScript can compile scripts in two forms: as contexts, which can contain handlers (for Apple events and user-defined subroutines) and properties, and as ordinary compiled scripts, which can only be executed. The Script Editor always compiles and saves scripts as script contexts. To the OSA, compiled scripts are values and can be stored in variables and manipulated just like numbers, text, lists, or records. When a program passes a script value to the OSA, or when the OSA passes a script value to a program, it's referred to through a special magic cookie called an OSAID. OSAIDs are 32-bit-long integers, and the OSA uses an internal mechanism to map these onto the data they refer to.

The OSA provides a pair of routines that you can use to convert OSAIDs to and from data handles you can save. OSAStore converts an OSAID into an AEDesc (Apple event descriptor), of which the data handle portion can be saved. OSALoad does the opposite, unpacking the contents of the data handle portion of an AEDesc (previously saved using OSAStore) to create a new OSAID. A compiled script context that's been saved using the OSAStore command is contained in resource number 128 of type 'scpt' (kOSAScriptResourceType, the same constant as typeOSAGenericStorage), one of the four resources of a Script Editor compiled script file.

The AEDesc is the basic Apple event data structure, described in *Inside Macintosh: Interapplication Communication*, Chapter 3, and the earlier *Inside Macintosh* Volume VI, Chapter 6.•

SimpliFace's LoadScriptFromFile routine, shown in Listing 1, imports a compiled script from a Script Editor file; you can use it as a model regardless of which of the three approaches outlined above you choose to take. The key tasks undertaken by this routine (apart from locating and opening the resource fork of the script file) are loading the resource handle, putting a reference to it into an AEDesc of type 'scpt' (typeOSAGenericStorage), and calling OSALoad to generate an OSAID that refers to the compiled script.

Listing 1. TScriptAdministrator::LoadScriptFromFile

```
OSAError TScriptAdministrator::LoadScriptFromFile(FSSpec *fileSpec,
                                                OSAID *theScriptID)
{
   short       fileRef = FSpOpenResFile(fileSpec, fsRdPerm);
   OSAError    err = ResError();

   if (err == noErr) {
      Handle   h = Get1Resource(kOSAScriptResourceType, 128);
      if (h != nil) {
         AEDesc   scriptData;
         scriptData.descriptorType = typeOSAGenericStorage;
         scriptData.dataHandle = h;
         err = OSALoad(gScriptingComponent, &scriptData,
                       kOSAModeNull, theScriptID);
         ReleaseResource(scriptData.dataHandle);
      }
      CloseResFile(fileRef);
   }
   return err;
}
```

RUNNING SCRIPTS

Now that you've loaded the script, it can be executed. So how do you run a script in your application? It's easy — all you do is pass the compiled script (which can be a script context or a simple compiled script) to the OSA routine OSAExecute. To show how it's done, here's some code from SimpliFace that executes the startup script:

```
FSSpec      theFileSpec;
OSAID       startupScript = kOSANullScript;

// ... Set up theFileSpec here....
err = LoadScriptFromFile(&theFileSpec, &startupScript);
if (err == noErr && startupScript != kOSANullScript) {
   OSAID       resultID = kOSANullScript;
   err = OSAExecute(gScriptingComponent, startupScript,
                 kOSANullScript, kOSAModeNull, &resultID);
   // ... More code goes here....
}
```

The first parameter to the OSAExecute routine and all other OSA routines is the scripting component instance. To make any OSA calls, you need to have opened a connection to a scripting component (in this case, AppleScript) by means of the OpenDefaultComponent call. This returns a component instance that you can save (in this case, as gScriptingComponent) and pass to future OSA calls.

The second and third parameters to OSAExecute are both OSAIDs referring to compiled scripts. The second parameter refers to the script to be executed and the third parameter refers to the script context in which global variables will be bound (if the script to be executed is a normal compiled script). Script contexts in the AppleScript OSA component are equivalent to script objects in the AppleScript language. Whenever a script object is compiled that contains commands in the body (not inside a handler), these commands are collected into a default run handler (the handler that's executed when the script object is sent the run message). The Script Editor uses this same method to execute scripts. The run handler executes using the context to access and store properties and global variables.

In the above fragment, we supply kOSANullScript for the third parameter because the script we've loaded from the Script Editor file is a script context. If a script context with a run handler is given as the second parameter to OSAExecute, the run handler is extracted from the context and used as the compiled script. In this case, the third parameter passed to OSAExecute is ignored.

The above code from SimpliFace executes a predefined compiled script that sets up the initial state of the program. You can use the same technique to attach scripts directly to menu functions or to buttons in dialog boxes and data entry forms, but I don't recommend that because you can gain more flexibility by generating Apple events from user actions and then letting scripts handle the events. You can also use this technique to extend your application so that scripts are triggered when interesting events occur. For instance, if you were writing a storage management utility you could let the user declare timers that triggered scripts at predefined intervals or at specified times of the day or week to perform backups and disk reorganizations.

ATTACHING SCRIPTS TO OBJECTS

Attaching a script to an application-domain object can be a simple matter of extending the definition of the object to include a script property. The Apple Event

Registry defines a class, property, and Apple event data type for script properties. The constants for all these have the same value: 'scpt' (typeOSAGenericStorage).

SimpliFace demonstrates how a script property can be attached to an object. As noted earlier, all application-domain objects in SimpliFace that have an object-model counterpart are derived from the class TScriptableObject. The definition of this class includes a field of type OSAID called fAttachedScript, referring to the object's attached script. Listing 2 shows the source code of the SetProperty function from the class TScriptableObject.

Listing 2. TScriptableObject::SetProperty

```
OSErr TScriptableObject::SetProperty(DescType propertyID,
                                    const AEDesc *theData)
{
   OSAError      err = errAEEventNotHandled;

   // Used switch statement instead of if statement to allow for
   // future expansion.
   switch (propertyID) {
   case pScript:
      OSAID theValueID = kOSANullScript;
      if (theData->descriptorType == typeChar
            || theData->descriptorType == typeIntlText)
         err = OSACompile(gScriptingComponent, theData,
                     kOSAModeCompileIntoContext, &theValueID);
      else  // If it's not text, we assume the script is compiled.
         err = OSALoad(gScriptingComponent, theData,
                     kOSAModeNull, &theValueID);
      if (err == noErr) {
         if (fAttachedScript != kOSANullScript)
            OSADispose(fAttachedScript);
         fAttachedScript = theValueID;
      }
      break;
   }
   return (OSErr)err;
}
```

You or the user can write an AppleScript script that sets the script property of an object. Here's an example that sets the script property of a window object:

```
tell application "SimpliFace"
   script myWindowScript
      on close
         global numTimesClosed
         set numTimesClosed to numTimesClosed + 1
         continue close
      end close
   end script

   set the script of window "MyWindow" to myWindowScript
end tell
```

If your application is constructed so that parts of the runtime behavior are defined using attached scripts, it becomes feasible to write other scripts that update the application to its latest version. These scripts could even update applications across a network. What a boon this would be to MIS departments!

COMPILING AND EXECUTING SCRIPTS

In some circumstances, your program will need to compile a script itself instead of importing a compiled script from the Script Editor. For instance, the Do Script Apple event lets the scripter supply the script source code as the event's direct parameter. To handle this Apple event, your program needs to compile the source code before it can be executed. You might also want the user to be able to set the script property of an object by supplying the source code instead of a compiled script.

The OSA provides a routine called OSACompile that compiles script source code and returns an OSAID. The SimpliFace function TScriptableObject::SetProperty, shown in Listing 2, uses OSACompile.

SimpliFace uses a different routine, OSACompileExecute, to implement the Do Script Apple event. This is a convenience routine that compiles some script source code and immediately executes it. Listing 3 shows DoScript, the SimpliFace function that handles the Do Script Apple event. In this function, the AEDesc scriptDesc contains the script to be executed. The data type is checked to see if it contains source code; if so, the script is compiled and executed using OSACompileExecute. If a script value result is returned by the OSA, it's coerced to an AEDesc and returned in resultDesc.

Listing 3. TScriptAdministrator::DoScript

```
OSAError TScriptAdministrator::DoScript(AEDesc *scriptDesc,
                                         AEDesc *resultDesc)
{
   OSAError    err = errAEEventNotHandled;

   if (scriptDesc != nil && (scriptDesc->descriptorType == typeChar
               || scriptDesc->descriptorType == typeIntlText)) {
      OSAID resultID = kOSANullScript;
      err = OSACompileExecute(gScriptingComponent, scriptDesc,
               kOSANullScript, kOSAModeAlwaysInteract, &resultID);
      if (err != noErr)
         DumpOSAerrorInfo(gScriptingComponent, err);
      else if (resultID != kOSANullScript)
         err = OSACoerceToDesc(gScriptingComponent, resultID,
               typeWildCard, kOSAModeNull, resultDesc);
      OSADispose(gScriptingComponent, resultID);
   }
   return err;
}
```

Eagle-eyed students of the Apple Event Registry will notice that the DoScript function doesn't implement the other standard form of the Do Script Apple event, which allows the script to be specified by reference to a script file on disk. As discussed earlier in the section "Importing Scripts from the Script Editor," the source code for SimpliFace includes a mechanism to read a script from a script file, so I'll leave it to you to modify TScriptAdministrator::DoScript.

DECOMPILING SCRIPTS

The counterpart of the Set Data Apple event is the Get Data Apple event. An attachable application should allow scripts to get, as well as set, the script property of objects. By default, the script property should be returned as a compiled script, but the definition of the Get Data Apple event permits the caller to request a property as a different data type. If this data type is text, and if the script property of an object is being requested, your program will need to decompile the script.

The function OSAGetSource is used to extract the source code from a compiled script. SimpliFace demonstrates the use of OSAGetSource in the GetProperty function from the class TScriptableObject (see Listing 4). The SimpliFace Get Data handler extracts the desired data type from the Apple event and passes it to the GetProperty function in the parameter wantType; if the caller doesn't specify a data type, typeWildCard is used to signify the default type for the property. If the object has an attached script, GetProperty checks to see what data type is requested by the caller. If the data type is text, the script is decompiled and the source code is returned. Otherwise, the compiled script is returned as the result using the OSAStore function (the converse of OSALoad, discussed earlier).

Listing 4. TScriptableObject::GetProperty

```
OSErr TScriptableObject::GetProperty(DescType propertyID,
                                   DescType wantType, AEDesc *result)
{
   OSErr    err = errAEEventNotHandled;

   switch (propertyID)    // Used to allow for future expansion
   {
   case pScript:
      if (fAttachedScript != kOSANullScript) {
         printf("::GetProperty(): get script as type '%.4s'\n",
                (char*)&wantType);
         if (wantType == typeChar || wantType == typeIntlText) {
            // If caller wants text, we need to decompile the script.
            err = (OSErr)OSAGetSource(gScriptingComponent,
                                   fAttachedScript, wantType, result);
         }
         else {
            if (wantType == typeWildCard)
               wantType = typeOSAGenericStorage;
            err = (OSErr)OSAStore(gScriptingComponent, fAttachedScript,
                                   wantType, kOSAModeNull, result);
         }
      }
      break;
   }
   return err;
}
```

ROUTING APPLE EVENTS TO SCRIPTS

As noted earlier in this article, the way to customize your application's handling of an object-model Apple event is to pass the event to an attached script first, passing it to the program's normal handler for that event only if the script fails to handle or continues the event. Note that this can be done only if the attached script was compiled as a script context; simple compiled scripts can't be used this way. The OSA provides two functions for passing Apple event handling to script contexts: OSAExecuteEvent returns the result of executing the script as an OSAID that you must coerce back to an AEDesc to supply to the reply event, and OSADoEvent automatically puts the result into the reply event.

OSADoEvent had a problem in version 1.0 of AppleScript: it never disposed of the temporary OSAID it used to hold the result of executing the script. This could cause the AppleScript 1.0 component to fail. The recommended workaround was to use OSAExecuteEvent. This problem was fixed in AppleScript 1.1.•

The AppleScript language permits a message to be "continued" — that is, passed to the parent of the script object that's currently handling the message. The AppleScript Continue statement is similar to the Pass statement in HyperTalk. The OSA allows your program to get in on the act when a message is continued, by specifying a resume/dispatch procedure (so called because it lets your program resume handling of a continued event or dispatch an event that isn't handled in the script). To do this, you use the OSASetResumeDispatchProc call. The resume/dispatch procedure takes the same parameters as an Apple event handler and might be your application's default handler for the Apple event in question. It will be called by the OSA during a call to OSAExecuteEvent or OSADoEvent if the script continues the Apple event.

The OSA also allows you to specify another kind of resume/dispatch handling: if instead of specifying an actual Apple event handler for the resume/dispatch procedure you pass the special constant kOSAUseStandardDispatch or kOSADontUsePhac, and if the script continues the handling of an Apple event, it will be dispatched directly to the default Apple event handler for that event, ignoring any special prehandling. To make full use of this facility, you need to implement an Apple event prehandler procedure in your program as well.

The prehandler gets first crack at any incoming Apple event; it's called for all Apple events that are dispatched to the application, except those that have been redispatched by the OSA (assuming you set up resume/dispatch handling as just described). A prehandler procedure is installed by calling the Apple Event Manager function AEInstallSpecialHandler.

To illustrate how Apple events are routed to scripts, let's look in detail at how SimpliFace handles an incoming Apple event.

1. The Apple Event Manager routine AEProcessAppleEvent passes the event to the program's Apple event prehandler procedure.
2. The prehandler procedure tries to resolve the application-domain object that should handle the Apple event, by resolving the event's direct parameter.
3. If an application-domain object is successfully resolved and if it has a script attached, the resume/dispatch mechanism is set up and the Apple event is passed to the script by a call to OSADoEvent. If the script handles the Apple event successfully, we're done with it.
4. If the script doesn't handle the Apple event, the OSA returns the error errAEEventNotHandled, which the prehandler returns as its result. The Apple Event Manager then redispatches the Apple event to the appropriate installed handler.
5. If the script continues the Apple event, it's redispatched by the OSA directly to the installed handler for that Apple event.

6. If there is no script to handle the event (maybe the application-domain object doesn't have an attached script) or the attached script handles but continues the Apple event, the handler you previously installed for the Apple event receives the event and tries to resolve the application-domain object that should handle it.
7. If an application-domain object is successfully resolved, it's asked to handle the Apple event, implementing the object's standard behavior.

The prehandler routine from SimpliFace is shown in Listing 5. The routine starts by extracting the Apple event class and ID from the event record. It then checks to see if the attached-script behavior should be ignored: SimpliFace doesn't pass the Open Application, Get Data, and Set Data Apple events to attached scripts. The routine then tries to resolve the direct object of the Apple event, creating a token object. If no token is resolved, a token that refers to the application object is created. A global script administrator object is then asked to locate and return the attached script; unless there was no attached script, the Apple event is passed to the script by calling the SimpliFace routine ExecuteEventInContext, which in turn calls OSADoEvent. If the script fails to handle the Apple event, or if an error occurs, the error is returned via the Apple Event Manager.

HANDLING USER-INTERFACE EVENTS

User-interface events, such as mouse clicks, keystrokes, and menu selections, are handled in a special way in SimpliFace. This allows SimpliFace to delegate the behavior of these user-interface objects to their attached scripts. User-initiated mouse and keyboard actions are intercepted by a routine in SimpliFace.cp called TSimpliFace::HandleEvent. This routine, in conjunction with a group of support routines, packages and dispatches the user-initiated event as a system event. This special kind of Apple event is dispatched to the script of the object the user clicked in, if there is one, or the current window if not. If there's a mouse click in a text field or button, the script for the field or button (if there is one) gets the event.

If any runtime error occurs in a script while an event is being handled, or if the message is continued out of the last handler that caught it, or if an event isn't handled at all, the normal behavior for that event takes place in the program code. For instance, when a SimpliFace window is closed (either by a script's sending a Close event or by the user's clicking in the window's close box), a Close event is dispatched to the window's script (if it has one). If this script handles the event and returns without error, the Close event goes no further and the window stays open. If the script fails to handle the event or continues the Close event, the window is closed by the default Apple event handler in the C++ program code.

FURTHER OSA SUPPORT

Now that you have a grasp of the basic requirements of an OSA-savvy program and know the techniques to import scripts from the Script Editor, run those scripts, attach scripts to objects, compile and decompile scripts, route Apple events to scripts, and handle user-interface events, you may want to go even further with your support of the OSA. There are two issues in particular that you may want to address: how to handle global variables so that variables can be shared between scripts, and how to allow scripts to share libraries of subroutine handlers so that the application object's attached script behaves something like HyperCard's stack script.

I've completed another version of SimpliFace that implements these features, and it may be the subject of a future article in *develop* — if you clamor loudly enough for it.

Listing 5. StdAEvtPreHandler

```
static pascal OSErr StdAEvtPreHandler(AppleEvent *theEvent, AppleEvent *theReply, long theRefCon)
{
   OSAError         err = errAEEventNotHandled;
   AEEventClass     theEvtClass;
   AEEventID        theEvtID;
   DescType         theType;
   Size             theSize;
   AEDesc           directParam, theTokenDesc;
   OSAID            theScriptID = kOSANullScript;
   objModelTokenPtr theToken = nil;
   Boolean          madeAppToken = false;

   theTokenDesc.descriptorType = typeNull;
   theTokenDesc.dataHandle = nil;
   // Extract the class and ID of the event from the Apple event.
   AEGetAttributePtr(theEvent, keyEventClassAttr, typeType, &theType, (Ptr) &theEvtClass,
                    sizeof(theEvtClass), &theSize);
   AEGetAttributePtr(theEvent, keyEventIDAttr, typeType, &theType, (Ptr) &theEvtID,
                    sizeof(theEvtID), &theSize);
   if ((theEvtClass == kCoreEventClass && theEvtID != kAEOpenApplication)
         || (theEvtClass == kAECoreSuite && theEvtID != kAESetData && theEvtID != kAEGetData)
         || (theEvtClass == kSignature && theEvtID == kAESystemEvent)
         || (theEvtClass == kASAppleScriptSuite && theEvtID == kASSubroutineEvent)) {
      // Above test skips the events we don't want to be scriptable.
      err = AEGetParamDesc(theEvent, keyDirectObject, typeWildCard, &directParam);
      if (err == noErr) {
         err = AEResolve(&directParam, kAEIDoMinimum, &theTokenDesc);
         AEDisposeDesc(&directParam);
         if (err == noErr)
            theToken = ObjModelTokenFromDesc(&theTokenDesc);
      }
      if (err != noErr || theToken == nil) {
         err = MakeAppToken((TObjModelToken**)&theToken);
         madeAppToken = (err == noErr);
      }
      if (theToken != nil)
         theScriptID = gScriptAdministrator->GetAttachedScript(theToken->GetTokenObj());
      if (theScriptID != kOSANullScript) // Pass to script.
         err = ExecuteEventInContext(theEvent, theReply, theRefCon, theScriptID,
               kOSAUseStandardDispatch, kOSADontUsePhac);
      else
         err = errAEEventNotHandled;
      if (theToken != nil)
         gScriptAdministrator->ReleaseAttachedScript(theToken->GetTokenObj());
      AEDisposeToken(&theTokenDesc);
      if (madeAppToken == true)  // Will be executed only
         delete theToken;        // if token was made locally.
   }
   return (OSErr)err;
}
```

Meanwhile, I’ve left as an exercise for you a couple of other enhancements to SimpliFace: implementing editable text fields and making all window and button kinds and object properties work. Roll up your sleeves and have a go at it.

RECOMMENDED READING

- “Apple Event Objects and You” by Richard Clark, *develop* Issue 10.
- “Better Apple Event Coding Through Objects” by Eric M. Berdahl, *develop* Issue 12.
- *Inside Macintosh: Interapplication Communication* (Addison-Wesley, 1993), Chapters 3–10, or *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 6.
- *Apple Event Registry: Standard Suites*, available on this issue’s CD or in print from APDA.
- The AppleScript Suite, available on this issue’s CD or in the AppleScript Software Development Toolkit from APDA.
- *AppleScript Language Guide* (Addison-Wesley, 1993). Also in the AppleScript Software Development Toolkit.

Thanks to our technical reviewers Jens Alfke, Warren Harris, Ron Karr, and Jeroen Schalk.•

9 Ways to Improve Your Mind In Your Spare Time

DEVELOPER UNIVERSITY

Self-Paced Courses from Apple Developer University bring you a full range of technical training in your own home, at your own pace.

- Macintosh Programming Fundamentals
- Intermediate Macintosh Application Programming
- Programmer’s Introduction to RISC and PowerPC
- PowerTalk Templates
- Object-Oriented Fundamentals
- AppleTalk for Programmers
- Apple Events/AppleScript Programming Tutorial
- Programming with MPW
- QuickTime Programming Tutorial

To order the self-paced classes, contact APDA at 1-800-282-2732; for Developer University class schedules contact the Registrar at (408) 974-4897.

GRAPHICAL TRUFFLES

The Debugging Version of QuickDraw GX

PETE ("LUKE") ALEXANDER

By now, many of you have installed one of the beta versions of QuickDraw GX onto your Macintosh — and possibly by the time you read this, QuickDraw GX Software Developer's Kit version 1.0 will be available from APDA. Maybe you've played with the various sample applications and are now ready to work on your first QuickDraw GX application. Perhaps you've even read my article in *develop* Issue 15, "Getting Started With QuickDraw GX." In this column, I'll talk about something I referred to briefly in that article: the two versions of QuickDraw GX's combined graphics and layout portions, and how to take advantage of the debugging version during the development of your QuickDraw GX–based application. Along the way I'll update you on a few changes since I wrote the article.

THE EXTENSIONS OF QUICKDRAW GX

The QuickDraw GX system extension comes in two flavors: a nondebugging and a debugging version. When you run the QuickDraw GX installer script, the nondebugging version is installed, including the complete QuickDraw GX system. The nondebugging version is lean and mean, so it's significantly faster than the debugging version; it performs quite a bit less error checking than the debugging version.

The debugging version of the extension provides extensive error checking and other debugging amenities. When developing a QuickDraw GX application, you should use this version to shake out the bugs. The debugging extension is in the DEBUG Init folder and is named "GXGraphics (debug)" in version 1.0 (it used to be named aSecretGraphics.debug); just drag it into your System Folder and reboot. As your system starts up you'll see the debugging extension's icon displayed before the QuickDraw GX icon.

An explanation of what's really going on here may help clarify things (and it has changed): There are actually three extensions within the QuickDraw GX extension — one for graphics and layout, one for printing, and one for the Finder printing extension. The QuickDraw GX extension knows, if "GXGraphics (debug)" has already loaded, to use that extension instead of the nondebugging version of the graphics and layout extension. (Note that since the debugging extension must load before the nondebugging version, you should not change the debugging extension's name.)

THE ADVANTAGES OF THE DEBUGGING VERSION DURING DEVELOPMENT

Let's look at some differences between the two extensions, and specifically how to take advantage of the debugging version during the development of your QuickDraw GX application.

Notices, warnings, and errors. With the debugging version of the extension, you can get three types of information about drawing problems: notices, warnings, and errors. For a complete list of these, look at the graphics errors.h interface file. The many notices, warnings, and errors defined between **#ifdef debugging** and **#endif** in that file are available only with the debugging version. You'll need to #define **debugging** in your application to take advantage of them. (Make sure **debugging** is not defined when you build your final version.)

With the nondebugging version, notices aren't available at all, and the list of errors and warnings you need to respond to in your application consists only of those relatively few that lie outside **#ifdef debugging** and **#endif** in the graphics errors.h file. Your application must be set up to handle these errors and

PETE ("LUKE") ALEXANDER has been providing developer support for QuickDraw GX ever since it was way up in the air. He's happy it's making its final approach, and he's hoping it lands smack dab in the middle of your software. As a glider pilot, Luke knows how important control is — and with QuickDraw GX, you'll be able to maneuver your software into spaces you never thought possible. Since QuickDraw GX can help you do lots of great graphics stunts that you used to have to ask him about, Luke is about to soar off into the wild blue yonder and do some stunts of his own. Soon he'll be ready for the preflight check that will launch his sabbatical faster than QuickDraw GX handles two-byte text. For many weeks he'll be thinking about nothing but white sand beaches, white puffy clouds, and white-capped mountains. He'll lie on his back and watch the sway of the trees until they stop reminding him of the swashes of L's. So if you see him somewhere in Montana, Utah, Nevada, California, or Idaho, be sure to say hello — but try not to use any words that have a G and an X in them.•

warnings, which in general indicate that the QuickDraw GX system could not honor your application's request. For example:

```
shape_is_nil
size_of_path_exceeds_implementation_limit
picture_index_out_of_range
```

The debugging version checks for errors that you're likely to run into while developing your application, such as passing a negative pen size to GXSetShapePen or a curve to GXGetGlyphMetrics. The nondebugging version doesn't check for these types of errors; it assumes you've already shaken them out of the code.

Validation routines. The GXSetValidation and GXValidateShape routines are available only in the debugging version. These routines allow your application to tell whether it's passing valid parameters into a QuickDraw GX function, to validate the contents of all QuickDraw GX objects (such as a shape, style, and ink) before their use, and to validate the QuickDraw GX memory your application is using.

Speed optimizations. The nondebugging version has optimizations for speed built in — not only fewer error checks but also inline functions. The debugging version doesn't optimize for speed. This shouldn't have any impact on your application development except that there's not a one-to-one correspondence between stack crawls using the debugging and nondebugging versions. A performance analysis of your QuickDraw GX application should only be done *without* the "GXGraphics (debug)" file in your Extensions folder.

GraphicsBug. The GraphicsBug debugging tool allows you to explore the contents of any QuickDraw GX object to make sure it contains the correct information. Another change from before is that GraphicsBug is available in both the debugging and nondebugging versions. This ability to spy on an object's contents is important to your application development because otherwise you could only access information in objects by making API calls, which would be very tedious during debugging.

There's a slight advantage when using GraphicsBug with the debugging version: heaps are listed by name rather than by hex address in the Heaps menu.

Memory. The debugging version's memory blocks in the QuickDraw GX heap are a bit bigger to help detect errors when your application writes over the end of a block: all the blocks end with the same signature, 'grfx'.

Special MacsBug messages. The debugging version generates MacsBug messages that are intended solely for the consumption of the Apple engineers in unusual circumstances. (One of my favorites is "Curious if this ever happens.") If we did our jobs right, you should never see one of these messages. In case you do, however, we're really interested in hearing what caused you to receive it; please let us know at AppleLink APPLE.BUGS.

CLEANING UP

You should design your application to take advantage of the extensive capabilities of the debugging extension, but turn those capabilities off when you create your final shipping application, to improve its performance. For example, calling GXValidateShape with the nondebugging extension installed will only result in a jump and return (that is, it will be a no-op). This is a wonderful method for testing the QuickDraw GX dispatcher, but it doesn't help the performance of your application.

In the final compile of your shipping application, you'll most likely want to remove all calls to validation routines, posting of notices, and extra warnings and errors available only in the debugging version. One approach would be to have various compilation flags associated with pairs of **#ifdef** and **#endif** to turn these features on and off.

For more information, see my article in Issue 15 if you haven't already — or just dig into the QuickDraw GX documentation. Enjoy your journey into the QuickDraw GX world!

RELATED READING

- "Getting Started With QuickDraw GX" by Pete ("Luke") Alexander, *develop* Issue 15.
- "Print Hints: Looking Ahead to QuickDraw GX" by Pete ("Luke") Alexander, *develop* Issue 13.

Thanks to Hugo Ayala, Cary Clark, and Herb Derby for reviewing this column.•

Exploiting Graphics Speed on the Power Macintosh

The new QuickDraw on the PowerPC platform substantially improves graphics performance. A study comparing the performance of QuickDraw and custom blitters on the Power Macintosh and 680x0-based machines provides information you can use to ensure that the user benefits from those improvements. Further analysis, detailing where CopyBits spends its time, leads to an implementation strategy for applications that demand the fastest possible graphics.

KONSTANTIN OTHMER, SHANNON HOLLAND, AND BRIAN COX

Understanding the motivation for and consequences of the changes to QuickDraw on the Power Macintosh can help you write faster applications. This article presents studies that show QuickDraw as one of the most speed-critical parts of the Macintosh Operating System together with studies that break down how applications spend CPU time. Knowing how much time applications actually spend in various system routines will help you develop a strategy for writing applications that perform well on both the Power Macintosh and 680x0-based machines.

In porting QuickDraw to the PowerPC™ platform, Apple took advantage of the opportunity to make some changes. We'll detail these changes and their consequences for writing code. With that foundation, we'll move on to an in-depth discussion comparing the QuickDraw CopyBits routine with custom blitters. The goal is to write applications using routines that result in the fastest possible graphics performance on both platforms — PowerPC and 680x0 — as well as on machines equipped with graphics accelerators such as the new Apple Macintosh Display Card 24 AC. Sample code on this issue's CD demonstrates a method of timing blitter routines so that your application can use the fastest routine at run time.

HOW SPEED-CRITICAL IS QUICKDRAW?

Most of the Macintosh Operating System is written in 680x0 assembly language. In order to reach time-to-market goals for the Power Macintosh, Apple had to focus porting efforts on the most speed-critical parts of the system, so a study was conducted to profile system usage of several common applications. System usage

KONSTANTIN OTHMER, SHANNON HOLLAND, AND BRIAN COX, who are always ready to explore new avenues in software development for the QuickDraw team, have finally hit the nail on the head. Their secret is high-tech equipment and proper delegation of work. Alternating between periods of sleep and contemplation, they use telepathic communication to transmit source code to each new team member, who can then look forward to many days of compile cycles on a trusty Macintosh Plus (providing our authors with even more time for sleep and contemplation). The team is on the lookout for new labor-saving devices. Donations are welcome — comfortable couches to make room for future expansion would be particularly appreciated.•

depended largely on the operations performed in particular applications, but many applications showed similar patterns.

Figure 1 is based on a subset of the study. It turns out that most applications spend from 50% to 95% of their time in system code, with many spending more than 80%. Figure 2 shows the percentage of total CPU time spent in the most frequently called system routines for typical applications and for a pointer-based application (one that avoids using handles).

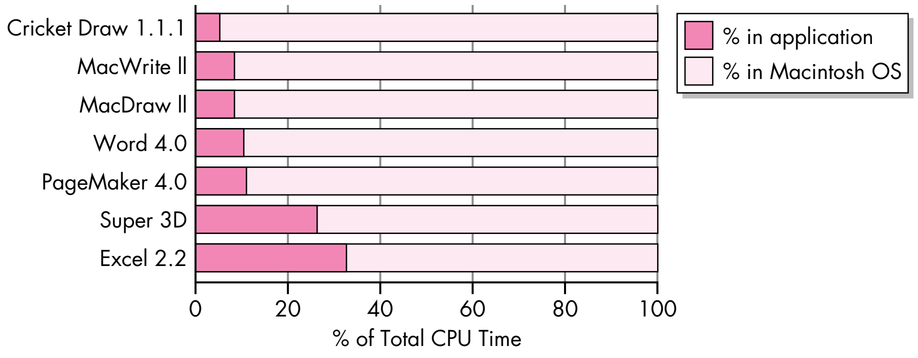


Figure 1. CPU time breakdown: application versus system

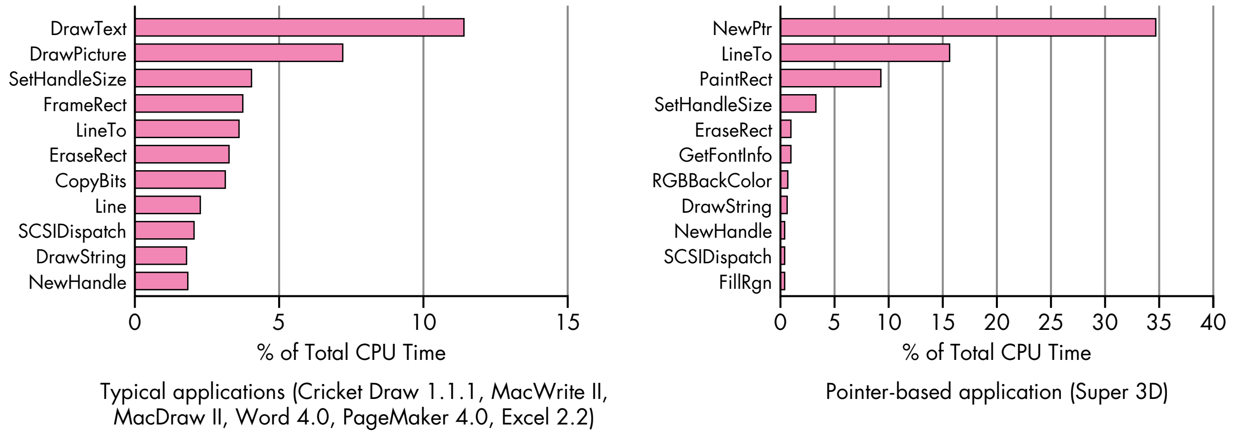


Figure 2. System routine usage

The data made it clear that QuickDraw was one of the most critical components of Apple's porting efforts. This article discusses QuickDraw version 1.3.5, which was developed to run on the PowerPC platform. The new QuickDraw is based on QuickDraw version 1.3.0, the most recent version of QuickDraw running on the Macintosh Quadra, but with some changes (see the section "What's Different With Version 1.3.5?"). The new version, written in C, was compiled for the Power Macintosh as QuickDraw version 1.3.5 and shipped with the new machines. The new QuickDraw C code can also be compiled for 680x0-based machines and will be available in future software releases.

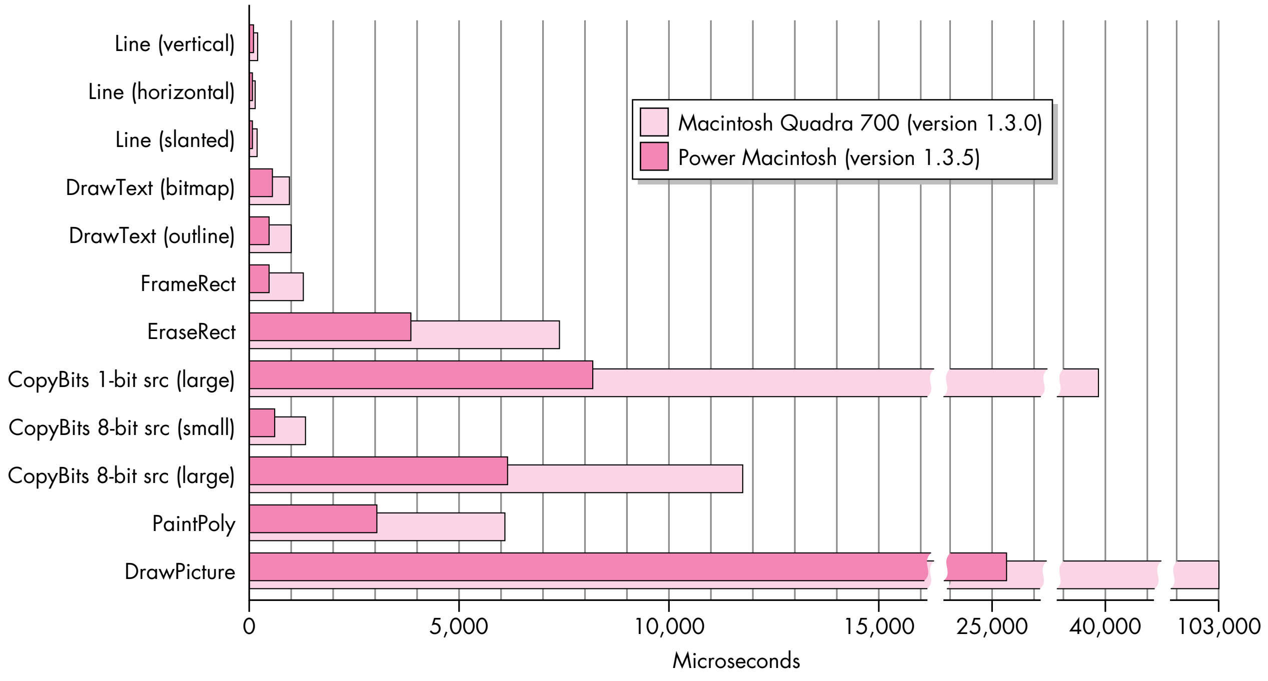
The graphics speed comparisons made in this article compare the following:

- QuickDraw version 1.3.0 or other 680x0 code running on a 680x0-based Macintosh (usually a Macintosh Quadra)

- QuickDraw version 1.3.0 or other 680x0 code running through the emulator on a Power Macintosh
- QuickDraw version 1.3.5 or other PowerPC code running on a Power Macintosh

TAKING ADVANTAGE OF THE SPEED

Figure 3 compares times of various QuickDraw routines for version 1.3.0 running on a Macintosh Quadra and version 1.3.5 running on a Power Macintosh — there's no question that the new QuickDraw routines run faster. However, published surveys comparing the speed of 680x0-based machines to the Power Macintosh haven't always shown the dramatic results indicated by Figure 3. This is partly because some operations offer greater increased speed than others, so depending on which operations an application uses heavily, overall speed varies. A second important factor is that the applications surveyed are often emulated applications.



Note: Times are to an 8-bit screen.

Figure 3. Comparing QuickDraw version 1.3.0 to version 1.3.5

Emulated applications are those written for 680x0-based machines that run through the emulator on the Power Macintosh (see "Making the Leap to PowerPC," *develop* Issue 16). These applications don't benefit fully from the PowerPC platform, because an application that spends 80% of its time in system code on 680x0-based machines, when emulated on a Power Macintosh, spends substantially more time in the application. In general, completely emulated application code runs at about half the speed of a Macintosh Quadra 700. Those same applications, when recompiled as PowerPC code, usually run four or five times faster than on a Macintosh Quadra; code that makes extensive use of floating point may be 20 times or more faster. However, emulated graphics-intensive code, assuming it uses QuickDraw, is substantially faster on a Power Macintosh than on a 680x0-based Macintosh because of the increased speed of QuickDraw 1.3.5.

Clearly, to take full advantage of QuickDraw version 1.3.5, you need to write your applications for the Power Macintosh in PowerPC code. Beyond that general strategy, developing awesome applications for the PowerPC platform means figuring out how to harness all that CPU power — how to take advantage of the speed. For example, the high speed of QuickDraw version 1.3.5 allows you to do high-quality animations. Figure 3 shows that you can now do twice as many (or more) CopyBits operations per second, which means that animations such as zooming, scrolling, and window dragging (leave this one to Apple) can be done in real time without being chunky or annoying. Text drawing is also much faster, so interactive word wrapping while positioning objects in text is easy to do and looks better than it would on a 680x0-based Macintosh. Overall, it's an open field for developers.

Tips for increasing the speed of PowerPC code are given in this issue's Balance of Power column.•

Although this article focuses on QuickDraw, of course there are other, nongraphical, ways of harnessing the power of the PowerPC processor. Floating point–intensive applications benefit tremendously from the speed of the new processor.

The Graphing Calculator desk accessory that ships with the Power Macintosh is an excellent example of harnessing CPU power for both the user interface and computation-bound part of an application. As a floating point–intensive application, Graphing Calculator benefits from the speed of the PowerPC processor. The user interface has a number of nice touches, such as live scrolling, live zooming, and interactive formula and graph manipulation.•

WHAT'S DIFFERENT WITH VERSION 1.3.5?

In the porting of QuickDraw to the PowerPC platform, many algorithms were rethought and reimplemented. The result is slightly different (and we hope better!) behavior. This section outlines some changes to keep in mind when you're writing code.

QDERROR

QuickDraw version 1.3.0 didn't do a very good job of setting and clearing QDError. In version 1.3.5, every call sets QDError (which can cause problems for applications that assume QDError will be preserved across most simple calls, like SetRect). In some cases, version 1.3.0 jumps to SysError if there isn't enough memory; version 1.3.5 returns an error in QDError instead. This is usually an improvement, but it can lead to strange behavior for applications that depend on SysError being invoked. For example, some applications might put up a dialog asking the user to increase the application partition size if QuickDraw invokes SysError. Since QuickDraw version 1.3.5 doesn't invoke SysError (returning a QDError instead), the application code that puts up the dialog isn't triggered, so the user doesn't know to increase the memory and the application might fail by not drawing anything. In choosing to always set QDError, Apple chose the lesser of two evils.

MATCHING COLOR TABLES

QuickDraw version 1.3.0 uses the color table of the pixMap for the current GDevice, not the color table of the destination pixMap, to map colors to the destination pixMap. QuickDraw version 1.3.5 sets up a surrogate GDevice to make sure that the the destination pixMap's and the GDevice's color tables always match. This may cause problems for applications that relied on undefined behavior when the color tables didn't match or for applications that were getting the right results by luck under QuickDraw version 1.3.0. Again, Apple chose the lesser of two evils, and added the surrogate device (known as the skank device). When QuickDraw is forced to set up the skank device, the application pays a slight performance penalty. Also, if you do operations such as index-to-color when your color tables don't match, and then later use that color in a drawing, you won't necessarily draw with the index you expect. The easiest cure: use GWorlds!

For more information on QDError, GDevices, pixMaps, and color tables, see *Inside Macintosh: Imaging With QuickDraw* or *Inside Macintosh* Volume V.•

TRANSFER MODES

There's no way to pass the transfer space (the bit depth at which transfer occurs) when doing transfer modes in QuickDraw. (QuickDraw GX remedies this shortcoming.) So if you're using an arithmetic mode from 8-bit to 16-bit, there are no guarantees whether the transfer will occur at 5 bits per component (16-bit), 8 bits per component (32-bit), or 16 bits per component (as in the 8-bit color table). It turns out that most arithmetic modes in QuickDraw version 1.3.0 perform the transfer operation at a resolution of 16 bits per color, while version 1.3.5 does most operations at a resolution of 8 bits per color. This sometimes causes slight cosmetic differences.

DITHERING

The dithering algorithm in QuickDraw version 1.3.5 is slightly different. This makes it a nightmare to programmatically determine whether version 1.3.5 is generating the same results as version 1.3.0, but visually the results are nearly identical.

STRETCHING AND SHRINKING IMAGES

The way CopyBits stretches and shrinks images for nonintegral ratios has been improved in QuickDraw version 1.3.5 (integral ratios still produce the same results). The advantage of this new algorithm is that it's symmetrical: if you stretch an image and then shrink it back to the original size, the same pixels that were replicated in the stretch are combined in the shrink.

The disadvantage of the new algorithm is that some applications stretch or shrink without knowing it (the classic off-by-one error, resulting in a destination rectangle that's smaller or larger than the source rectangle by one pixel). Such applications may now drop (or replicate) a different scan line. This can cause slight cosmetic blemishes in some applications.

UNEXPECTED REGISTER CONTENTS

Because QuickDraw version 1.3.5 runs PowerPC code, all emulated 680x0 registers are preserved across calls. Thus, applications that expect the contents of volatile registers (A0, A1, D0, D1, D2) to contain specific values on exit from a QuickDraw call will break. (Conversely, don't rely on 680x0 registers being preserved, either!) There's one exception: for compatibility with some existing applications, CopyBits always sets D0 to 0.

PATCHING

Patching any QuickDraw version 1.3.5 routine with 680x0 code degrades performance because of mode-switch overhead time. A mode switch occurs when a 680x0 caller is calling PowerPC code, or vice versa. 680x0 patches on ShieldCursor are particularly expensive because ShieldCursor is called by nearly every QuickDraw drawing routine.

For more information on the Mixed Mode Manager and mode switching, see "Making the Leap to PowerPC" in *develop* Issue 16.•

DISABLED ACCELERATOR CARDS
QuickDraw version 1.3.0 makes calls through many low-level (undocumented) vectors. Version 1.3.5 doesn't use these trap vectors, which disables most accelerator cards. Of course, the frame buffer on these cards continues to work.

THE COPYBITS/CUSTOM BLITTER RACE

A favorite developer sport is complaining about how slow CopyBits is and writing custom blit loops to replace it. A favorite sport among QuickDraw engineers is working all night trying to speed up some part of CopyBits. This competition is healthy so long as speed-critical applications call the faster code.

"Blitter" informally refers to any routine that moves memory, usually visual information to the screen or an off-screen buffer; the operation is called a "blit." These terms derive from the PDP-10 block transfer instruction, BLT.•

Through the years, Apple engineers have yearned for a way to get a substantial lead in the race with the speed-hungry special-case developer. The answer lies in the Power Macintosh: raw 680x0 code runs substantially slower through the emulator, while QuickDraw version 1.3.5 CopyBits takes advantage of the lightning-fast RISC processor.

Figure 4 compares various ways of moving the memory used by an 8-bit, 32-by-32 pixMap and an 8-bit, 400-by-400 pixMap to the screen. BlockMove gives a baseline: the typical amount of time needed to move that much raw memory. The 680x0 blitter is a custom blitter written for 680x0-based machines and emulated on the Power Macintosh. The PowerPC blitter is a custom blitter written for the Power Macintosh (it can't be run on a 680x0 machine).

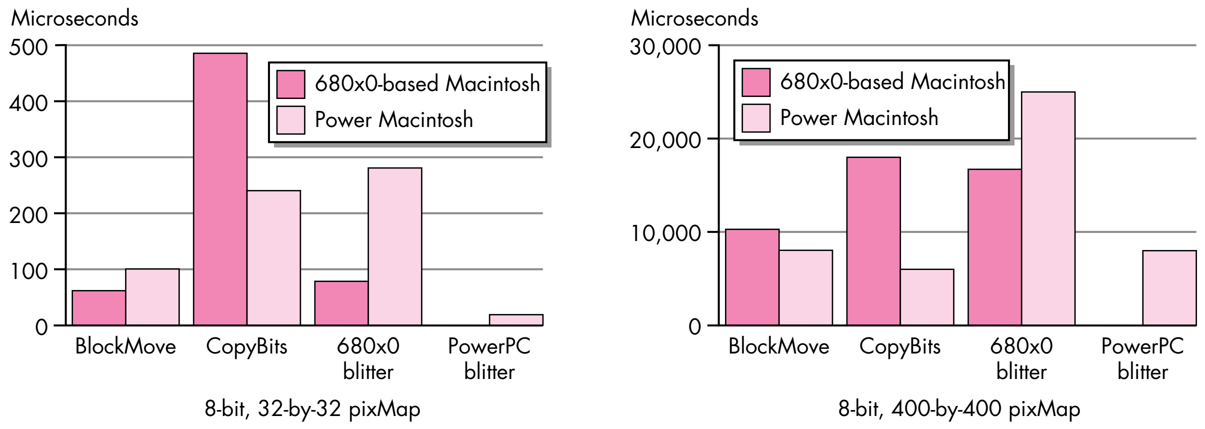


Figure 4. CopyBits versus custom blitters

As you can see, the custom PowerPC blitters beat QuickDraw's CopyBits for the small image hands down for both 680x0-based machines and the Power Macintosh. (With the small image the constant overhead of CopyBits has a big impact on the overall time.) However, the 680x0 blitter is much slower than CopyBits on a Power Macintosh. This is due to the overhead of emulation.

The interesting case is the custom PowerPC blitter versus CopyBits for the large image on the Power Macintosh. Here CopyBits wins. This is due to optimizations that CopyBits has for large images that the PowerPC blitter doesn't have. In this case, CopyBits is also faster than BlockMove, because of optimizations in CopyBits for the PowerPC processor's frame buffer (which has a 64-bit data path). BlockMove is optimized for copying to main memory, so it's slower when copying to the frame buffer. (This is why the PowerPC blitter is faster than BlockMove for the small image.) If you compare BlockMove and CopyBits using an off-screen pixMap as the destination, you discover that BlockMove is faster.

For maximum performance of emulated applications, the emulator treats BlockMove as a special case.•

The design of a frame buffer can have a great impact on overall blit speed. These times were measured on the on-board video for the Macintosh Quadra and a fast processor-direct slot video card for the Power Macintosh. If you install a NuBus™ frame buffer on both machines and do a similar comparison, you find that the difference in times is less. That's because NuBus is the bottleneck for the copy operation. The situation changes radically, however, if the NuBus card is accelerated. Then only calls to CopyBits get the acceleration; custom blit loops are still bottlenecked by NuBus transfer rates.

Most of the comparisons in this section compare raw memory-moving power. While QuickDraw is efficient at stretching bits, it's very inefficient at large indexed shrinks. The problem is that CopyBits looks at every pixel and preserves the highest index value. (This was done so that when icons are shrunk, they don't inadvertently go to solid white.) For a shrink by a factor of four, this means that CopyBits is looking at 16 times too much data.•

REDUCING QUICKDRAW OVERHEAD

There are two aspects to any given QuickDraw operation: setup and actual drawing. Much of the time saved when an application uses a custom blit loop instead of CopyBits is a consequence of avoiding the overhead of QuickDraw's setup. While QuickDraw has extremely efficient blit routines, its downfall is that it has no idea how it's going to be called from one time to the next, so it has to do all the setup every time it's called. (See "Drawing in GWorlds for Speed and Versatility" in *develop* Issue 10 for a discussion of QuickDraw's setup.)

An application knows exactly how many of what it's drawing to where, so it can do the setup for many operations once at the beginning, use custom blitters to do the drawing, and then restore everything to its previous condition at the end, thus eliminating much of the setup time. This is where you get the biggest gains when writing your own blitters. On large operations, the overhead is relatively small, so you don't gain much with custom routines. Small operations are often dominated by setup time, so a custom routine can improve performance significantly.

Figure 5 compares setup time to total time for two CopyBits operations. Both are a copy of a 32-by-32, 8-bit, off-screen pixMap to the screen (no stretching or shrinking, long aligned). The difference is that in the first CopyBits call, the color tables match and in the second call they don't match (the first case is faster because there's no need to invoke a pixel translation loop). Figure 6 shows the same two tests as Figure 5, but this time the pixMaps being copied are 400-by-400. If you look carefully, you can see that the setup time remained almost the same, but the proportion between setup time and total time has changed drastically.

In general, the setup time on the Power Macintosh is minimal, since the setup is computation-intensive and doesn't depend on memory access. Remember that setup time is constant — it remains the same no matter how much data is being copied.

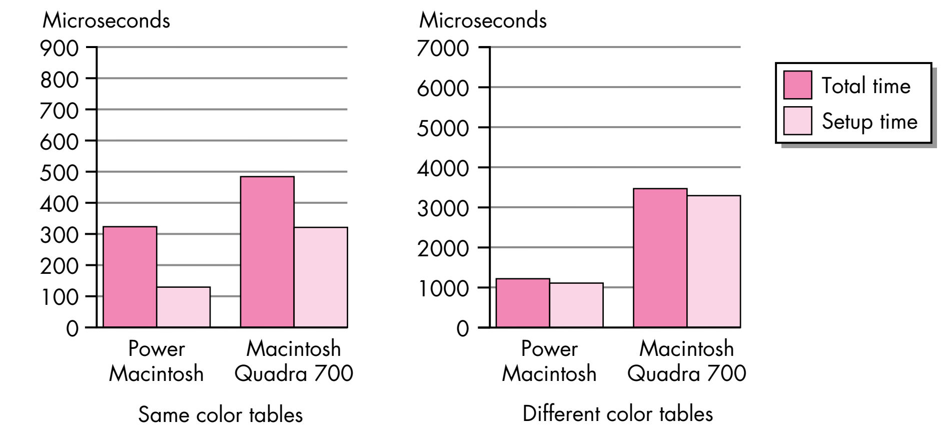


Figure 5. CopyBits setup time to total time for a small copy

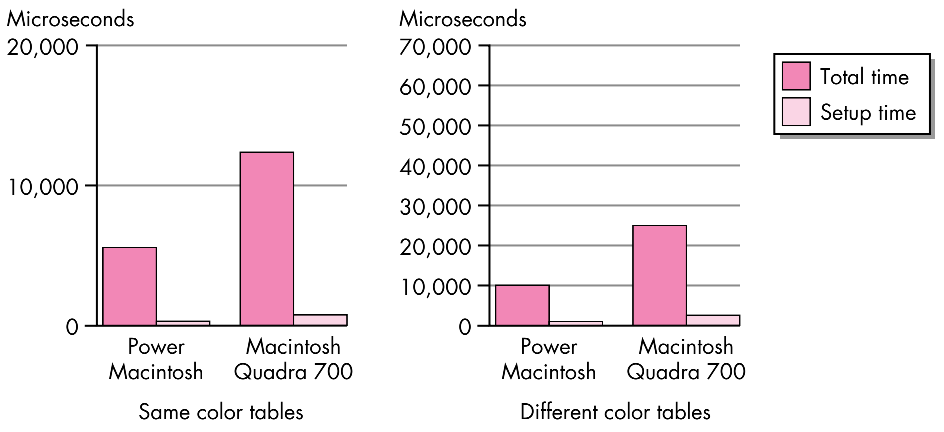


Figure 6. CopyBits setup time to total time for a large copy

Therefore, the relative efficiency of CopyBits depends on the amount of data being copied.

The systems compared in Figures 5 and 6 are a Power Macintosh 8100/80 running QuickDraw version 1.3.5 and a Macintosh Quadra 700 running QuickDraw version 1.3.0. These comparisons show that QuickDraw blit times can vary greatly across different machines and different versions of QuickDraw.

QuickDraw GX uses caches extensively to keep intermediate results. This allows part of the overhead to be short-circuited when a similar operation is performed multiple times.•

Accelerator vendors use a number of different strategies for boosting QuickDraw's performance. The Macintosh 8•24 GC card attempted to accelerate entire operations, while most third-party accelerators just concentrate on the blits. These cards often use custom chips to substantially increase the speed of writing to memory; you're still forced to pay for the setup time, but the blit time decreases substantially.

The upshot of this is that you're only guaranteed the best results if you profile the candidates and pick a winner at run time. This is the topic of the following section.

STRATEGY FOR SPEED-CRITICAL APPLICATIONS

For applications in which speed is critical, you want to run as fast as possible on every machine. The easiest way to do this is to time the system code and any custom code and use the faster version, perhaps even on a call by call basis. By comparing the speed of a custom implementation with the Toolbox implementation and picking the faster one at application initialization time, applications can automatically take advantage of hardware accelerators when they exist, or highly specialized custom blit loops when required. Of course, you would use this strategy only when speed is extremely important. While developing your application, you should always try to use system calls when they're available before reinventing (a sometimes square) wheel.

Listing 1 shows two routines, TimeBlitProc and BestBlitter, that compare CopyBits with a custom blitter and return the address of the faster routine. (The code is also on this issue's CD.) Writing the custom blitter is left as an exercise for the reader.

BestBlitter takes a pointer to a BlitProc, a PixMapHandle, and a source and destination rectangle and returns the address of the faster routine — the custom BlitProc or CopyBits. It assumes that the destination rectangle is for the current graphics port and current GDevice. For the sake of simplicity, the mode is assumed to be srcCopy and there's no mask region.

BestBlitter gets the address of CopyBits (factoring out trap dispatch overhead if running on a 680x0-based Macintosh, as you might want to do in your speed-critical loops) and calls TimeBlitProc to get the time taken by each of the calls. If the

Listing 1. Timing routines

```
#include <Timer.h>
#include <FixMath.h>
#include <Traps.h>
#if powerc
   extern QDGlobals qd;
#endif

// Decide how many microseconds represent a "meaningful" difference.
#define     kMeaningfulDiff     0
#define     ABS(x)              ((x < 0)? (-x) : (x))

unsigned long TimeBlitProc(BlitProcPtr theBlitProc, BitMapPtr srcBits,
   BitMapPtr dstBits, Rect *srcRect, Rect *dstRect, short mode,
   RgnHandle mask)
{
   UnsignedWide   startMicroSec, endMicroSec;

   Microseconds(&startMicroSec);
   (*theBlitProc)(srcBits, dstBits, srcRect, dstRect, mode, mask);
   Microseconds(&endMicroSec);
   // WideSubtract isn't defined for 680x0-based machines; however, a
   // version is included on the CD.
   WideSubtract((wide *) &endMicroSec, (wide *) &startMicroSec);
   return endMicroSec.lo;
}
```

(continued on next page)

Listing 1. Timing routines *(continued)*

```
BlitProcPtr BestBlitter(BlitProcPtr customBlitProc,
   PixMapHandle srcPixHandle, Rect *srcRect, Rect *dstRect)
{
   unsigned long  customBitsTime, copyBitsTime;
   long           leDifference;
   PixMapHandle   portPixMap;
   BlitProcPtr    copyBitsPtr;
   Str255         numStr;

   // To factor out the trap overhead, get the trap address for CopyBits.
   // PowerPC can get the address of the shared library routine directly.
   // By getting the address of the library routine like this, we don't
   // need to worry about calling CopyBits through CallUniversalProc.
#if powerc
   copyBitsPtr = (BlitProcPtr) &CopyBits;
#else
   copyBitsPtr = (BlitProcPtr) GetToolTrapAddress(_CopyBits);
#endif

   portPixMap = ((CGrafPtr) qd.thePort)->portPixMap;

   // Normally, it's not necessary to lock a pixMap or its pixels before
   // calling CopyBits. But in this case, we're calling TimeBlitProc,
   // which could hit the Segment Loader and cause memory to move. So we
   // lock the pixMap handles before dereferencing them here.
   HLock((Handle) portPixMap);
   LockPixels(portPixMap);
   copyBitsTime = TimeBlitProc(copyBitsPtr,
      (BitMapPtr) *srcPixHandle, (BitMapPtr) *portPixMap, srcRect,
      dstRect, srcCopy, nil);
   customBitsTime = TimeBlitProc(customBlitProc,
      (BitMapPtr) *srcPixHandle, (BitMapPtr) *portPixMap, srcRect,
       dstRect, srcCopy, nil);
   UnlockPixels(portPixMap);
   HUnlock((Handle) portPixMap);
   leDifference = (long)(customBitsTime - copyBitsTime);
   if (ABS(leDifference) > kMeaningfulDiff && leDifference < 0)
      return customBlitProc;
   else
      return copyBitsPtr;
}
```

difference is enough to be meaningful (more than a few microseconds) and favors the new BlitProc, BestBlitter returns a pointer to the BlitProc; otherwise, it returns a pointer to CopyBits.

The actual timing is done by TimeBlitProc, which assumes that the current graphics port and GDevice are set up and ready for copying. TimeBlitProc takes a pointer to the BlitProc to be timed and a list of arguments expected by CopyBits.

We've made the assumption that the caller has flushed or loaded the caches appropriately for the test. In comparing the routines, it would be unfair to one routine if it had to spend time loading the data into the cache and the other routine didn't! FlushInstructionCache and FlushDataCache are no longer available for applications written in PowerPC code, so it's up to the caller to decide whether to test these BlitProcs cached or uncached. (See "Here's the Cache" for a discussion of caching on the Power Macintosh.) In any case, TimeBlitProc assumes that the caches are already in the proper state.

Since caching is such a hardware-specific operation and can have both very obvious and subtle effects on the execution of your code, it's hard to predict how different cache architectures will affect your performance. In general, if you try to optimize for smaller caches, you'll achieve better overall performance across a range of platforms. To be completely fair, TimeBlitProc should also disable interrupts. If file sharing comes in to work on a background copy in the middle of the timing, that blit loop will appear to be really slow compared to the uninterrupted time.

HERE'S THE CACHE

The traps FlushInstructionCache and FlushDataCache were originally created to give direct control over the instruction and data caches on 68040-based Macintosh Quadra models. These two traps are very closely tied to the 68040 processor, both conceptually and in their implementation. The PowerPC 601 chip has a unified cache — a single 32K cache for both data and instructions. Rather than trying to contort the definition of the two existing traps to make sense on the PowerPC processor, Apple engineers asked why you need to flush caches in the first place. The new cache-management strategies are intended to be better abstracted, less dependent on a specific processor, and definitely forward compatible.

Following are the four main reasons you might want to flush the caches and how they've been (or need to be) addressed on the Power Macintosh.

Generate code dynamically.
Normally, to execute some data as instructions, you need to flush the caches. On the Power Macintosh, you call the new system routine MakeDataExecutable, passing the base address and the length of the data to be executed. (This routine doesn't exist — even in an undocumented form — on the 680x0-based machines, so to flush instructions in the data cache, you need to call FlushInstructionCache and FlushDataCache.)

Ensure that the data shared by other hardware is actually written.
For example, memory that's shared by a coprocessor has to be accessible when the other processor needs to read it. To address this problem, the PowerPC-family architecture includes a type of "bus snooping." Whenever someone wants to read an address that's represented in the cache, the cache is flushed automatically before the data is returned. This way, you don't need to anticipate all the different ways the cache can get out of sync.

Ensure that data gets written to memory in the correct order.
For example, if you're writing to the screen, make sure the title bar gets written before the contents. A caching mechanism could screw up this ordering, so to ensure the proper ordering, the data cache must be flushed between writes. Screen memory is marked as *write-through*, which sends the data to the cache and on through to the screen memory. Writes for write-through memory are as slow as for uncached memory. The benefit is that reads from write-through memory can still take advantage of the cache. This feature is present on the 68040 Macintosh and remains unchanged on the Power Macintosh.

Ensure that timing data you get when you compare two similar routines hasn't been distorted by the caching mechanism.
Unfortunately, you're out of luck here. There's no officially sanctioned method for doing this. But there are some techniques you can use to get around the caching.

If you anticipate that your procedure will usually have its data cached when it's called, compare the routines for the cached condition. Simply call the routines twice and time only the second call.

To compare the routines for the noncached case, you can "flush" the cache by reading every byte in a 32K buffer. Not only is this ugly, but it's not even guaranteed to work with future machines (such as the PowerPC 603, which goes back to using separate data and instruction caches). And even on the 601 chip, this would flush only the on-chip cache; it wouldn't necessarily flush the much larger, but slightly slower, external cache.

TimeBlitProc calls a new trap, Microseconds, that takes a pointer to an UnsignedWide (two longs) and fills it with the number of microseconds that have elapsed since the system was booted. It calls Microseconds before and after the call to the BlitProc that was passed in, calls WideSubtract to get the delta, and returns the low-order 32 bits of the subtraction. This assumes that the elapsed time will fit into an unsigned long, or that the BlitProc will take less than 71 minutes to complete!

OFF AND RUNNING

The Power Macintosh provides a new range of computing power for the next generation of the Macintosh line. The challenge for Apple is converting from a largely 680x0 assembly code base to PowerPC-code system services and substantially improving the user experience in the process. The challenge for application developers is inventing new uses for all the power provided by RISC, and designing creative user interface elements that take advantage of the horsepower.

Use the studies presented here as a guide to writing graphics-intensive applications that shine on both platforms. By using techniques such as runtime determination of the most efficient routines, you can guarantee that your application will get the most out of the system today and in the future.

Thanks to our technical reviewers Lew Cirne, Jean-Charles Mourey, Guillermo Ortiz, and Andy Stadler; to Kate Cremer for generating the graphs; and to Tom Adams, Becky Hammaker, Marianne Hsiung, Mac MacDougall, and David Searles for conducting the application evaluations.•

We're looking for developers to join us in creating the next generation of innovative computing products and services.

Apple Developer Group

With a market of more than 13 million customers worldwide, Apple continues to outpace the industry in unit growth. As Macintosh application units continue to grow, Apple offers developers the best platform for sustained growth.

At Apple, we know that our success depends on providing our developers with the programs and resources needed to create great products and services. Join our community of developers and receive answers to technical questions; stay informed of special Apple events; and communicate with Apple, fellow developers, and customers.

For more information on the support and services offered through Apple's developer programs contact:

Apple Computer, Inc.
Developer Support Center
20525 Mariani Avenue, M/S 303-2T
Cupertino, CA 95014
(408) 974-4897

DAVE EVANS

BALANCE OF POWER

Enhancing PowerPC Native Speed

When you convert your applications to native PowerPC code, they run lightning fast. To get the most out of RISC processors, however, you need to pay close attention to your code structure and execution. Fast code is no longer measured solely by an instruction timing table. The PowerPC 601 processor includes pipelining, multi-issue and speculative execution, branch prediction, and a set associative cache. All these things make it hard to know what code will run fastest on a Power Macintosh.

Writing tight code for the PowerPC processor isn't hard, especially with a good optimizing compiler to help you. In this column I'll pass on some of what I've learned about tuning PowerPC code. There are gotchas and coding habits to avoid, and there are techniques for squeezing the most from your speed-critical native code. For a good introduction to RISC pipelining and related concepts that appear in this column, see "Making the Leap to PowerPC" in Issue 16.

MEASURING YOUR SPEED

The power of RISC lies in the ability to execute one or more instructions every machine clock cycle, but RISC processors can do this only in the best of circumstances. At their worst they're as slow as CISC processors. The following loop, for example, averages only one calculation every 2.8 cycles:

```
float a[], b[], c[], d, e;
for (i=0; i < gArraySize; i++) {
   e = b[i] + c[i] / d;
   a[i] = MySubroutine(b[i], e);
}
```

By restructuring the code and using other techniques from this column, you can make significant improvements. This next loop generates the same result, yet averages one calculation every 1.9 cycles — about 50% faster.

```
reciprocalD = 1 / d;
for (i=0; i < gArraySize; i+=2) {
   float result, localB, localC, localE;
   float result2, localB2, localC2, localE2;

   localB = b[i];
   localC = c[i];
   localB2 = b[i+1];
   localC2 = c[i+1];

   localE = localB + (localC * reciprocalD);
   localE2 = localB2 + (localC2 * reciprocalD);
   InlineSubroutine(&result, localB, localE);
   InlineSubroutine(&result2, localB2, localE2);

   a[i] = result;
   a[i+1] = result2;
}
```

The rest of this column explains the techniques I just used for that speed gain. They include expanding loops, scoping local variables, using inline routines, and using faster math operations.

UNDERSTANDING YOUR COMPILER

Your compiler is your best friend, and you should try your hardest to understand its point of view. You should understand how it looks at your code and what assumptions and optimizations it's allowed to make. The more you empathize with your compiler, the more you'll recognize opportunities for optimization.

An optimizing compiler reorders instructions to improve speed. Executing your code line by line usually isn't optimal, because the processor stalls to wait for dependent instructions. The compiler tries to move instructions that are independent into the stall points. For example, consider this code:

```
first = input * numerator;
second = first / denominator;
output = second + adjustment;
```

DAVE EVANS may be able to tune PowerPC code for Apple, but for the last year he's been repeatedly thwarted when tuning his 1978 Harley-Davidson XLCH motorcycle. Fixing engine stalls, poor timing, and rough starts proved difficult, but he was recently rewarded with the guttural purr of a well-tuned Harley.•

Code examples were compiled with the PPCC compiler using the speed optimization option, and then run on a Power Macintosh 6100/66 for profiling. A PowerPC 601 microsecond timing library is provided on this issue's CD.•

Each line depends on the previous line's result, and the compiler will be hard pressed to keep the pipeline full of useful work. This simple example could cause 46 stalled cycles on the PowerPC 601, so the compiler will look at other nearby code for independent instructions to move into the stall points.

EXPANDING YOUR LOOPS

Loops are often your most speed-critical code, and you can improve their performance in several ways. Loop expanding is one of the simplest methods. The idea is to perform more than one independent operation in a loop, so that the compiler can reorder more work in the pipeline and thus prevent the processor from stalling.

For example, in this loop there's too little work to keep the processor busy:

```
float a[], b[], c[], d;
for (i=0; i < multipleOfThree; i++) {
   a[i] = b[i] + c[i] * d;
}
```

If we know the data always occurs in certain sized increments, we can do more steps in each iteration, as in the following:

```
for (i=0; i < multipleOfThree; i+=3) {
   a[i] = b[i] + c[i] * d;
   a[i+1] = b[i+1] + c[i+1] * d;
   a[i+2] = b[i+2] + c[i+2] * d;
}
```

On a CISC processor the second loop wouldn't be much faster, but on the PowerPC processor the second loop is twice as fast as the first. This is because the compiler can schedule independent instructions to keep the pipeline constantly moving. (If the data doesn't occur in nice increments, you can still expand the loop; just add a small loop at the end to handle the extra iterations.)

Be careful not to expand a loop too much, though. Very large loops won't fit in the cache, causing cache misses for each iteration. In addition, the larger a loop gets, the less work can be done entirely in registers. Expand too much and the compiler will have to use *memory* to store intermediate results, outweighing your marginal gains. Besides, you get the biggest gains from the first few expansions.

SCOPING YOUR VARIABLES

If you're new to RISC, you'll be impressed by the number of registers available on the PowerPC chip — 32 general registers and 32 floating-point registers. By having so many, the processor can often avoid slow memory operations. Your compiler will take advantage of this when it can, but you can help it by carefully scoping your variables and using lots of local variables.

The "scope" of a variable is the area of code in which it is valid. Your compiler examines the scope of each variable when it schedules registers, and your code can provide valuable information about the usage of each variable. Here's an example:

```
for (i=0; i < gArraySize; i++) {
   a[i] = MyFirstRoutine(b[i], c[i]);
   b[i] = MySecondRoutine(a[i], c[i]);
}
```

In this loop, the global variable gArraySize is scoped for the whole program. Because we call a subroutine in the loop, the compiler can't tell if gArraySize will change during each iteration. Since the subroutine might modify gArraySize, the compiler has to be conservative. It will reload gArraySize from memory on every iteration, and it won't optimize the loop any further. This is wastefully slow.

On the other hand, if we use a *local* variable, we tell the compiler that gArraySize and c[i] won't be modified and that it's all right to just keep them handy in registers. In addition, we can store data as temporary variables scoped only within the loop. This tells the compiler how we intend to use the data, so that the compiler can use free registers and discard them after the loop. Here's what this would look like:

```
arraySize = gArraySize;
for (i=0; i < arraySize; i++) {
   float localC;
   localC = c[i];
   a[i] = MyFirstRoutine(b[i], localC);
   b[i] = MySecondRoutine(a[i], localC);
}
```

These minor changes give the compiler more information about the data, in this instance accelerating the resulting code by 25%.

STYLING YOUR CODE

Be wary of code that looks complicated. If each line of source code contains complicated dereferences and typecasting, chances are the object code has wasteful memory instructions and inefficient register usage. A great compiler might optimize well anyway, but don't count on it. Judicious use of temporary variables (as mentioned above) will help the compiler understand exactly what you're doing — plus your code will be easier to read.

Excessive memory dereferencing is a problem exacerbated by the heavy use of handles on the Macintosh. Code often contains double memory dereferences, which is important when memory can move. But when you can guarantee that memory *won't* move, use a local pointer, so that you only dereference a handle once. This saves load instructions and allows further optimizations.

Casting data types is usually a free operation — you're just telling the compiler that you know you're copying seemingly incompatible data. But it's *not* free if the data types have different bit sizes, which adds conversion instructions. Again, avoid this by using local variables for the commonly casted data.

I've heard many times that branches are "free" on the PowerPC processor. It's true that often the pipeline can keep moving even though a branch is encountered, because the branch execution unit will try to resolve branches very early in the pipeline or will predict the direction of the branch. Still, the more subroutines you have, the less your compiler will be able to reorder and intelligently schedule instructions. Keep speed-critical code together, so that more of it can be pipelined and the compiler can schedule your registers better. Use inline routines for short operations, as I did in the improved version of the first example loop in this column.

KNOWING YOUR PROCESSOR

As with all processors, the PowerPC chip has performance tradeoffs you should know about. Some are processor model specific. For example, the PowerPC 601 has 32K of cache, while the 603 has 16K split evenly into an instruction cache and a data cache. But in general you should know about floating-point performance and the virtues of memory alignment.

Floating-point multiplication is wicked fast — up to *nine times* the speed of integer multiplication. Use floating-point multiplication if you can. Floating-point division takes 17 times as long, so when possible multiply by a reciprocal instead of dividing.

Memory accesses go fastest if addressed on 64-bit memory boundaries. Accesses to unaligned data stall while the processor loads different words and then shifts and splices them. For example, be sure to align floating-point data to 64-bit boundaries, or you'll stall for four cycles while the processor loads 32-bit halves with two 64-bit accesses.

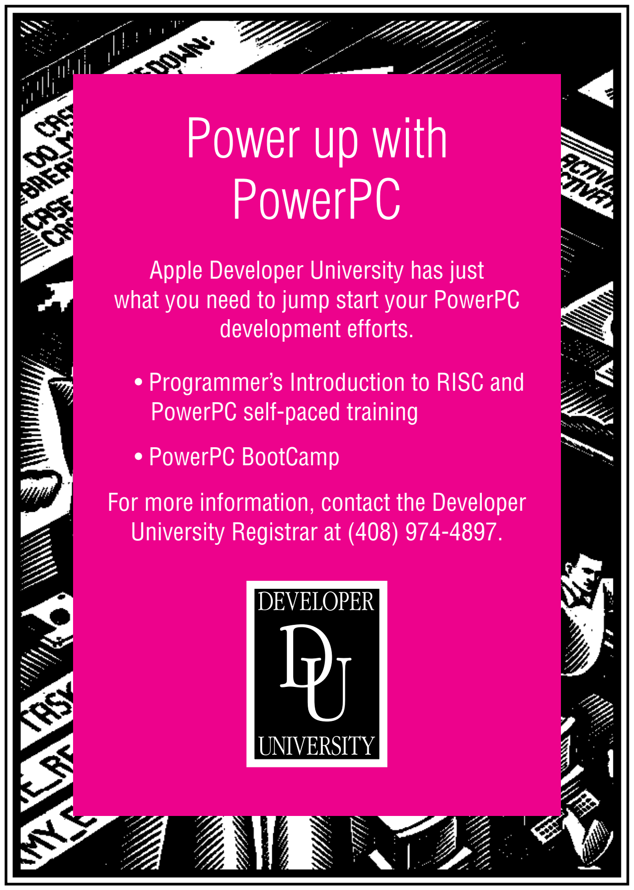
MAKING THE DIFFERENCE

Native PowerPC code runs really fast, so in many cases you don't need to worry about tweaking its performance at all. For your speed-critical code, though, these tips I've given you can make the difference between "too slow" and "fast enough."

RECOMMENDED READING

- *High-Performance Computing* by Kevin Dowd (O'Reilly & Associates, Inc., 1993).
- *High-Performance Computer Architecture* by Harold S. Stone (Addison-Wesley, 1993).
- *PowerPC 601 RISC Microprocessor User's Manual* (Motorola, 1993).

Thanks to Nick Kledzik, Andy Nicholas, and Dave Radcliffe for reviewing this column.•



Power up with PowerPC

Apple Developer University has just what you need to jump start your PowerPC development efforts.

- Programmer's Introduction to RISC and PowerPC self-paced training
- PowerPC BootCamp

For more information, contact the Developer University Registrar at (408) 974-4897.

DEVELOPER DU UNIVERSITY

Displaying Hierarchical Lists

Much of the data you manage on a Macintosh has a hierarchical nature. This article shows how your application can provide a clear, coherent data organization with a user-controlled display by using classic linked lists, storing the list data in handles, and displaying it with the List Manager. A triangular button mechanism, similar to that used in the Finder to open and close folders in list views, lets the user decide how much data to view on the screen.

MARTIN MINOW

If your application makes its hierarchical data accessible but not overwhelming, it will have an advantage over applications that provide only two alternatives — “throw it all on the screen” or “hide everything.” You can find many examples of flexible organization: The Finder presents file and folder information in a variety of display formats revealing more or less information according to the user’s own desires. Document-based applications such as NewsWatcher provide a hierarchy of text information. The JMP statistical application provides buttons that reveal more detailed information about an analysis. Programming languages such as Frontier allow the programmer to display as much of a module’s code as is needed.

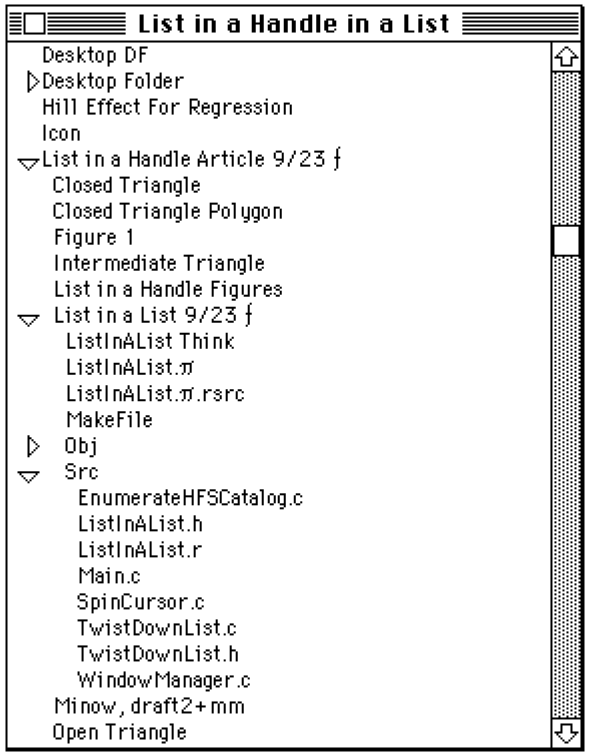
This article shows how to do the following:

- store hierarchical data in a linked list along with the information needed to display it properly
- extend the List Manager by storing a button object in a List Manager cell that controls how the data hierarchy is displayed
- build the data hierarchy and display it
- let the user manipulate the buttons to view more or less of the data

The accompanying code on this issue’s CD includes a sample library that stores data, displays it, and manages the buttons, and a simple application that uses the library. The techniques described in the article are appropriate for displaying and organizing moderate amounts of data; they’re less useful for static data or large amounts of data and are inefficient with small amounts of data.

MARTIN MINOW (AppleLink MINOW, Internet minow@apple.com) is an aged, wrinkled hacker who, having determined that it’s far too late to whine about his receding hairline, instead takes perverse delight in informing his young colleagues at excessive length how much better programming was when you had to punch out your programs, one machine word after another, on oily paper tape, back in the good old days when hex digits were KSNJFL, words were 40 bits long, and a supercomputer had 1024 of them. In real life, he works at Apple’s Developer Support Center, drinks beer, and runs marathons.•

Figure 1 shows a Finder-like window that was created with this library; the files displayed are from the sample library. I've called the library *TwistDown* to emphasize how the display acts when you click the buttons. The Finder development team calls the buttons *triangular buttons*.



```
  Desktop DF
▷ Desktop Folder
  Hill Effect For Regression
  Icon
▽ List in a Handle Article 9/23 ƒ
    Closed Triangle
    Closed Triangle Polygon
    Figure 1
    Intermediate Triangle
    List in a Handle Figures
▽   List in a List 9/23 ƒ
      ListInAList Think
      ListInAList.π
      ListInAList.π.rsrc
      MakeFile
▷     Obj
▽     Src
        EnumerateHFSCatalog.c
        ListInAList.h
        ListInAList.r
        Main.c
        SpinCursor.c
        TwistDownList.c
        TwistDownList.h
        WindowManager.c
    Minow, draft2+mm
    Open Triangle
```

Figure 1. Window created with TwistDown library

STORING DATA IN A LINKED LIST

There isn't much in this article on linked lists or handles; I assume you struggled with "classical" list processing when you learned to program and have done enough programming on the Macintosh to understand how handle-based storage operates. Keep in mind that two kinds of lists are discussed here: linked lists and Macintosh List Manager display lists. To keep confusion to a minimum, *element* refers to a component of a linked list and *cell* refers to a component of a List Manager list. Note also that linked lists contain the data, while the List Manager list only controls the appearance of that data. A good understanding of the List Manager is needed to follow the code examples later in the article. (For details on the List Manager, see *Inside Macintosh: More Macintosh Toolbox*, or *Inside Macintosh* Volume IV.)

In the context of this article, a list element is a chunk of data, a few flags, and two linkages. This section discusses the linkages, which connect list elements into sequences and hierarchies, and the creation and disposal of list elements. The flags, which simplify formatting the data, are discussed later in the section "Controlling Data Appearance."

Figure 2 illustrates the linkages that connect list elements. The important thing to remember about the hierarchical linked lists we're using is that any element may have a *successor* — the sibling element that follows it — and a *descendant* — a child element that begins a lower level of the hierarchy. For example, a document outline has a sequence of chapters (siblings) and each chapter has, as its descendants, a sequence of sections.

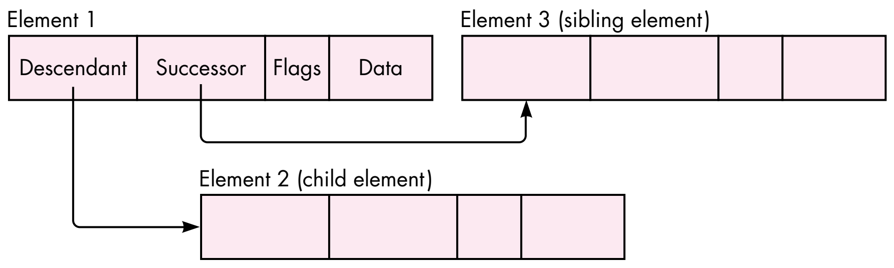


Figure 2. List element organization

In the sample code, each list element is stored in a handle. This allows the Memory Manager to reorganize memory to store data efficiently. However, don't forget that the application program is responsible for disposing of data that's no longer needed.

Two library functions manage the list elements: MakeTwistDownElement creates an element and connects it to the list hierarchy and DisposeTwistDownElement deletes an element along with its descendants and successors.

CREATING A LIST ELEMENT

Listing 1 shows the definition of our element structure, TwistDownRecord. Each field of this structure will be explained as it's encountered in the sample code.

Listing 1. TwistDownRecord

```
struct TwistDownRecord {
   struct TwistDownRecord **nextElement; /* -> successor element      */
   struct TwistDownRecord **subElement;  /* -> descendant element     */
   short                  indentLevel;   /* Indentation depth         */
   unsigned short         flag;          /* Control flags             */
   unsigned short         dataLength;    /* Actual data length        */
   unsigned char          data[1];       /* Data to display           */
};
typedef struct TwistDownRecord  TwistDownRecord,
      *TwistDownPtr, **TwistDownHandle;
```

MakeTwistDownElement (Listing 2) is called with the data to store in the element and a handle to its predecessor. The predecessor is either NULL or the previous (elder) sibling element. For example, when creating element 3 in the list shown in Figure 2, the previous element is element 1.

TwistDownRecord is a variable-length structure and NewHandle creates an instance that's large enough to hold the caller's data record. The structure definition, however, contains one bit of trickery that's required by the ANSI C standard — it must specify at least one byte for the data[] placeholder. This is why the parameter to NewHandle adjusts the handle size to eliminate the extra byte.

DISPOSING OF A LIST ELEMENT

The DisposeTwistDownHandle function disposes of a list element and then disposes of its descendant and successor lists. To dispose of an entire list, call this function with the first list element.

A simplified version of DisposeTwistDownHandle is shown in Listing 3. The library implementation allows the application developer to specify a function that's called when disposing of each element in the list. This is needed if an application has to store complex structures — themselves containing Ptr or Handle references — in a TwistDownHandle.

Listing 2. MakeTwistDownElement

```
OSErr MakeTwistDownElement(TwistDownHandle  previousElement,
                           short            indentLevel,
                           unsigned short   dataLength,
                           Ptr              dataPtr,
                           TwistDownHandle  *result)
{
   TwistDownHandle   twistDownHandle;

   twistDownHandle = (TwistDownHandle) NewHandle(sizeof(TwistDownRecord)
                  - sizeof(unsigned char) + dataLength);
   *result = twistDownHandle;
   if (twistDownHandle != NULL) {
      if (previousElement != NULL)
         (**previousElement).nextElement = twistDownHandle;
      (**twistDownHandle).nextElement = NULL;
      (**twistDownHandle).subElement = NULL;
      (**twistDownHandle).indentLevel = indentLevel;
      (**twistDownHandle).flag = 0;
      (**twistDownHandle).dataLength = dataLength;
      if (dataPtr != NULL)
         BlockMove(dataPtr, (**twistDownHandle).data, dataLength);
   }
   return (MemError());
}
```

Listing 3. DisposeTwistDownHandle

```
void DisposeTwistDownHandle(TwistDownHandle twistDownHandle)
{
   TwistDownHandle   nextElement, subElement;

   while (twistDownHandle != NULL) {
      nextElement = (**twistDownHandle).nextElement;
      subElement = (**twistDownHandle).subElement;
      DisposeHandle((Handle) twistDownHandle);
      if (subElement != NULL)
         DisposeTwistDownHandle(subElement);
      twistDownHandle = nextElement;
   }
}
```

Note that DisposeTwistDownHandle is a recursive function; it calls itself to dispose of the descendants of a hierarchy. If it's called with the list shown in Figure 2, it disposes of elements in the order 1, 2, and 3.

Using recursion simplifies the list-management algorithms in the TwistDown function library. However, it's not without its pitfalls in the real world. Each time a function such as DisposeTwistDownHandle encounters a subelement list, it calls itself to dispose of that list, and each of these calls uses stack space. While this isn't usually a problem for applications, you should avoid recursive algorithms in device drivers or other nonapplication code segments because they must work within the constraints of some other application's stack.•

CONTROLLING DATA APPEARANCE

Once the data is organized as a hierarchical list, the application could simply display the whole list by just storing handles to the list elements in List Manager cells. However, if your application lets the user control how much of the data is displayed, there must be a way for the user to browse through the data and to specify which elements are visible and which are hidden.

A familiar mechanism for doing this exists in the Finder, where small buttons indicate which cells have subhierarchies and whether the subhierarchy is visible. These triangular buttons have two stable states: ▷ for closed (invisible) hierarchies and ▽ for open (visible) hierarchies. There are also three transient states: an intermediate button, ◢, is displayed briefly when the user clicks a triangular button to change between the open and closed states; and the closed and open buttons are drawn filled when a mouse-down event is located on the button.

To manage these buttons and the display of visible data, each list element needs a few flags and an indentation variable. These are stored in the TwistDownRecord structure.

The indentation variable — indentLevel — specifies the hierarchical depth of an element and is used to display sublists so that the data for an element appears under its parent, but indented to show its place in the hierarchy.

The bits in the flag field are used to record the record's state and to communicate between the application and the List Manager's list definition function (LDEF):

```
/* These are the values that can appear in the flag word. */
#define  kHasTwistDown      0x0001     /* This element has a sublist   */
#define  kShowSublist       0x0002     /* Display the sublist content  */
#define  kOldShowSublist    0x0004     /* Saved kShowSublist state     */
#define  kSelectedElement   0x0008     /* Copy "selected" from list    */
#define  kDrawButtonFilled  0x0010     /* Signal "mouseDown" in button */
#define  kOnlyRedrawButton  0x0020     /* Signal "tracking mouse"      */
#define  kDrawIntermediate  0x0040     /* Draw the animation polygon   */
#define  kEraseButtonArea   0x0080     /* Need complete button redraw  */
```

The flag field is defined as an unsigned short with explicitly defined bits rather than as a bitfield, which would have made the program slightly easier to read. However, the ANSI C standard doesn't specify how the bits in a bitfield are arranged, and different compilers are free to choose their own organization of the bits. This means that if you write parts of your code using several compilers, or distribute modules in object form for others to use, you may cause a debugging nightmare. This is especially true if you use bitfields to construct data records that are sent in network messages between different systems. The flag bits could have been defined as an enum, but this can also cause portability problems. Using explicit bit definitions will also make it easier to convert your code to run on the Power Macintosh.•

The first four flag bits have the following meanings:

- kHasTwistDown is set if the element should have a triangular button when it's drawn. While you might assume that the existence of a non-null subElement pointer would be sufficient, the Finder illustrates a better design: it displays triangular buttons for all folders, even if there are no files in a folder. This immediately tells the user that the line on the display represents a folder, rather than a file.
- kShowSublist is set if the sublist should be displayed. This is normally controlled by the user clicking a triangular button, which changes the kShowSublist state.
- kOldShowSublist is used to save the old kShowSublist setting in case you need to temporarily change the display hierarchy. This lets you undo a display state change or provide a Show All Hierarchies command. It's not used in the sample code.
- kSelectedElement records the selection state of the list cell. It's needed to properly retain the selection status of visible cells.

The other flag bits are needed to handle mouse events. They're set by the mouse-down event handler and the LDEF references them to control its actions:

- kDrawButtonFilled is set when the user presses a button (the cursor is over a triangular button while the user has the mouse buttton held down). It causes the LDEF to fill the triangular button to indicate that the user is pressing it.
- kOnlyRedrawButton is set to constrain the LDEF so that the entire display line doesn't blink when the user presses a button. As the user moves the cursor in and out of the button, the drawing procedure must redraw the button to show whether the user is pressing the button. However, there's no need to redraw the actual data contents. This flag tells the LDEF to redraw only the button.
- kDrawIntermediate is set when the user releases the mouse button within the button area: the button state changes from closed (▷) to open (▿) or vice versa. To indicate this change, the LDEF draws the button in an intermediate state (◢), delays for an instant, and then draws the button in its new, stable state.
- kEraseButtonArea is set to erase the button area before it's drawn. If not set, the button is redrawn in its new form, but not erased; this eliminates unnecessary button flicker.

The TwistDown library uses the following four macros internally to access the flag word:

```
#define SetTDFlag(tdHandle, mask)     ((**tdHandle).flag |= (mask))
#define ClearTDFlag(tdHandle, mask)   ((**tdHandle).flag &= ~(mask))
#define InvertTDFlag(tdHandle, mask)  ((**tdHandle).flag ^= (mask))
#define TestTDFlag(tdHandle, mask)    (((**tdHandle).flag & (mask)) != 0)
```

CREATING THE LIST RECORD

When you first look at the List Manager, it may appear to be the solution to all your display needs. Unfortunately, it has a number of characteristics that restrict its usefulness. It's designed to store limited amounts of data, and performance slows appreciably as you increase the number of cells or the amount of data stored in the cells. Also, if your list cells are not all the same size or your application needs fine control over scrolling, you'll probably find life simpler if you create your own function library. For example, both MacApp and the THINK Class Library offer flexible libraries for displaying and managing structured data. However, the List Manager serves well for straightforward lists of a small number of items — and with the addition of the triangular buttons it becomes a very useful tool.

The TwistDown subroutine library creates a one-column list with a vertical scroll bar. The code has only two unusual features:

- NewTwistDownList stores a small amount of private information in a handle that's stored in the userHandle field of the list record. This includes a pointer to a user-defined drawing function, the display font and font size, and a flag that signals whether clicking on cell data should highlight the cell contents.
- It establishes a private LDEF that manages the visual display. Normally, the LDEF is stored in a resource. In the sample code, however, it's linked into the application and a stub resource is created for the benefit of the List Manager. This stub, a three-instruction procedure that the List Manager calls, jumps to the twist-down LDEF. This is not necessary for this library — it could have been separately compiled — but is useful for debugging and for LDEF procedures that need to access application globals.

When you recompile this program to run on a Power Macintosh as a "native" application (rather than in 680x0-compatibility mode), you'll have to redo this sequence slightly. The time to worry about conversion is now, before your customers are tapping you on the shoulder asking, "Not today? How about next Tuesday?" There's more on converting for Power Macintosh at the end of this article.•

Note that there are two separate drawing procedures: the twist-down LDEF manages the buttons and drawing for simple text displays, while the application program can specify its own drawing function to draw more complex data.

THE TWISTDOWNPRIVATERECORD

The twist-down LDEF requires a small amount of global information to properly process the list. This is stored in a handle-based structure defined as shown in Listing 4.

Two Boolean variables in this record haven't been described: canHiliteSelection and isLeftJustify. The canHiliteSelection field controls whether the LDEF highlights selected cells. The isLeftJustify flag is set for left-to-right languages (such as English) and cleared for languages such as Arabic and Hebrew. This flag isn't used in the code shown in this article, but the TwistDown library on the CD shows how an application might handle a right-to-left language.

CREATING TRIANGULAR BUTTON POLYGONS

The triangular buttons are defined as QuickDraw polygons, rather than as bitmaps, with the advantage that the function is independent of the list cell size and script direction. This is useful for localization or for programs used by people who are visually impaired or have diminished motor skills: the program will display larger

Listing 4. TwistDownPrivateRecord

```
struct TwistDownPrivateRecord {
   TwistDownDrawProc    drawProc;     /* User-defined drawing function */
   PolyHandle           openTriangle;           /* The expanded button */
   PolyHandle           closedTriangle;         /* The closed button   */
   PolyHandle           intermediateTriangle;   /* Animation           */
   short                tabIndent;              /* Child indentation   */
   short                fontSize;               /* For TextSize        */
   short                fontNumber;             /* For TextFont        */
   Boolean              canHiliteSelection;     /* Highlight cell OK?  */
   Boolean              isLeftJustify;          /* GetSystJust value   */
   short                triangleWidth;          /* Button width        */
};
typedef struct TwistDownPrivateRecord TwistDownPrivateRecord,
      *TwistDownPrivatePtr, **TwistDownPrivateHandle;
```

buttons if the application or user chooses a large font. It also lets the program draw the closed and intermediate buttons pointing in the proper direction for right-to-left script systems such as Arabic and Hebrew. Figure 3 illustrates an expanded view of the triangular buttons. As an example, the code in Listing 5 shows how you would create the polygons for a left-to-right script. See the sample on the CD for more general code, which accounts for the writing direction of the script.

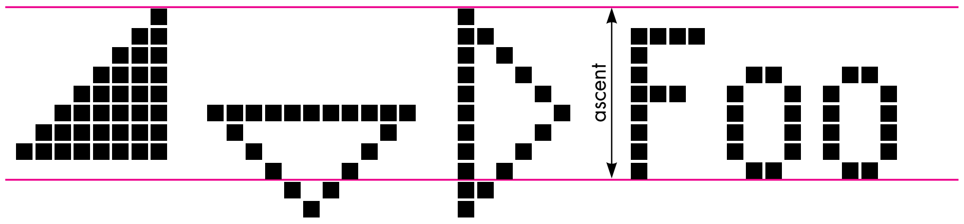


Figure 3. The triangular buttons

PUTTING DATA INTO THE LIST

After creating the List Manager list, the application builds its hierarchical structure (the linked list). List elements are created by the MakeTwistDownElement function. As described earlier in “Creating a List Element,” MakeTwistDownElement obtains the necessary (handle) storage, initializes all flags, and stores the application data in the list element. It also links the new element to the previous (elder sibling) element in the list.

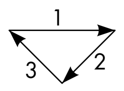
Normally, a twist-down list is built by a recursive function such as the one shown in Listing 6, MyBuildHierarchy. MyBuildHierarchy calls a function named MyGetInfo that stores a small amount of data into a structure called MyInfoRecord. Neither of these is defined here: they’re application specific.

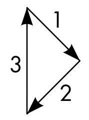
CREATING THE VISIBLE DISPLAY

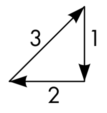
After you’ve built the data hierarchy, the next step is to determine which elements are visible initially and build the visible list. The CreateVisibleList function constructs a new visible display given the head of a hierarchical list and a List Manager handle. It

Listing 5. Creating triangular button polygons

```
GetFontInfo(&info);
buttonSize = info.ascent;            /* The button height           */
buttonSize &= ~1;                    /* Round down to an even number */
halfSize = buttonSize / 2;           /* For 45-degree triangles      */
intermediateSize = (buttonSize * 3) / 4;
(**privateHdl).openTriangle = OpenPoly();
   MoveTo(0, halfSize);
   LineTo(buttonSize, halfSize);
   LineTo(halfSize, buttonSize);
   LineTo(0, halfSize);
ClosePoly();
(**privateHdl).closedTriangle = OpenPoly();
   MoveTo(halfSize, 0);
   LineTo(buttonSize, halfSize);
   LineTo(halfSize, buttonSize);
   LineTo(halfSize, 0);
ClosePoly();
(**privateHdl).intermediateTriangle = OpenPoly();
   MoveTo(intermediateSize, 0);
   LineTo(intermediateSize, intermediateSize);
   LineTo(0, intermediateSize);
   LineTo(intermediateSize, 0);
ClosePoly();
```







Listing 6. Building a twist-down list

```
TwistDownHandle MyBuildHierarchy(ListHandle theList, short indentLevel)
{
   OSErr            status;
   TwistDownHandle  previousElement, thisElement, firstElement;
   MyInfoRecord     myInfoRecord;
   Boolean          isHierarchy;
   EventRecord      currentEvent;

   firstElement = NULL;
   previousElement = NULL;
   /*** Other initialization here                                  */
   do {
      /*** Call EventAvail here to give time to background tasks.    */
      EventAvail(everyEvent, &currentEvent);
      status = MyGetInfo(&myInfoRecord, &isHierarchy);
      if (status == noErr)
         status = MakeTwistDownElement(previousElement, indentLevel,
               sizeof(MyInfoRecord), (Ptr) &myInfoRecord, &thisElement);
      if (status == noErr) {
         /*** Remember the first element in this sibling sequence;    */
         /*** it's needed by our caller.                              */
         if (firstElement == NULL)
            firstElement = thisElement;
```

(continued on next page)

Listing 6. Building a twist-down list *(continued)*

```
         /*** If this data begins a hierarchy, descend by calling this */
         /*** function recursively. Store the first element of the new */
         /*** sublist in the subElement pointer. (flag & kHasTwistDown) */
         /*** will be TRUE even if the child list is empty.           */
         if (isHierarchy) {
            SetTDFlag(thisElement, kHasTwistDown);
            (**thisElement).subElement =
                  MyBuildHierarchy(theList, indentLevel + 1);
         }
         /*** Set sibling linkage for next element.                  */
         previousElement = thisElement;
      }
   } while (status == noErr);
   return (firstElement);
}
```

stores the head in the first cell (cell [0, 0]) and calls BuildVisibleList to store the visible elements in the subsequent cells.

BuildVisibleList is called in two situations: when the application first constructs the list and when the user changes the visual hierarchy by clicking a triangular button. It calls CountVisibleElements to determine the number of cells needed, adjusts the size of the list to the desired number, and calls a recursive function, SetElementsInList, to do the actual storage. SetElementsInList needs what is essentially a global counter to know which cell will receive the current list element.

BuildVisibleList associates list cells with elements in the hierarchical list as follows:

1. It copies the current selection status of each list cell from the List Manager cell to the associated TwistDownHandle element.
2. It counts the number of elements that will be displayed and adds or removes rows from the List Manager list as needed.
3. Finally, it stores references to the visible elements in the list cells, updating the selection status as needed.

The utility functions used in adding and removing elements from the List Manager list aren't shown here but may be examined in the sample library. BuildVisibleList uses several local, recursive functions to process the hierarchical list that all have a similar overall structure. For example, CountVisibleElements (Listing 7) computes the number of list elements that should be displayed.

HANDLING MOUSE EVENTS

Now that we have a visible list, we're ready to let the user manipulate the hierarchy by clicking the triangular buttons. When the user presses the mouse button, the application decides whether the cursor is in one of its windows and whether this window might just happen to have a twist-down list. If so, the application calls DoTwistDownClick with the list handle, a pointer to an event record, and a pointer to a Cell structure that identifies the selected cell on exit. DoTwistDownClick returns one of five action states, as shown in Table 1.

Listing 7. CountVisibleElements

```
short CountVisibleElements(TwistDownHandle twistDownHandle)
{
   short    result;

   result = 0;
   while (twistDownHandle != NULL) {
      ++result;
      if (TestTDFlag(twistDownHandle, kShowSublist))
         result += CountVisibleElements((**twistDownHandle).subElement);
   }
   return (result);
}
```

Table 1
DoTwistDownClick action states

Action State	**Meaning**
kTwistDownNotInList	The mouse-down event was not in the list area. Your application should handle this event.
kTwistDownNoClick	The user pressed the mouse button in a triangular button but released it outside the button. Your application should ignore this click.
kTwistDownButtonClick	The mouse click was in the triangular button. DoTwistDownClick has handled this, but your application may need to do further processing.
kTwistDownClick	The user clicked, once, on list data. Your application may need to do further processing.
kTwistDownDoubleClick	The user double-clicked on list data. Your application may need to do further processing.

The techniques described here for handling mouse events can be used to create lists whose cells contain other kinds of active elements, such as buttons or checkboxes.•

DoTwistDownClick, together with the subroutines it calls, hides a fairly complex process consisting of the following steps; these steps are described further in the following sections and illustrated in the simplified version of DoTwistClick shown in Listing 8.

1. Check that the mouse-down event is in the list area.
2. Check that the user pressed a triangular button.
3. Track the mouse while it's held down.
4. Take appropriate action when the mouse button is released.

Did the user press in the list rectangle?

Get the mouse location in local coordinates and the window rectangle that contains the list and its scroll bar. If the mouse location is not in the list, just return.

Did the user press a triangular button?
The user pressed in the list area; is it in a button? The code sample on the CD has a test for left or right alignment (so that you can use the function with Arabic or Hebrew script systems) but that test is ignored here. The central algorithm determines the rectangle that encloses all the cell buttons. If the cursor is in that area, it then checks whether there is a cell under the cursor and, if so, whether this cell actually displays a button.

Track the mouse while it's in the button area.
If we get past all that, we know that the user pressed a triangular button. The click-handler sets and clears flag bits that the LDEF references when redrawing the list cell. The LDEF starts by drawing the button in its active (filled) state. Note that each call to LDraw redraws the list cell — but, as pointed out earlier, the kOnlyRedrawButton flag prevents the entire display line from blinking.

The sequence beginning with "if (StillDown())" in the code shows how you can track your own visual elements, such as icons, as if they were normal buttons. You can also use this technique to add checkboxes or other button-like objects to list cells.

The user released the mouse button.
When the user releases the mouse button in the triangular button area, the application changes the button state (for example, from ▷ to ▽). This is a two-step process that briefly flashes an intermediate button (◢) to give the user the illusion of change. While your application would certainly work without this subtle touch, it wouldn't look as good. Call the ExpandOrCollapseTwistDownList function after flashing the intermediate button to redraw the button in its new state. Note that kEraseButtonArea is set so that the intermediate button is drawn properly.

Listing 8. DoTwistDownClick

```
TwistDownClickState DoTwistDownClick(ListHandle       theList,
                                     const EventRecord *eventRecordPtr,
                                     Cell              *selectedListCell)
{
   Cell                theCell;      /* Current list cell            */
   Rect                hitRect;      /* The button area in this cell */
   Boolean             inHitRect;    /* Cursor is in the button area */
   Boolean             newInHitRect; /* Cursor moved into the button */
   short               cellHeight;   /* Height of a list cell        */
   short               visibleTop;   /* Top pixel in the list area   */
   TwistDownHandle     twistDownHandle;  /* Current twist-down element*/
   TwistDownPrivateHandle  privateHandle; /* Private data            */
   Point               mousePt;      /* Where the mouse is located   */
   TwistDownClickState result;       /* Function result              */
   long                finalTicks;   /* For the Delay function       */

   /*** 1. Did the user press in the list rectangle?                 */
   mousePt = eventRecordPtr->where;
   GlobalToLocal(&mousePt);          /* Mouse in local coordinates   */
   hitRect = (**theList).rView;      /* Here's the list area         */
   hitRect.right += kScrollBarWidth; /* Include the scroll bar, too  */
```

(continued on next page)

Listing 8. DoTwistDownClick *(continued)*

```
if (PtInRect(mousePt, &hitRect) == FALSE) {
   result = kTwistDownNotInList;
   return (result);
}

/*** 2. Did the user press a triangular button?                    */
privateHdl = (TwistDownPrivateHandle) (**theList).userHandle;
hitRect.right = (**theList).rView.left + (**theList).indent.h
               + (**privateHdl).triangleWidth;
inHitRect = FALSE;
if (PtInRect(mousePt, &hitRect)) {
   /*** The mouse is in the button area; is there a cell?          */
   cellHeight = (**theList).cellSize.v;
   theCell.h = 0;
   theCell.v = ((mousePt.v - (**theList).rView.top) / cellHeight
            + (**theList).visible.top;
   /*** This is a list cell that should have data. Get the twist-   */
   /*** down element handle. If there's no data, or no hierarchy,   */
   /*** the click will be ignored.                                   */
   twistDownHandle = GetTwistDownElementHandle(theList, theCell);
   if ((twistDownHandle != NULL)
    && TestTDFlag(twistDownHandle, kHasTwistDown))
      inHitRect = TRUE;
}
if (inHitRect == FALSE) {
   /*** There's no button here, or the user didn't click it. Just   */
   /*** call the normal list click-handler and return its value.    */
   /*** This is needed to handle scroll bars correctly.             */
   if (LClick(mousePt, eventRecordPtr->modifiers, theList))
      return (kTwistDownDoubleClick);
   else {
      return (kTwistDownClick);
   }
}

/*** 3. Track the mouse while it's in the button area.              */
SetTDFlag(twistDownHandle, kDrawButtonFilled | kOnlyRedrawButton);
LDraw(theCell, theList);
/*** Set hitRect to the triangular button dimensions.               */
hitRect.top = (theCell.v - (**theList).visible.top) * cellHeight
            + (**theList).rView.top;
hitRect.bottom = hitRect.top + cellHeight;
/*** Track the mouse while it's still down: if it moves into the    */
/*** rectangle, redraw it filled; if it moves out, redraw it        */
/*** unfilled.                                                       */
if (StillDown()) {
   while (WaitMouseUp()) {
      GetMouse(&mousePt);
      newInHitRect = PtInRect(mousePt, &hitRect);
```

(continued on next page)

Listing 8. DoTwistDownClick *(continued)*

```
         if (newInHitRect != inHitRect) {
            /*** The cursor moved into or out of the triangle.        */
            InvertTDFlag(twistDownHandle, kDrawButtonFilled);
            LDraw(theCell, theList);
            inHitRect = newInHitRect;
         }
      }
   }

   /*** 4. The user released the mouse button.                      */
   if (inHitRect == FALSE) {
      /*** The user canceled the operation by releasing the mouse     */
      /*** outside the triangular button area. drawButtonFilled will  */
      /*** normally be clear. It can be set, however, if the user     */
      /*** clicks so briefly that the StillDown() test above is       */
      /*** FALSE.                                                      */
      if (TestTDFlag(twistDownHandle, kDrawButtonFilled)) {
         ClearTDFlag(twistDownHandle), kDrawButtonFilled);
         LDraw(theCell, theList);
      }
      ClearTDFlag(twistDownHandle), kOnlyRedrawButton);
      return (kTwistDownNoClick);
   }
   SetTDFlag(twistDownHandle, (kDrawIntermediate | kEraseButtonArea));
   LDraw(theCell, theList);
   Delay(kAnimationDelay, &finalTicks);
   ClearTDFlag(twistDownHandle,
      (kDrawIntermediate | kDrawButtonFilled | kEraseButtonArea));
   ExpandOrCollapseTwistDownList(theList, theCell);
   *selectedListCell = theCell;
   ClearTDFlag(twistDownHandle, kOnlyRedrawButton);
   return (kTwistDownButtonClick);
}
```

EXPAND OR COLLAPSE THE HIERARCHY

ExpandOrCollapseTwistDownList (Listing 9) is normally called directly by DoTwistDownClick, as shown in the preceding section. It can also be called directly by the application. When called, it inverts the "expansion" state of the designated list cell, redraws the triangular button, and calls BuildVisibleList (described earlier in "Creating the Visible Display") to revise the visible hierarchy. Note that BuildVisibleList will modify the display starting with the current cell: the cells above will not change and thus need not be modified or redrawn.

DRAWING THE LIST CELL

When the contents of a list cell change or the display requires updating, the List Manager calls the TwistDownLDEF function. This function draws the button in its current state and either draws the list cell (for simple text cells) or calls a user-defined drawing function to draw more complex cells. The code is generally straightforward (again, ignoring right or left considerations). Basically, it examines the state of the kOnlyRedrawButton flag and proceeds as follows:

- If the flag is set, “shrink” the display rectangle so that only the button is redrawn. Choose the correct triangular polygon and draw it in its proper state.
- If the flag is clear, draw the cell data and the triangular polygon.

Let’s look more closely at the TwistDownLDEF drawing code (Listing 10):

1. First we determine what to draw. To begin drawing the list, we first need the cell content. This is the handle that contains the list element. We also check that userHandle has been set up correctly. Note that we don’t use the List Manager’s LFind function because the data might not be aligned in the list cell storage. (Actually, the data is aligned, because only handles are stored in the cells, but it doesn’t hurt to be suspicious.) If the handle contains a list element, the values of the flags determine what to draw.
2. Next, we call DrawTriangle to draw the triangular button. The value of theFlag determines the button state and its location.
3. Finally, after checking to be sure kOnlyRedrawButton is not set and that we have the data, TwistDownLDEF redraws the cell data with the proper indentations. Here’s where the code allows you to specify a user-defined drawing function.

Listing 9. ExpandOrCollapseTwistDownList

```
twistDownHandle = GetTwistDownElementHandle(theList, theCell);
if ((twistDownHandle != NULL)
 && TestTDFlag(twistDownHandle, kHasTwistDown)) {
   InvertTDFlag(twistDownHandle, kShowSublist);
   /*** Redraw the triangular button in its new state.              */
   ClearTDFlag(twistDownHandle, kDrawButtonFilled);
   SetTDFlag(twistDownHandle, (kOnlyRedrawButton | kEraseButtonArea));
   LDraw(theCell, theList);
   ClearTDFlag(twistDownHandle, (kOnlyRedrawButton | kEraseButtonArea));
   /*** If some other part of the list will change, rebuild the List  */
   /*** Manager cells and redraw the list.                            */
   if ((**twistDownHandle).subElement != NULL)
      BuildVisibleList(theList, theCell.v);
}
```

Listing 10. TwistDownLDEF drawing code

```
pascal void TwistDownLDEF(
      short            listMessage,
      Boolean          listSelect,
      Rect             *listRect,
      Cell             listCell,        /* Unused                    */
      short            listDataOffset,  /* Unused                    */
      short            listDataLen,
      ListHandle       theList
   )
```

(continued on next page)

Listing 10. TwistDownLDEF drawing code *(continued)*

```
{
   short            indent;            /* Cell data indentation     */
   TwistDownHandle  twistDownHandle;   /* The cell data             */
   TwistDownPtr     twistDownPtr;      /* Cell data (locked handle) */
   short            cellSize;          /* sizeof(TwistDownHandle)   */
   PolyHandle       polyHandle;        /* Button polygon            */
   Point            polyPoint;         /* Where to draw the button  */
   Rect             viewRect;          /* Actual cell drawing area  */
   signed char      elementLockState;  /* twistDownHandle lock state*/

   #define TestFlag(flagBit) ((theFlag & (flagBit)) != 0)

   . . . /*** Other LDEF processing isn't shown.                    */

   /*** 1. Determine what to draw.                                  */
   cellSize = sizeof twistDownHandle;
   LGetCell(&twistDownHandle, &cellSize, listCell, theList);
   if ((cellSize == sizeof twistDownHandle) && twistDownHandle != NULL) {
      /*** There is a list element. (This if statement extends all    */
      /*** the way to the end of the sequence.) Lock the element in    */
      /*** memory and look at the flag values. Set viewRect to the     */
      /*** part of the List Manager cell that will be drawn.           */
      elementLockState = HGetState((Handle) twistDownHandle);
      HLock((Handle) twistDownHandle);
      twistDownPtr = (*twistDownHandle);
      privateHdl = (TwistDownPrivateHandle) (**theList).userHandle;
      viewRect = *listRect;
      theFlag = (*twistDownPtr).flag;
      if (TestFlag(kOnlyRedrawButton)) {
         /*** Shrink the display area when only the button is redrawn.*/
         viewRect.right = viewRect.left + (**theList).indent.h
                        + (**privateHdl).triangleWidth;
      }
      if (TestFlag(kOnlyRedrawButton) == FALSE
       || TestFlag(kEraseButtonArea))
         EraseRect(&viewRect);

      /*** 2. Draw the triangular button.                              */
      if (TestFlag(kHasTwistDown)) {
         polyPoint.v = listRect->top + 1;
         polyPoint.h = listRect->left + (**theList).indent.h
                     + kTriangleOutsideGap;
         if (TestFlag(kDrawIntermediate))
            polyHandle = (**privateHdl).intermediateTriangle;
         else if (TestFlag(kShowSublist))
            polyHandle = (**privateHdl).openTriangle;
         else
            polyHandle = (**privateHdl).closedTriangle;
         DrawTriangle(polyHandle, polyPoint, theFlag & kDrawButtonFilled);
      }
```

(continued on next page)

Listing 10. TwistDownLDEF drawing code *(continued)*

```
      /*** 3. Draw the cell data.                                 */
      if (TestFlag(kOnlyRedrawButton) == FALSE
       && (*twistDownPtr).dataLength > 0) {
         /*** Indent the text to show the depth of the hierarchy. Then */
         /*** build a display rectangle for the cell text and set the  */
         /*** pen to the leftmost position of the text.                */
         indent = (**theList).indent.h + (**privateHdl).triangleWidth
               + ((**privateHdl).tabIndent * (*twistDownPtr).indentLevel);
         viewRect = *listRect;
         viewRect.left += indent;
         TextFont((**privateHdl).fontNumber);
         TextSize((**privateHdl).fontSize);
         /*** If the user didn't provide a drawing procedure, draw a   */
         /*** text string. Otherwise, call the user's procedure.       */
         if ((**privateHdl).drawProc == NULL) {
            MoveTo(viewRect.left, viewRect.top + (**theList).indent.v);
            DrawText((*twistDownPtr).data, 0, (*twistDownPtr).dataLength);
         }
         else {
            (*(**privateHdl).drawProc)( /* Call user's drawing function */
               theList,                     /* The list handle   */
               (const Ptr) (*twistDownPtr).data,  /* Data to draw     */
               (*twistDownPtr).dataLength,  /* Size of the data */
               &viewRect);                  /* Where to draw it  */
         }
      }                          /* If we're drawing cell data      */
      HSetState((Handle) twistDownHandle, elementLockState);
   }                             /* If we have cell data            */
}
```

THE DRAWTRIANGLE FUNCTION

If you look closely at the triangular buttons on a color or grayscale display, you'll notice that the button is filled with a grayish background color. (The Finder uses the color the user assigned to the file, while we use a light gray color.) The DrawTriangle function called by TwistDownLDEF takes three parameters: the polygon, where it's to be drawn, and a Boolean that specifies whether the user is currently pressing the triangular button. DrawTriangle uses the DeviceLoop procedure, DrawThisTriangle, which calls the actual drawing function for each type of device so that drawing can be optimized for different screen depths. (See Listing 11.)

The DeviceLoop procedure is described in "DeviceLoop Meets the Interface Designer," *develop* Issue 13, and in *Inside Macintosh* Volume VI.•

THE SAMPLE PROGRAM

The sample program illustrates how you can use twist-down lists to display a directory of all files on a volume. It's a very simple program and you would be well advised not to use it on a huge disk with many folders and files, because there's no protection against storage overflow.

The sample program compiles and runs in five environments: THINK C 6.0, Metrowerks DR1, and MPW 3.2 for the 680x0-based Macintosh; and, for the Power

Listing 11. DrawTriangle and DrawThisTriangle

```
typedef struct TriangleInfo {      /* Passed to DrawThisTriangle      */
   PolyHandle       polyHandle;    /* The polygon to draw             */
   Point            polyPoint;     /* Where to draw it                */
} TriangleInfo, *TriangleInfoPtr;

static void DrawTriangle(PolyHandle   polyHandle,
                        Point        polyPoint,
                        Boolean      isSelected
   )
{
   TriangleInfo     triangleInfo;
   RgnHandle        drawingRgn;
   long             savedA5;

   /*** Refresh A5 so that we can use the current QuickDraw globals.  */
   savedA5 = SetCurrentA5();
   triangleInfo.polyHandle = polyHandle;    /* Save our drawing       */
   triangleInfo.polyPoint = polyPoint;      /*  parameters.           */
   /*** Position the polygon properly on the display.                 */
   OffsetPoly(polyHandle, polyPoint.h, polyPoint.v);
   if (isSelected)
      FillPoly(polyHandle, &qd.black);
   else {
      /*** Get drawing region and call DeviceLoop to do the work.     */
      drawingRgn = NewRgn();
      OpenRgn();
      FramePoly(polyHandle);
      CloseRgn(drawingRgn);
      DeviceLoop(
         drawingRgn,                     /* Region to draw into       */
         (DeviceLoopDrawingProcPtr) DrawThisTriangle,
         (long) &triangleInfo,           /* Drawing parameters        */
         0                               /* DeviceLoop flags (ignored)*/
      );
      DisposeRgn(drawingRgn);
   }
   /*** Frame the button in black and move the polygon back to its    */
   /*** default [0,0] position.                                       */
   FramePoly(polyHandle);
   OffsetPoly(polyHandle, -polyPoint.h, -polyPoint.v);
   SetA5(savedA5);
}

static pascal void DrawThisTriangle(       /* Called by DeviceLoop    */
      short              depth,            /* Screen pixel depth      */
      short              deviceFlags,      /* Device info (ignored)   */
      GDHandle           targetDevice,     /* The display (ignored)   */
      TriangleInfoPtr    triangleInfoPtr   /* The data to be drawn    */
   )
```

(continued on next page)

Listing 11. DrawTriangle and DrawThisTriangle *(continued)*

```
{
   RGBColor            foreColor;
   RGBColor            saveForeColor;
   RGBColor            backColor;
   short               i;
   Rect                polyRect;

   polyRect = (**(*triangleInfoPtr).polyHandle).polyBBox;
   LocalToGlobal(& ((Point *) &polyRect)[0]);
   LocalToGlobal(& ((Point *) &polyRect)[1]);
   if (depth > 1) {
      /*** Drawing in color or grays: fill the triangle with a very    */
      /*** light gray.                                                 */
      GetForeColor(&foreColor);
      saveForeColor = foreColor;
      GetBackColor(&backColor);
       /*** This loop sets foreColor to a very light gray.             */
      for (i = 0; i < 8; i++) {
         if (GetGray(GetGDevice(), &backColor, &foreColor) == FALSE)
            break;
      }
      RGBForeColor(&foreColor);
      FillPoly((*triangleInfoPtr).polyHandle, &qd.black);
      RGBForeColor(&saveForeColor);
   }
   else {
      /*** Monochrome: erase the interior of the polygon.              */
      ErasePoly((*triangleInfoPtr).polyHandle);
   }
}
```

Macintosh, Metrowerks DR1 and the MPW provided in the Macintosh on RISC Software Developer's Kit. Converting the code for Power Macintosh took about one day (it was my lab exercise when I took the Apple Developer University "PowerPC BootCamp" course). To learn more about what I did to accomplish this conversion, see "Converting for Power Macintosh."

When you start up the sample program, it begins enumerating the disk; you can click to stop it at any time. The hierarchical list is built using the algorithm illustrated by the MyBuildHierarchy function, described in the section "Putting Data Into the List."

TWISTED LISTERS

So, what's this method of displaying data good for? If you have data that's hierarchical, coherent, line-oriented, and not too large, you'll find that the twist-down list functions are both useful and easy to incorporate into your applications.

- Hierarchical. If the data doesn't separate into a strict hierarchy, the presence of triangular buttons will only confuse your users: they're expecting your application to operate like the Finder. Also, if the hierarchy is very limited (a single topic with a block of text), you'll probably find some other solution easier to use.

CONVERTING FOR POWER MACINTOSH

Here's a simplified checklist of the kinds of things you'll need to do to convert code for the Power Macintosh, based on what I did to convert my program (see the sample code on the CD, especially TwistDownList.c). I borrowed heavily from the document "Moving Your Source to PowerPC" on the Macintosh on RISC Software Developer's Kit CD.

- Convert to standard C using the most restrictive environment so that the application runs correctly in THINK C and MPW with no compiler or linker errors or warnings. In THINK C, for example, you would enable the Check Pointer Types and Require Prototypes options and remove the MacHeaders option. In all cases, you should add prototypes and explicit function return types and fix potential trigraph problems.
- Replace all instances of int and unsigned by explicit short and long declarations. Be very careful about structure definitions that are shared among code modules (or written to files or resources), as the Power Macintosh aligns structures differently from the 680x0-based Macintosh: you may need to add #pragma statements to override the compiler.
- Create a makefile for the MPW development system. Try to build "fat binaries" that will run native on both the 680x0-based Macintosh and Power Macintosh.
- If at all possible, convert to the universal interfaces provided in the Software Developer's Kit (and on this issue's CD). In particular, ProcPtr references must be converted to UniversalProcPtr. Also, all low-memory references must be replaced by the access functions provided as part of the universal interfaces.
- Isolate system and compiler dependencies by using #ifdef statements. For MPW-based compilers, add **-d MPW=1** to your makefiles. This lets you add compiler- and system-dependent #pragmas and code sequences without encountering compiler warnings.

```
#ifdef THINK_C    /* THINK C         */
#ifdef __powerc   /* Power Macintosh */
#ifdef MPW        /* MPW: see above  */
#ifdef applec     /* Apple compilers */
```

- Remove or isolate all assembly language and inline statements. You can probably eliminate all assembly language from your Power Macintosh applications.
- Power Macintosh will not support a number of obsolete system traps; when you convert your program, you may need to rewrite small sections of your code. Of course, you should check new code on both the 680x0-based Macintosh and Power Macintosh.
- Applications must explicitly allocate space for the QuickDraw globals. Add the following to your application's main program file:

```
#ifdef __powerc
QDGlobalsqd;
#endif
```

- Avoid storing application-specific data in your application's data fork: the Power Macintosh stores its code there. (You can reserve a fixed amount of space at the beginning of the data fork, if necessary.)
- Add a 'cfrg' (code fragment) resource to your application's resource fork. This tells the Process Manager that you've built a Power Macintosh native application.

```
#ifdef __powerc
#include "CodeFragmentTypes.r"
resource 'cfrg' (0) {
   {
      kPowerPC,
      kFullLib,
      kNoVersionNum, kNoVersionNum,
      0, 0,
      kIsApp, kOnDiskFlat,
      kZeroOffset, kWholeFork,
      "MyFirstPowerPCApp"
   }
};
#endif
```

- Make good use of the compatibility built into the Power Macintosh: your application should run native on Power Macintosh and still run correctly on a 680x0-based machine. The user shouldn't notice any difference.
- In most cases, native Power Macintosh applications will be about the same size as their 680x0 counterparts. Of course, if you use the compatible "fat binary" capability, the application file size will increase.

The above list isn't complete by any means, but, together with the sample code, it should get you started. Also, there are several Developer University courses available to help bring you up to speed quickly.

- Coherent. Your users expect consistency between the parent "title" and child "content" data. Try another technique if clicking a triangular button does something other than reveal a lower-level hierarchy. For example, the JMP statistical package uses standard Macintosh buttons to expand a hierarchy. Clicking a button may reveal a table of data or a graphical element. (JMP hierarchies are also quite shallow.)
- Line-oriented. Again, the user is expecting Finder-like behavior. If the data is not line-oriented, you'll discover that coaxing the List Manager to deal with your data isn't worth the considerable effort it takes. In particular, avoid varying the height of each line as this makes the triangular buttons look weird (some small, some large) unless you normalize their size.
- Not too large. This is a restriction of the List Manager. Because of the way it stores data, there's an absolute limit of 32,767 cells in a list, but it becomes very slow and clumsy with more than a few hundred cells.

I wrote the TwistDown library because I wanted to display an AOCE catalog specification that can contain many internal components of varying size and complexity. It offered a friendly interface into a structure that is convoluted, warped, and — indeed — twisted.

REFERENCES

- "Standalone Code on PowerPC" by Tim Nichols, *develop* Issue 17.
- "Making the Leap to PowerPC" by Dave Radcliffe, *develop* Issue 16.
- *Inside Macintosh: More Macintosh Toolbox* (Addison-Wesley, 1993), Chapter 4, "List Manager," or *Inside Macintosh* Volume IV (Addison-Wesley, 1986), Chapter 30, "The List Manager Package."
- "DeviceLoop Meets the Interface Designer" by John Powers, *develop* Issue 13. DeviceLoop is also described in *Inside Macintosh* Volume VI (Addison-Wesley, 1991) on page 21-23.

ACKNOWLEDGMENTS
The TwistDown library is generally based on code from the NewsWatcher application by Steve Falkenburg. Steve's code is based on code written by John Norstad, author of Disinfectant.•

Thanks to our technical reviewers Jon Callas, Godfrey DiGiorgi, Steve Falkenburg, Dave Radcliffe, and Dean Yu. And thanks to Richard Clark for the PowerPC BootCamp course.•

View From the Ledge

TAO JONES

Dear Tao,

Someone at work is stealing all my pens. I know it doesn't sound like a big deal, but it's getting to the point where I'm going through dozens a week. Even more frustrating is that I'm certain it's happening during normal business hours, not at night.

I just know the Internal Revenue Service isn't going to believe a huge deduction for office supplies at the end of the year. What can I do?

Pound Wise But Pen Foolish

Dear Pound,

You are in one of those unique positions where you can not only solve the mystery of the missing pens, but also spruce your office up a bit.

The first thing you need to do is cover your walls with blacklight posters. As a minimum you should get a tiger, a Jimi Hendrix, and a flaming dirigible. What's really great is that although interior design has taken huge leaps forward since the 60s, blacklight art has remained remarkably the same. Any investment you make now is sure to be preserved for years.

Once you've got the posters, you naturally must have a blacklight to properly show them off. I would recommend the most powerful one you can get, but you should stay away from the strobes — although rare, there are some people who experience seizures from strobe light.

At this point the trap is set. Now all you need to do is dust your pens with ultraviolet powder (available from any burglar alarm shop). Make sure that you leave the light off for the first day or two. Then turn it back on and watch the people who come back and forth from your office. The one with the glowing yellow hands is your best suspect.

Note that if you're going to spend much time working under these lamps, you should get yourself a pair of UV goggles. It's not a bad idea to have a pair anyway; they make a great fashion accessory to spice up any wardrobe.

If things ever get a bit dull around the office, you can always set up your own security desk. Imagine the thrill you'll get saying things like "Excuse me, miss, you'll need to be stamped in order to reenter the building." Putting up a sign that reads "No bottles, cans, knives, or tape recorders" will just add to the ambiance.

Dear Tao,

Lately I've been pondering a real big question that I'm not making any headway with: what, exactly, is it that people are trying to accomplish? Sure, all these software companies are trying to change the world, make profits, and all of that, but why?

Puzzled

Dear Puzzled,

You're asking the question that has plagued people from time immemorial. It's been phrased lots of different ways, usually by big people thinking big thoughts and wearing strange clothes, but the crux is always the same: just what is the deal?

Different religions and philosophies will give you different answers. Buddhists will tell you about enlightenment, Christians will expound on heaven, existentialists will ask "Why do you even care?" and a ten-year-old kid will point to a candy store. Unfortunately, all of these solutions look too far forward into the future, are too imbued with concepts of the human spirit, and still are not answering the basic question as it concerns computing.

TAO JONES paid his way through college by volunteering as a subject for psychology experiments. He became obsessed with trying to figure out what the experiments he was participating in were trying to determine and then defeating them. One day he was told to go to the testing room down the hall and on the right. He went down the hall and entered a room that was completely dark. Figuring it was an experiment in sensory deprivation, he went in and sat down. Two days later, he emerged, nearly dead from dehydration. It was then that he discovered he'd gone down the hall and turned *left* only to end up in the janitor's closet.•

I know the answer, but as part of the fraternity of philosophers, advice columnists, and magicians, I'm not supposed to release our secrets. However, I've never been comfortable with being a part of the "in" crowd, so I'll tell you the answer: it's Pac-Man.

From Day 1 people have wanted to be entertained, but for millennia this need was never truly fulfilled. Then in the 1980s Pac-Man came along and there was a brief period of bliss. Money could actually buy happiness — assuming you had at least a quarter and a Pac-Man machine nearby.

Of course, a person can take only so much of any given kind of happiness, especially one that goes "wokka, wokka, wokka, GOINK!" People became burned out, and the search has been on ever since. That's right: this $200 billion-a-year industry, and all those government think tanks, are actually doing nothing more than searching for the next Pac-Man. Most experts agree that the next big breakthrough will be in a driving game of some type, which explains why you've been hearing so much about the digital highway lately.

Dear Tao,

Believe it or not, I actually like wearing a suit to the office. I've tried the standard jeans and T-shirt outfit, but I just don't feel comfortable in them. My problem is that when I do dress up, my colleagues continually criticize me for it. What can I do?

Pinstriped in Pennsylvania

Dear Pinstriped,

Unfortunately, you're in a very difficult situation that probably isn't "curable." The ailment was discovered in the 1950s and is most commonly referred to today as the "Liberace Syndrome." The studies of the human genome seem to indicate that there's some sort of defect in the appearance gene that will make affected individuals want to start dressing flashier and flashier. It's not clear what causes it, although chronic exposure to jewelry, candelabra, or Las Vegas clearly will make the condition worse. You'll also find that your condition will become more severe with age.

The disease starts very mild. At first you'll shun sneakers. Then you'll start thinking that cotton has too rough of a feel. As things progress to the final and most outrageous stages, you'll find yourself wanting to wear sequined capes and feather boas. Thousands of people have been afflicted by the Liberace Syndrome: Little Richard, Elvis Presley, Elton John, Madonna, James Brown, and Rip Taylor, to name a few.

So it's bad news and good news about your affliction. The good news is that it's possible to live a full, relatively happy life. The bad news is that you'll never be able to do so in the computer industry. My recommendation is to start singing every morning in the shower, find an agent, and figure out which colors best match your hair and complexion.

RECOMMENDED READING AND LISTENING

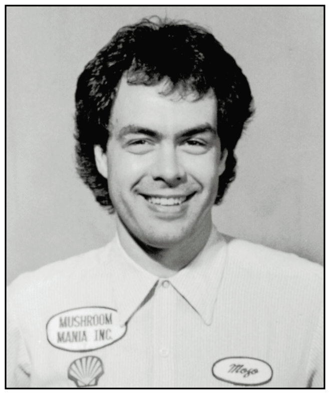
- *The Official Scrabble Players Dictionary* by Selchow & Righter. Great words from *aa* to *zyzzyva*.
- *Pop Art Book of 30 Postcards* by Magna Books. High time to send your friend a Lichtenstein.
- *School's Out* by Alice Cooper (Warner Bros. Records). Try to find a used copy of the LP, which includes paper panties that were banned as a "fire hazard."

Tao Index: A person's belief in the truth of a particular argument is inversely proportional to their emotional fury in delivering it.•

You can determine the future of Tao Jones. Simply put, this may be Tao's last column. If he receives no more questions, we will put him in a job better suited to his skills: repairing Lisas and Apple IIIs. Will Tao be saved? Only if enough of you AppleLink DEVELOP with a question on office survival.•

The Right Way to Implement Preferences Files

Many Macintosh applications use preferences files to keep track of user preferences. This article describes the characteristics of a well implemented preferences file and introduces a library that manages this work for you.



GARY WOODCOCK

Preferences files have become a standard feature of most Macintosh applications. With these files being so commonplace, it's timely to investigate how to implement them properly. We'll first take a look at what constitutes a well implemented preferences file; then we'll inspect a library API that provides a simple means of creating and interacting with preferences files. On this issue's CD you'll find source code for a standard preferences library and a test application that illustrates how to use the library. This library is written with version 6.0.1 of Symantec's THINK C for Macintosh using the universal interface files (which can also be found on the CD).

Before we begin, let's take a look at when and when not to create a preferences file. A preferences file needs to be created only when the user has altered the default configuration of the application. Upon being launched for the first time, many applications automatically search for a preferences file and, if it isn't found, immediately create one. But why? Has the user changed anything? Nope. The user probably hasn't even had an opportunity to pull down an application menu at this point. So why do we need to create a preferences file? We don't, and we shouldn't.

But given those situations in which your application *should* create a preferences file, read on to learn how to do it the right way.

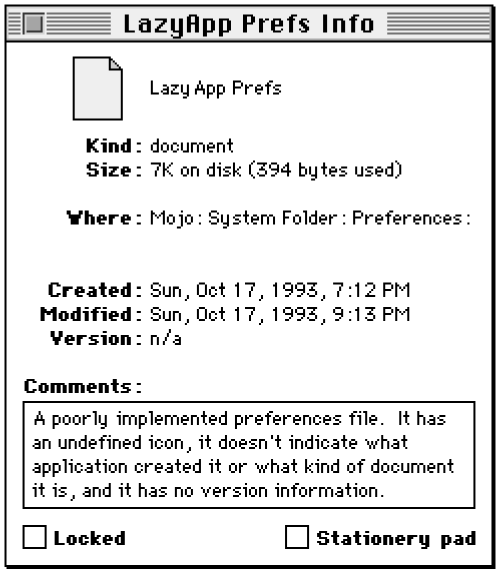
WHAT MAKES A WELL IMPLEMENTED PREFERENCES FILE

The two Finder info windows shown in Figure 1 illustrate a poorly implemented and a well implemented preferences file. We can categorize the visible differences in Figure 1 by file type, document kind, Finder icon, and version information, as explained in the following sections. Note that the concepts presented here apply to implementing *any* kind of document file, not just preferences files.

FILE TYPE

Like any other file created by your application, your preferences file should have a unique file type, which is specified in a file reference ('FREF') resource in the

GARY WOODCOCK, formerly of Apple but now at 3DO, is currently assimilating experiences accrued during his annual pilgrimage to the South by Southwest Music Conference in Austin, Texas. For those readers who find stimulation in spending four days and nights attending live performances by a diverse assortment of interesting, boisterous, and relatively unknown bands playing in a variety of beer-drenched pubs, in the company of equally interesting, boisterous, and beer-drenched people, a better venue Gary cannot recommend.•


LazyApp Prefs Info

Lazy App Prefs

Kind: document
Size: 7K on disk (394 bytes used)

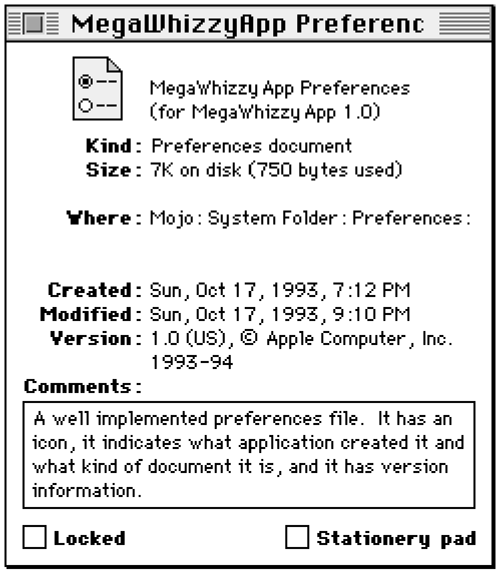
Where: Mojo : System Folder : Preferences :

Created: Sun, Oct 17, 1993, 7:12 PM
Modified: Sun, Oct 17, 1993, 9:13 PM
Version: n/a

Comments:
A poorly implemented preferences file. It has an undefined icon, it doesn't indicate what application created it or what kind of document it is, and it has no version information.

☐ Locked ☐ Stationery pad

Poorly implemented preferences file


MegaWhizzyApp Preferenc

MegaWhizzy App Preferences
(for MegaWhizzy App 1.0)

Kind: Preferences document
Size: 7K on disk (750 bytes used)

Where: Mojo : System Folder : Preferences :

Created: Sun, Oct 17, 1993, 7:12 PM
Modified: Sun, Oct 17, 1993, 9:10 PM
Version: 1.0 (US), © Apple Computer, Inc. 1993-94

Comments:
A well implemented preferences file. It has an icon, it indicates what application created it and what kind of document it is, and it has version information.

☐ Locked ☐ Stationery pad

Well implemented preferences file

Figure 1. Info windows for example preferences files

application. The file type of the preferences file should be associated with the application signature so that the Finder can display which application created the file when its info window is shown. This association is made in your application's bundle ('BNDL') resource, which includes the application signature, the file reference resources, and the application and document file icons. The info window for the poorly implemented preferences file shown in Figure 1 doesn't display any information about which application created the file because the file type isn't associated with an application signature.

Incidentally, you should *not* use 'pref' as the file type for your preferences file; this file type is reserved for the Finder Preferences file (as you'd find out if you tried to register this file type with Apple's Developer Support Center — you *were* going to do that for the file type of your preferences file, right?). If you were to give your preferences file this file type, you'd notice that when the user turns on Balloon Help, the Balloon Help for your preferences file is actually the Balloon Help for the Finder Preferences file. I'll go out on a limb here and assume that we can all agree this is confusing for users, and should therefore be avoided.

DOCUMENT KIND

A file's Finder info window also indicates the kind of document the file is. In many cases, this field reads either "document" or something like "MegaWhizzyApp document." But with the introduction of the Translation Manager (first made available with Macintosh Easy Open), a new resource type has been defined that your application can use to more accurately describe its files, including its preferences file. This resource is called the 'kind' resource, and its Rez definition is as follows:

```
type 'kind' {
      literal longint;     /* App signature */
      integer;             /* Region code of kind string localizations */
      integer = 0;
      integer = $$CountOf(kindArray); /* Array size */
      wide array kindArray {
            literal longint;           /* File type */
            pstring;                   /* Custom kind strings */
            align word;
      };
};
```

An example 'kind' resource might look like this:

```
resource 'kind' (128) {
   'TSTR',
   0,
   {
      'TEXT', "MegaWhizzyApp text document"
   }
};
```

In this example, assuming the Translation Manager is present, a file with creator 'TSTR' and file type 'TEXT' will display the string “MegaWhizzyApp text document” in the document kind field of its info window.

FINDER ICON

As suggested in *Macintosh Human Interface Guidelines*, the best icon for preferences files is either the standard preferences file icon or an icon that incorporates some elements of the standard preferences file icon (see Figure 2). It’s not taboo to use a unique preferences file icon, but this may make it difficult for users to recognize the file as a preferences file.

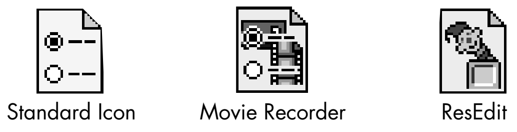


Figure 2. Example preferences file icons

As noted above, you provide a Finder icon for your preferences file the same way that you provide Finder icons for your other document files — through a bundle resource in your application.

VERSION INFORMATION

A file’s 'vers' (version) resources determine what version information is displayed in the Finder info window. There are normally two 'vers' resources for a file — 'vers' IDs 1 and 2 — and each contains the following:

- A numeric code representing the version. This code consists of a major revision number, a minor revision number, an optional bug fix revision number, a stage code (development, alpha, beta, or final), and a revision level (for non-final stages).
- A region code, which indicates the localized version of system software appropriate for use with the file.
- A short version string identifying the version number.
- A long version string consisting of the file version number and company copyright in 'vers' ID 1, and the product version number and name in 'vers' ID 2.

For more information about these resources and how they affect what's displayed in the Finder info window, see pages 7-31 to 7-32 of *Inside Macintosh: Macintosh Toolbox Essentials.* For the structs corresponding to the 'vers' resource, check out the Files.h header file.

Providing version information in your preferences files does more than just make the Finder info window look pretty. You can use this information to identify the format of the data contained in a preferences file. This is useful in determining how to translate preferences data created by an older version of an application into the preferences format for the current version of the application.

CAN USERS OPEN PREFERENCES FILES?

Users have a penchant for running rampant through their hard disks, looking for interesting files to open, particularly if the files weren't explicitly created by them. Files kept in the System Folder are no exception to this experimentation, so it should come as no surprise to you that your application should be prepared to correctly handle the case where its preferences file has been double-clicked.

There are several interesting possible behaviors that a preferences file might exhibit under these circumstances. One alternative is that the application the preferences file belongs to could launch, configuring itself with the data contained in the file. A variation of this behavior would be to display the application's preferences dialog after launching, if such a dialog were supported. A third behavior is that when a user double-clicks a preferences file, its application isn't launched, but instead a dialog is displayed describing what the file is, where it belongs, and why it can't be opened.

Today, the only preferences file behavior approved and documented by Apple is the last one. The Human Interface Design Center of Apple's system software group has this to say about preferences files:

- Preferences files should not be treated as if they were documents created by the user. Launching an application and optionally opening a preferences dialog support the misconception that they *are* like documents.
- Not all applications have preferences dialogs. Launching an application that doesn't have a preferences dialog when its preferences file is double-clicked is confusing, because there's nothing specific to show the user that represents the information stored in the file.
- The current thinking is that a user doesn't open the System Folder, see a preferences file for MacWrite (for example), and wonder what the preferences for MacWrite are set to; he's more likely to see the file and wonder what it's used for. In this case, an informative dialog that's displayed when the user attempts to open the preferences file better satisfies the user's intent.
- Ideally, users shouldn't really even have to know about preferences files (and therefore shouldn't be able to double-click them), but they're an unavoidable artifact of the current system.

It should be pointed out that launching an application by double-clicking a file containing configuration data is not necessarily a bad thing — it just shouldn't be done with a preferences file. Your application could use an application-specific configuration file to allow users to store and set up custom configurations. By creating a specific file type for this information, you make it explicit to users that the behavior of this file type is different from that of preferences files, and that it is unique to your application.

Now let's take a closer look at how to implement the recommended preferences file behavior.

SUPPRESSING LAUNCH UPON DOUBLE-CLICK

As we learned in the previous section, a preferences file isn't intended to be a document that users can open and directly interact with in the application that created it. So how can you keep users from launching your application when they double-click your preferences file?

Let's take a brief look at how the System 7 Finder handles opening document files. Normally, when a user tries to open a document file, the Finder searches for an application that has a signature matching the file creator. If the search is successful, the Finder launches the application and (if the application supports the required Apple event suite) sends an Open Documents Apple event to the application, with the document file as a parameter. If the search for the application that created the file isn't successful, an alert similar to the one shown in Figure 3 is displayed.

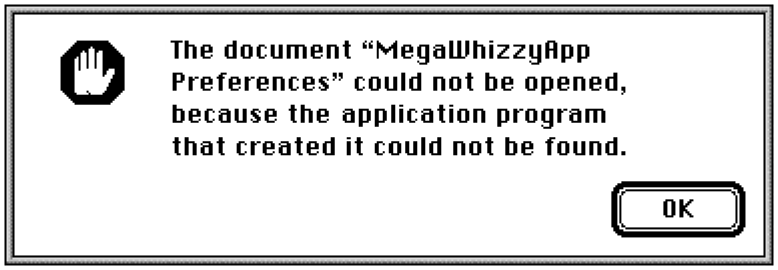

The document "MegaWhizzyApp Preferences" could not be opened, because the application program that created it could not be found.

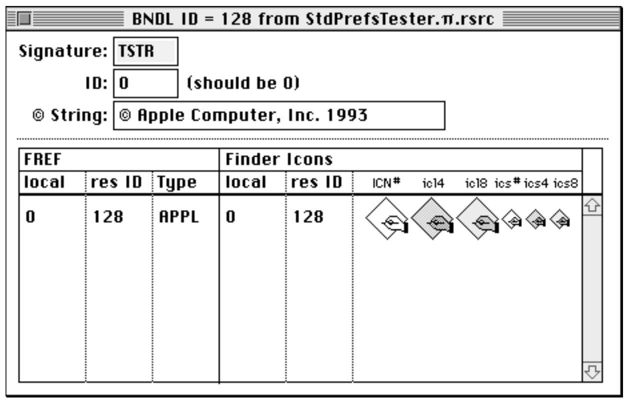
Figure 3. Application-unavailable alert

Our problem is that we don't want the Finder to launch our application when a user attempts to open its preferences file. One way to solve this (as suggested on pages 7-29 to 7-30 of *Inside Macintosh: Macintosh Toolbox Essentials*) is to register a "dummy" application signature that's different from the real signature of your application, and then set this signature as the preferences file creator.

Normally, an application has only a single signature resource and a single bundle resource. By creating a second bundle resource with a dummy application signature and associating our preferences file with that signature, we fool the Finder into looking for a nonexistent application for our preferences file, thus preventing the Finder from launching our application when the preferences file is double-clicked. Figure 4 shows examples of a normal application bundle resource and a dummy bundle resource for the application's preferences file.

Remember to register both creators — the real one and the dummy one — with the Developer Support Center (AppleLink DEVSUPPORT, or Apple Computer, Inc., Creator/File Type Registration, 20525 Mariani Avenue, M/S 303-2T, Cupertino, CA 95014).

If you use this technique, you should also supply an application-missing message string resource — an 'STR ' resource with an ID of -16397 — in the preferences file. When the user tries to open a file and the application can't be found, the Finder looks for this resource in the file. That string is displayed in the alert that comes up when



BNDL ID = 128 from StdPrefsTester.π.rsrc

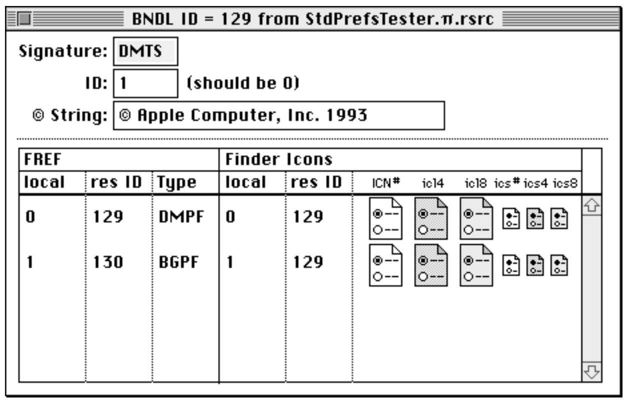
Signature: TSTR

ID: 0 (should be 0)

© String: © Apple Computer, Inc. 1993

FREF			Finder Icons							
local	res ID	Type	local	res ID	ICN#	icl4	icl8	ics#	ics4	ics8
0	128	APPL	0	128						

Application bundle resource



BNDL ID = 129 from StdPrefsTester.π.rsrc

Signature: DMTS

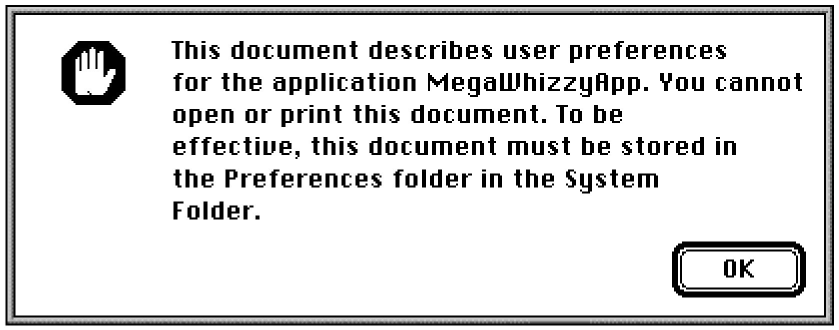
ID: 1 (should be 0)

© String: © Apple Computer, Inc. 1993

FREF			Finder Icons							
local	res ID	Type	local	res ID	ICN#	icl4	icl8	ics#	ics4	ics8
0	129	DMPF	0	129						
1	130	BGPF	1	129						

Dummy bundle resource for preferences files

Figure 4. Bundle resources



This document describes user preferences for the application MegaWhizzyApp. You cannot open or print this document. To be effective, this document must be stored in the Preferences folder in the System Folder.

Figure 5. Can't open preferences file alert

the user tries to open the file and — due to the dummy bundle resource — fails (see Figure 5).

Now for the catch: There are two ways this mechanism can be circumvented. First, it can be overridden if Macintosh Easy Open is running. Macintosh Easy Open uses the Translation Manager to open a file with an alternate application if the application that created the file isn't available. It determines the file type of the file that the user is trying to open and then looks for applications that support that file type by checking their bundle resources. If no applications can be found that support the file type, an alert notifying the user is displayed (see Figure 6). However, if any alternate applications are found, an alert allowing the user to open the file with one of them is displayed (see Figure 7). When Macintosh Easy Open is installed, one of these two alerts will always be displayed in place of the normal application-unavailable alert.

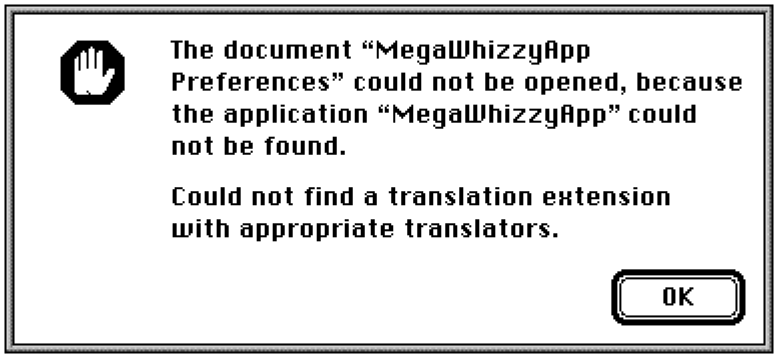
Because our application has a dummy bundle resource that refers to the file type of our preferences file, Macintosh Easy Open will always find the application that created the preferences file as an alternate application. Fortunately, the Translation Manager provides a mechanism to help us hide our preferences file — the 'open' resource. In the 'open' resource, an application can specify to the Translation Manager exactly what file types it can actually open, regardless of what may be specified in any bundle resources. By creating an 'open' resource that doesn't contain our preferences file type, we can prevent Macintosh Easy Open from trying to open the file.

The Rez definition for the 'open' resource is given below.

```
type 'open' {
      literal longint;                /* App signature */
      integer = 0;
      integer = $$CountOf(typeArray); /* Array size */
      wide array typeArray {          /* File types that app can open */
            literal longint;          /* File type */
      };
};
```

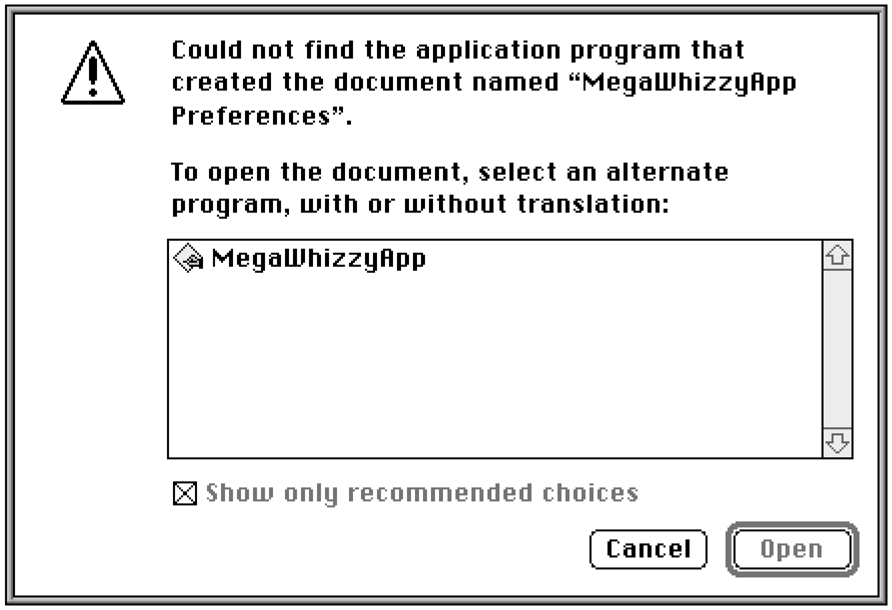
An example 'open' resource might look like this:

```
resource 'open' (128) {
   'TSTR',
   {
      'TEXT'
   }
};
```


The document “MegaWhizzyApp Preferences” could not be opened, because the application “MegaWhizzyApp” could not be found.

Could not find a translation extension with appropriate translators.

Figure 6. Modified application-unavailable alert


Could not find the application program that created the document named “MegaWhizzyApp Preferences”.

To open the document, select an alternate program, with or without translation:

MegaWhizzyApp

☒ Show only recommended choices

Figure 7. Translation choices alert

In this example, we've declared that the application with creator 'TSTR' can only open files of type 'TEXT' — the Translation Manager will not attempt to use this application to open any other type of file.

Our mechanism is also circumvented if the file type of the preferences file is 'PICT' or 'TEXT' and TeachText is installed. In this case, the Finder displays an alert asking whether the user wants to open the document in TeachText, even if you supplied an application-missing message string resource in the file and provided the appropriate dummy bundle and 'open' resource in your application. This is a pretty good reason for not using 'PICT' or 'TEXT' as the file type for your preferences file.

WHAT GOES IN A PREFERENCES FILE

If you find the size of your preferences file breaking the megabyte (or even the 100K) barrier, you're probably keeping information in the file that really doesn't belong there. What follows are a few thoughts on what to put in your preferences files.

Keep your preferences as resources.
As we all know, you can put program data into either the data fork or the resource fork of a Macintosh HFS file. The popular response to the question "What is a resource?" is "Everything!", and this is still reasonable advice. It's fairly easy to design C structs that mirror the structures of your preferences data, and you can even design 'TMPL' resources that will let you look at your raw preferences data in a structured way by opening your preferences file with ResEdit. Plus, you've got the Resource Manager to do all the work of finding and loading the preferences data into memory for you. What more could you ask?

Keep only global user preferences in your preferences file.
Bear in mind that some user preferences are global (application or environment related) and some are local (document related) — don't confuse the two. For example, keeping track of the size and position of every document window that's ever been opened by your application in its preferences file isn't a good idea (this information should be kept with the document file corresponding to the document window), but keeping a user-specified default window size and position in its preferences file *is* a good idea.

Avoid keeping machine-specific information in your preferences file.
It's rare that you really need to keep information such as the system software version or the Macintosh model that your software is running on in your preferences file. Your application generally should examine its environment at launch, and take appropriate measures at that time to avoid using features that aren't supported in the current environment.

Be careful of storing dynamic information in your preferences file.
A good example of something not to store in your preferences file is a volume reference number. To quote *Inside Macintosh: Files*, "A volume reference number is valid only until the volume is unmounted — if a single volume is mounted and then unmounted, the File Manager may assign it a different volume reference number when it is next mounted." In this example, a better solution is to store an alias to the volume of interest in the preferences file.

Use discretion in deciding what preferences to keep.
To a large degree, this means limiting user options to those that actually improve the overall user experience. The only proven way to find out what those options are is to conduct user testing on a variety of interface prototypes. It may turn out that while you thought keeping track of the last known number of documents open during the

user's last session with your application was important, it actually adds little or nothing when translated into the user's experience.

Don't keep static program data in your preferences file.
A file that just contains a lot of canned program data (such as tax forms) is *not* a preferences file — it's simply a program data file, and it should normally reside in the same folder as the application. That way, the user doesn't disable the application if she throws away what she thinks is simply the application's preferences file (users often do this to cause an application to reset back to its "factory" defaults, which, when you think about it, is a pretty nasty way to make users reset a program).

THE STANDARD PREFERENCES LIBRARY

We'll now take a look at the standard preferences library, a library for creating and interacting with preferences files that follows all the rules we've set in the preceding sections. The library API consists of 11 calls, described in detail below.

CREATING AND MANIPULATING A PREFERENCES FILE

In the routines described here, you'll notice that a preferences file is primarily identified not by its name, but by its creator and file type. This is done to allow you to localize the name of your preferences file easily, without requiring any code or resource changes. Also, should a user inadvertently rename your preferences file, your application can still find it. Because preferences files are identified in this manner, it's important to remember to give each preferences file that your application uses a unique creator/file type combination. The search sequence for a preferences file is as follows:

1. If there's a Preferences folder in the System Folder, search it and any folders within it (including nested ones).
2. If there's no Preferences folder, or if no preferences file is found in the Preferences folder, search the files (not the folders) in the System Folder.

```
pascal OSErr NewPreferencesFile (OSType creator, OSType fileType,
   ConstStr31Param fileName, ConstStr31Param folderName,
   ConstStr31Param ownerName);
```

NewPreferencesFile creates a preferences file with the specified creator, file type, and filename in the System Folder (prior to System 7) or in the Preferences folder (in System 7 or later). In the folderName argument, you can specify a custom preferences folder in which to put a collection of preferences files; pass nil for this argument if you don't want to use a custom preferences folder. If a folder name is provided and the folder doesn't exist, NewPreferencesFile creates it. Another option is to provide, in the ownerName argument, the name of the program (application, extension, or whatever) that's creating the preferences file; if this argument is supplied, a custom application-missing message string resource is created for the preferences file.

```
pascal OSErr OpenPreferencesFile (OSType creator, OSType fileType,
   short *refNum);
```

OpenPreferencesFile opens the preferences file having the specified creator and file type. You must have created the file with NewPreferencesFile before making this call. The preferences file reference is returned in the refNum argument; you'll use this to identify which preferences file you're operating on when making other calls in the standard preferences library.

```
pascal OSErr ClosePreferencesFile (short refNum);
```

ClosePreferencesFile closes the preferences file having the specified file reference. That's it.

```
pascal OSErr DeletePreferencesFile (OSType creator, OSType fileType);
```

DeletePreferencesFile deletes the preferences file having the specified creator and file type. Be sure the specified preferences file is closed before making this call.

```
pascal OSErr DeletePreferencesFolder (ConstStr31Param folderName);
```

DeletePreferencesFolder deletes the custom preferences folder specified by folderName, along with any contents. Be sure there are no open preferences files in the specified folder before making this call.

```
pascal OSErr PreferencesFileExists (OSType creator, OSType fileType);
```

PreferencesFileExists simply checks to see if a preferences file having the specified creator and file type already exists. You can use this call to determine whether you need to create a new preferences file.

ACCESSING INFORMATION IN A PREFERENCES FILE

These routines get or set the indicated resource information in the preferences file specified by the refNum parameter.

```
pascal OSErr GetPreferencesFileVersion (short refNum, short versID,
   NumVersion *numVersion, short *regionCode, ConstStr255Param
   shortVersionStr, ConstStr255Param longVersionStr);
```

GetPreferencesFileVersion returns the contents of the specified 'vers' resource of the preferences file. Only 'vers' ID 1 or 2 is allowed.

For those of you wondering why GetPreferencesFileVersion doesn't simply pass a pointer to a VersRec (defined in Files.h) instead of passing each of the separate fields of a VersRec, it's because the short version string and the long version string are packed in a VersRec (it's possible that the shortVersion field of the VersRec will actually contain part of the long version string). GetPreferencesFileVersion takes care of unpacking the strings properly, and SetPreferencesFileVersion takes care of packing them properly.

```
pascal OSErr SetPreferencesFileVersion (short refNum, short versID,
   NumVersion *numVersion, short regionCode, ConstStr255Param
   shortVersionStr, ConstStr255Param longVersionStr);
```

SetPreferencesFileVersion allows you to set the contents of the specified 'vers' resource of the preferences file. Only 'vers' ID 1 or 2 is allowed.

```
pascal OSErr ReadPreference (short refNum, ResType resourceType, short
   *resourceID, Handle *preference);
```

ReadPreference reads from the preferences file the resource specified by the resourceType and resourceID arguments, and returns it in the preference argument. If you pass nil as the address of the resourceID argument or 0 as its value, ReadPreference finds the first resource with the specified type in the preferences file. If you pass 0 as the value of resourceID, ReadPreference returns the resource ID corresponding to the preference resource it found.

```
pascal OSErr WritePreference (short refNum, ResType resourceType, short
   *resourceID, Handle preference);
```

WritePreference writes the resource specified by the resourceType and resourceID arguments to the preferences file. If you pass nil as the address of the resourceID argument or 0 as its value, WritePreference assigns the preference resource a resource ID and returns it in the resourceID argument (if its address isn't nil). If a resource with the same resource type and ID already exists in the preferences file, the existing resource is replaced with the new one. WritePreference checks that there will be a minimum amount (the exact amount varies as a function of the volume's allocation block size) of disk space remaining on the volume containing the blessed System Folder before actually updating the preferences file; if there isn't enough disk space, the preferences file is unmodified and an error is returned. Note that it's the caller's responsibility to release the memory occupied by the preference argument; WritePreference does not dispose of this memory automatically.

```
pascal OSErr DeletePreference (short refNum, ResType resourceType, short
   resourceID);
```

DeletePreference deletes the resource specified by the resourceType and resourceID arguments from the preferences file. If you pass 0 as the value of resourceID, DeletePreference deletes the first resource with the specified type in the preferences file.

PLEASE PROVIDE PREFERRED PREFERENCES FILES (SAY *THAT* THREE TIMES FAST!)

In this article, we've explored what constitutes good design for preferences files, and we've examined a library that helps with implementing preferences files. Armed with this knowledge and the standard preferences library, you too can add preferences files to your application the preferred way!

REFERENCES

- *Inside Macintosh: Macintosh Toolbox Essentials* (Addison-Wesley, 1992), Chapter 7, "Finder Interface," or *Inside Macintosh* Volume VI (Addison-Wesley, 1991), Chapter 9, "The Finder Interface."
- *Inside Macintosh: More Macintosh Toolbox* (Addison-Wesley, 1993), Chapter 7, "Translation Manager."
- *Inside Macintosh: Overview* (Addison-Wesley, 1992), Chapter 3, "Resources," pages 60 to 67.
- *Macintosh Human Interface Guidelines* (Addison-Wesley, 1992), Chapter 3, "Human Interface Design and the Development Process," pages 37 and 38, and Chapter 8, "Icons," page 250.

Thanks to our technical reviewers Nick Kledzik, Jim Luther, and John Mitchell. Special thanks to Jim Luther for his excellent file system utilities (collect 'em all — they're on this issue's CD) and to Elizabeth Moller for her assistance on human interface issues.•

Macintosh Q & A

Q *I'm having trouble figuring out how to convert some assembly trap patches for selector-based traps into PowerPC native code. Do you have any suggestions for a clean way of doing this, ideally so that it works on both 680x0-based machines and the Power Macintosh?*

A Currently there's no way to patch a selector-based trap with a native or fat patch. (A list of selector-based traps can be found in the back of *Inside Macintosh X-Ref.*) The problem arises from the fact that each routine associated with a single dispatch-based trap can have a different parameter list (that is, a different number of parameters and different sizes for each parameter). Basically there's no way for mixed mode to handle the variable stack frame sizes associated with selector-based traps. This is the same thing that makes head/tail patching them so much of a pain in C.

We're in the process of trying to determine whether developers have a pressing need to patch selector-based traps with native code. For now, keep all such patches in 680x0 code. If a patch to a particular routine is itself very time-intensive (which is rarely the case), the 680x0 patch can call through to a native implementation.

Q *I'm writing a QuickDraw GX print driver for a plotter and need to send initialization and termination strings to the plotter. How can I determine from my driver if I'm printing the very first or last page, regardless of the number of copies? I'd like to have this information in GXStartSendPage and GXFinishSendPage, respectively, so that I can send my strings then.*

A We recommend that you do any pre-first-page setup in GXOpenConnection (after forwarding) and any post-last-page teardown in GXCloseConnection (before forwarding). (Although the documentation is a bit ambiguous on this point, you *can* send information with GXBufferData and GXWriteData from GXCloseConnection, before forwarding.) If there's some reason that this won't work for you, and you really need the information in GXStartSendPage and GXFinishSendPage, you'll have to use global data.

In some override before GXStartSendPage (perhaps GXImageDocument), initialize a global page counter to 0. In your GXStartSendPage override, check this flag, and then bump it after forwarding. If your check finds that the flag is 0, no pages of the document have been sent yet.

To determine when the last page has printed, you'll also need to use global data. Somewhere (again, GXImageDocument is fine) call GXCountPages, get the number of copies from the job's 'copy' collection item, and multiply. In your GXFinishSendPage override, compare the value you bumped in GXStartSendPage to this multiplied value. When they're equal, you're just finishing the last page.

Q *How do client and server desktop printers (shared printers) synchronize in QuickDraw GX? Specifically, what we're trying to find out is (for a server desktop printer on one machine, and a matching client on another):*

- *If a user on the client machine sets a papertype in an input tray, is it reflected on the server in the Input Trays dialog? If not, what's the correct behavior for the client and the server machines (for example, should the client Input Trays dialog be read-only)?*

- *Generally, what resources and data are transmitted between the host and the client, and when? Is there a mechanism for controlling which resources will be sent or kept local (preferably on a resource-by-resource basis)?*

A Resources are transmitted to desktop printers in only one direction — server to clients. Also, only resources with IDs greater than 0 are moved to the clients. Therefore, it's appropriate to make the Input Trays dialog on clients read-only. Even though Apple's drivers don't do this, it's the more correct approach. You can find out whether you're on a server or a client by checking the desktop printer's 'comm' resource; if its type identifier is 'ptsr', you're on the client; otherwise, you're on the server.

If you need to have data sent from the client to the server, you should fetch the resources at GXImageJob time (before forwarding) and then roll them into a job collection item. In the appropriate communication message, look for the collection item and use that data. Since GXImageJob is always called (shared printers or not), and it's called on the client if you're working with shared printers, this method should always work.

Q *If I want to add additional properties to the paper stock (such as paper color), and I call AddCollectionItem(GXGetPaperTypeCollection(paper), . . .), will that new collection be stored in the desktop printer's configuration file, or is that something I must manage? I seem to be losing my collection between invocations of the Trays dialog.*

A The papertype collection items you add won't be flattened to disk and stored in the desktop printer via your Trays dialog. There's no way to make collection item changes and have them saved with the disk-based papertype, wherever it may be stored. You need to manually save the information in your desktop printer (or some other place) as resources, and then match that up with the papertypes when you want to use them. You should match up the papertypes and the resources based on the names of the papertypes.

You can still take advantage of the papertype collection to hold your paper color information, as long as you also have the information stored on disk. The papertype collection will be around as long as the papertype's job is around. For example, if you load the color info from the desktop printer and put it in a papertype collection item at despoolpage time, it stays there throughout the entire print cycle, and wherever the job goes, it goes.

Q *I'm having a problem resizing text elements in our QuickTime application. I'm trying to modify the element's size by calling SetTrackDimensions, and it seems to do what I want for all element types except text. For text tracks, the element's bounding box is resized correctly, but the text characters are scrunched into the upper left corner of that rectangle, still at their original size. In other words, SetTrackDimensions seems to scale the track bounds, but not the text characters themselves. Any idea what's going on?*

A This is a bug. As you determined, SetTrackDimensions is only changing the size of the track box, not setting the correct scaling factor or internal flags. To work around this problem, use GetTrackMatrix to retrieve the current matrix, then ScaleMatrix to change it, and finally SetTrackMatrix to make it take effect.

Q *Do you know why "OCE Mail Enclosures" appears as a volume when I index through all volumes using PBHGetVInfo? Is there any way to filter out this "volume"?*

A The reason “OCE Mail Enclosures” shows up is that it’s the volume for an external file system (XFS) that AOCE installs in order to support access of letter enclosures via FSSpecs from the mailer and other parts of the AOCE system. This enables the direct access that the mailer provides in the enclosure fields of letters, allowing users to manipulate enclosures like any other file in the Finder, copying and even launching them directly from the enclosure pane.

To filter out the “OCE Mail Enclosures” volume, you should check the Finder flags of the root directory in each volume to determine whether that volume should be visible to the user. The Finder flags for directories are located in the ioDrUsrWds field in the dirInfo variant of the CInfoPBRec structure. If the fInvisible bit is set, you should not display that volume to the user. Here’s a snippet:

```
void main(void)
{
   HVolumeParam   pBlock;
   CInfoPBRec     cBlock;
   Str255         volName, fName;
   OSErr          err;

   pBlock.ioNamePtr = volName;
   err = noErr;
   for (pBlock.ioVolIndex=1; err==noErr; pBlock.ioVolIndex++) {
      err = PBHGetVInfo((HParmBlkPtr)&pBlock, false);
      if (err==noErr) {
         cBlock.dirInfo.ioNamePtr = fName;
         cBlock.dirInfo.ioVRefNum = pBlock.ioVRefNum;
         // Query the directory info ioDrDirID.
         cBlock.dirInfo.ioFDirIndex = -1;
         // This is the root directory.
         cBlock.dirInfo.ioDrDirID = 2;
         err = PBGetCatInfo(&cBlock, false);
         if (err==noErr) {
            if ((cBlock.dirInfo.ioDrUsrWds.frFlags &
                  fInvisible)!=0)
               // It's invisible.
               . . .
         }
      }
   }
}
```

Q *I have a dialog with two editText fields. When I populate the two fields with text, whichever field I populate first is displayed two pixels too high within its item. The second field is fine, and it has the “focus” of the dialog. When I click in the first field, any new text is added at the correct height, but unfortunately that’s two pixels below where the previous text was drawn. The fields are both 10-point plain Geneva, and the editText boxes are 16 pixels high. Any ideas?*

A The Dialog Manager has a bug that causes problems when you use an alternate font or size for the editText items. The problem is how it draws the text initially in the dialog: the text for the currently active item is drawn by manipulating the dialog’s TextEdit record, and the text for all other items is drawn by calling

TextBox. The solution is to call SelIText just before you call SetIText each time you populate a field with text.

Q *How can I convert an RGB color into an index to a palette created by my application? Color2Index converts the RGB color to an index to the current device's color table, but that's not what I want.*

A There's no single call that will give you a palette match to an RGB color. You'll have to do this: call Color2Index to get the closest match to your RGB request; call Index2Color to get the device's indexed color from your match; search the palette yourself to find the color match (according to RGB value); and call Color2Index to verify that you have the color you're looking for.

Alternatively, you can create an off-screen GWorld, call Palette2CTab to convert your palette to a color table, and call UpdateGWorld to insert your new color table in your off-screen GWorld. Then, to find the index of an RGB color, make your GWorld the active device and call Color2Index.

Q *I've tried in vain to find a way to print white text on a black background. Is there a way to do this, and if so, how?*

A The trick is to use the srcBic pen mode:

```
FillRect(theRect, black);
PenMode(srcBic);
DrawString(myString);
```

Q *In our application, the user can select an area of an image and drag it around. I want to show this visually by inverting the region under the current mouse coordinate as the user moves the mouse around. Inverting the region is nice because I can invert it again to get the unselected pixels back. It's not nice, however, in that a 50% gray color looks the same when it's inverted. To fix this problem, I tried using PaintRgn with an RGBForeColor of r,g,b = 0x8000 and a transfer mode of addOver. This works great on 24-bit screens, but it seems that on 256-color screens, applying this operation twice doesn't quite return to the original color. Am I going to have to use a custom color search procedure?*

A You get the results you want on direct devices but not on indexed ones, and unless you're extremely lucky with your color table, this is how it will always work. The problem is that the mode calculations are done with the actual RGB values used (the ones available in the color table), not the ones you request. On indexed devices there's almost always a difference between the two, so unless your color table happens to have the exact color you request, there will be "errors." This never happens on direct devices because all colors are available — the operations work on direct RGB values and are never mapped through color tables.

The solution is either to set up your color tables or palettes to make sure you get the results you want each time, or to install a custom color search procedure if that's what you'd prefer.

Q *After we call CMOpen and a connection is established, a dialog is displayed and eventually goes away. Unfortunately, the C++ object framework we use is bombing* *because it's getting a deactivate event for that window, which belongs to the Communications Toolbox. We wrote a kludge that sets a flag after the call to CMOpen is finished and eats the deactivate event if the flag is set. Is there a better way for us to tell whether to let the class library handle the event or to handle it ourselves?*

A A window or dialog created by a connection tool has the connection record handle stored in the refCon field. The sequence, then, is to check the event record to find out if the event is tied to operations in a window and, if so, check the window's refCon against your connection handles. If there's a match, call CMEvent for that event; otherwise pass it on to the framework. You'll probably need to write a handler for your class library to do this properly, overriding the default window-handling routines for this special case.

Q *When our application opens a Communications Toolbox tool, we issue a CMOpen with the asynchronous flag true and go into a loop, calling CMIdle and then CMStatus until we see the cmStatusOpening flag go down or the cmStatusOpen flag come up. When we use the Express Modem Tool (on a Macintosh Duo 230), those flags never change. Should we be doing something different or is there a problem with that tool?*

A The Express Modem Tool uses a background process (coupled tightly with the hardware implementation) to actually move data. Unless your application yields processor time through the WaitNextEvent cycle, the background process is stuck when you call the tool asynchronously. (The synchronous call has been massaged to give the process time, of course.)

What you're doing, essentially, is making the asynchronous call synchronous by trapping your application in this kind of loop. The proper thing to do would be either to use the call synchronously or to continue to use it asynchronously but exit back to the main event loop and look for the flags from there. When the appropriate flag is set, you can then dispatch off to a handler routine. Even better, use a completion routine to notify the application that the CMOpen has completed and obtain the function result from the ConnHandle errors field.

Q *I've implemented a variant of the CMChoose dialog based on the Choose.p sample code in Inside the Macintosh Communications Toolbox, page 323. The problem I have is that all the fields of the dialog appear in 12-point Chicago rather than the 9-point Geneva that tool dialogs usually use, so the dialog looks really tacky. How can I fix this?*

A The critical thing is knowing when and where to set the window's text characteristics. Tools provide a resource ('finf' or 'flst', defined in SysTypes.r and CTBTypes.r) that gives you the font information for the tool's DITL. Between the CMSetupPreflight and CMSetupSetup calls, you should fetch the font, size, face, and mode from the resource and set your custom dialog's port to match it. You also need to stuff the same information into the dialog's TextEdit record so that the editable fields show up correctly. Controls provided in a DITL by Communications Toolbox tools have the useWFont bit set so that they always follow the settings in the dialog's port.

Q *We're calling CMListen synchronously in our application, and if it times out an error alert is displayed that doesn't go away until the user clicks OK (or after a very long time). Is it possible not to have this dialog displayed, or to have it go away quickly as the "connected" dialog does?*

A With regard to all Communications Toolbox interface components, you can only leave them all on or turn them all off with the flag parameter to CMNew (cmQuiet and cmNoMenus). We don't know of any way to affect the behavior of specific elements like the error dialog raised by CMListen. (CMListen is best implemented in an application as an asynchronous call, particularly in the cmQuiet mode.)

Q *The Connection Manager sample code in Inside the Macintosh Communications Toolbox sets the buffer sizes for cmDataIn and cmDataOut to 1K and the rest to 0, and there's a comment that the other channels are to be ignored. Then, in the description of CMNew, it says, "To have the tool set the size of these buffers, your application should put zeros in the array." What's the recommended way to go?*

A You should consider the buffer sizes you set in the CMBufferSizes array as a request for buffers. The tool's implementation will always override your choices based on what the developer felt was the proper thing to do. Many programmers initialize the array to all zeros and let the tool defaults be set; there's some argument for increasing the sizes on network protocols for efficiency, but it's up to the tool designer to determine what makes the best sense. Rather than depend on any particular buffer sizes in your application, you should deal with what the tool allocates dynamically.

Q *What is that green slime that you can buy in toy stores made of?*

A We're not exactly sure what the composition of that stuff is, and the toy companies aren't about to tell us, but we're pretty sure that it's some sort of polymer that's cross-linked via hydrogen bonding. Hydrogen bonds are relatively weak, and can be easily pulled apart. That's why these materials behave like "slow liquids" and eventually seek their own level.

A very satisfying white version of slime can be concocted at home from common ingredients as follows: Mix 1/2 cup water and 1/2 cup white glue in a bowl. In another bowl (or a cup) dissolve 1 teaspoon borax in 1/2 cup water (make sure the borax is completely dissolved; it may take a minute of stirring). Pour the borax solution into the glue solution, stirring rapidly and constantly. Keep stirring for a minute; then reach in with your fingers and keep mixing, trying to break up the lumps. At first the material will be lumpy and wet, but soon it will become smooth and rubbery. This recipe makes a blob the size of a grapefruit. Use less water with the glue for stiffer slime. Store in an airtight container, and keep it away from carpets!

Q *Nothing in the documentation for MacTCP deals with the state of register A5 if an ioCompletion routine is specified within struct CntrlParam (MacTCP version 1.1 documentation, page 7). I've been manually preserving and setting A5 with some inline assembly code but wonder if this is really necessary. If I must set A5, and want to use the same source code in the 680x0 and PowerPC environments, how do I go about it?*

A When MacTCP calls the application's ioCompletion routine, it restores the application's A5 register, so the application shouldn't worry about this (the MacTCP driver takes care of it). On a Power Macintosh, you can still set register A5 in the emulator as you did before (with SetA5 and SetCurrentA5). However, be aware that with native code, register A5 is no longer used to store references to global variables. Any piece of PowerPC native code, even standalone code, can have its own global variables without making its own A5 world.

Q *When a driver is operating synchronously, what kinds of system calls are prohibited? Specifically, can I make memory allocation calls and file system calls from the driver?*

A If your driver is called synchronously, you should be able to allocate memory and make file system calls and other system calls that move memory. If it's called asynchronously, you should *not* make these calls. There's one important exception to this guideline: the Macintosh file system isn't reentrant, so in a disk driver or a network driver that serves the file system, you must not make any calls to the file system, as you will tie it into metaphorical knots.

Q *I'm creating an application from an existing file by adding CODE resources to it, setting the bundle bit, clearing the inited bit, closing the file, and flushing the volume. My problem is that the Finder doesn't recognize the change immediately. I have to move the file to another folder before the icon changes and it's recognized as an application. What do I need to do to have the Finder recognize the change immediately?*

A The problem you're having stems from the fact that the Finder only scans for changes about every 10 seconds. To make the Finder aware of changes before that, you need to change the modification date of the parent directory. Use a routine like this:

```
OSErr TouchDir(short vRefNum, long dirID)
{
   CInfoPBRec     info;
   Str255         name;
   OSErr          theErr;

   info.dirInfo.ioDrDirID = dirID;
   info.dirInfo.ioVRefNum = vRefNum;
   info.dirInfo.ioNamePtr = name;
   info.dirInfo.ioFDirIndex = -1;
   theErr = PBGetCatInfo(&info, false);
   if (!theErr) {
      info.dirInfo.ioCompletion = 0;
      info.dirInfo.ioDrDirID = info.dirInfo.ioDrParID;
      info.dirInfo.ioFDirIndex = 0;
      GetDateTime(&info.dirInfo.ioDrMdDat);
      theErr = PBSetCatInfo(&info, false);
   }
   return theErr;
}
```

The Finder will rescan the specified directory immediately after this routine updates the modification date, usually well within one second.

Q *Does the idleProc of AESend get called before every event is sent, even those to the current process (which are directly dispatched)? What I really care about is whether WaitNextEvent is called each time.*

A AESend's idleProc will be called if kAEWaitReply is the sendMode. In this mode the Apple Event Manager uses the Event Manager to send the event. The

Event Manager then calls WaitNextEvent on behalf of your application. This causes your application to yield the processor, giving the server application a chance to receive and handle the Apple event. You must supply an idleProc in order to process any update events, null events, operating system events, or activate events that occur while your application is waiting for a reply.

If you use kAENoReply or kAEQueueReply as the sendMode, AESend will immediately return after using the Event Manager to send the event. Your idleProc will never be called (in the case of kAEQueueReply, it's assumed that you want to receive your reply via your application's event queue, and you must install a handler for the reply Apple event).

Likewise, your idleProc will never be called in the case of a direct dispatch. In doing a direct dispatch you're sending an Apple event to yourself using the typeProcessSerialNumber and kCurrentProcess. These events are delivered directly, bypassing the event queue and executing your handler routine directly. For more information, see the Macintosh Technical Note "SendToSelf: Getting in Touch With Yourself Via the Apple Event Manager" (Interapplication Communication 1).

Q *I'm having a problem sending custom Apple events over the network. We have a background-only application on one machine sending custom Apple events to another machine via LocalTalk. If we manually pull the LocalTalk cable out from the back of the sending Macintosh, the event is never received on the remote machine, but an error is never returned by AESend. AETracker logs show that the event is being sent with no error, but AETracker on the remote machine shows no sign of the event. We've also seen a similar thing happen when phone lines are bad. Note that in both cases, other events are sent between the machines just fine. What gives?*

A The reason AESend doesn't report an error in cases like the one you mention is because it can't. As soon as the event is sent, AESend returns noErr indicating that the event has been handed off to the PPC Toolbox for sending. You're then back in your main event loop and doing other things. If for some reason the connection goes down (or there's any other transmission problem), there may be a resulting error from the network layer that's actually transporting the event, but the resulting error may not occur for seconds or even minutes. At that point there's no way for the AESend that sent the event to detect the error.

We saw a good example of this recently using a standard Ethernet connection between two machines. The network connection was broken between the machines and an event was sent from one to the other. AESend returned noErr on the sending Macintosh, and as long as we reconnected the two machines before the end of the associated timeout period — two minutes — the event was received. If we waited longer the event never made it. But in either case AESend returned noErr.

There are a couple of ways to address the problem. The first way is to use kAEWaitReply when sending your event; however, you give up the processor in favor of ensuring a reply. The other solution is to pass kAEWantReceipt in the sendMode parameter of AESend and have a timeout for the amount of time you're willing to wait for a reply.

Q *We have two questions regarding AppleScript. First, what's the significance of Begin Transaction and End Transaction for a single-threaded application? Do we need to* *support these two events if we don't send events or scripts to another application? The Apple Event Registry says to return a transaction ID for the Begin Transaction call and to check for the transaction ID of all the incoming Apple events. Is this really necessary? Second, what's the user interface guideline for the Print Document ('pdoc') event in the required suite? Currently, I bring the application to the front by calling AEInteractWithUser (only if both the server and the client are in the same machine) and open the print job dialog box.*

A A single-threaded application doesn't need to support Begin or End Transaction. If you need transactions, you may implement them as you see fit. Regarding the 'pdoc' event, what you're doing is correct. When you receive a 'pdoc' event, you should call AEInteractWithUser and check the result: if you get noErr (meaning you can interact), open the standard print job dialog; if you get errAENoUserInteraction (you can't interact), just do whatever the default is for that document, printing without interaction.

Q *I'm doing a project where I need about 1500K for my own off-screen GWorlds and sundry data structures, and we're targeting the 4 MB Macintosh LC. Here's the kicker: we need Text to Speech. I'd like to know more precisely how the memory allocation works in the Speech Manager so that I know what our options are — for example, how much memory gets allocated from the system versus how much from my application heap.*

A When trying to preflight the memory needs for an application that uses Text to Speech, keep in mind that there are at least three different managers involved in the production of speech on the Macintosh: the Component Manager, the Speech Manager, and the Sound Manager. All these have their own memory allocation schemes and take memory from different places.

As a rough rule of thumb, to use Text to Speech in a robust manner, plan on adding 250K for each SpeechChannel you expect to keep open at any given time; this should accommodate both MacinTalk 2 and MacinTalk Pro voices. If this causes a minimum application size that's not acceptable, you can add only 50K for each MacinTalk 2 channel you allow to be open at a time and include in your documentation instructions on how to increase the size if the user decides to use MacinTalk Pro voices instead.

The more complete scenario goes like this: The Component Manager takes up about 20K of the system heap (possibly slightly less when no components are open). The Speech Manager code and data use around 20K of system heap, and should be a one-time investment (note that little variance should be expected from version to version). The Sound Manager memory usage depends on the version and other factors, but a good estimate is 30K per SndChannel. Note that the Sound Manager code goes into the system heap, with sound buffers and sound data being allocated in the application heap.

The amount of space needed for the Text to Speech engine code and data (such as pronunciation dictionaries and rules data) varies quite a bit between MacinTalk 2 (about 100K) and MacinTalk Pro (about 300K); this memory is allocated in the system heap whenever possible to make it available to different applications using the same engine. If the Speech Manager can't allocate the necessary space in the system heap, it tries to get it from the application heap. Naturally when this happens the code and data cannot be shared across applications. There's also some Text to Speech engine–specific SpeechChannel data whose size varies from engine to engine: MacinTalk 2 uses roughly 10K and MacinTalk Pro uses about 175K. Finally, there's the voice data and code: MacinTalk 2 takes between 20K to 40K depending on the chosen voice, while MacinTalk Pro voices can use between 300K and 2.25 MB of RAM. Again, the memory needed for voice data and code is allocated from the system heap if possible to allow sharing between applications using the same voices; if there's no room in the system heap, the Speech Manager tries to load this in the application heap, and no sharing is possible.

Finally (you knew this was coming, didn't you?), be aware that these numbers may change in the future. Use them as a guide, but as always, don't depend on them.

Q *What should we do when renaming a document that contains a publisher section? I try to call AssociateSection with the already registered section and the new FSSpecPtr. AssociateSection returns no error and I unregister the publisher section. But the next time I open up and register the publisher section of that document, I get a -463 error code on RegisterSection. What am I doing wrong?*

A AssociateSection doesn't change any information in the edition container file, which is where this needs to be changed; it only acts on the "hot" links the Edition Manager is currently maintaining with any open documents that are using a section. What you have to do if you rename a publishing document is to open the publisher and update it. When you call OpenNewEdition with the new file name, that will update things. This means that a "save as" must dirty all your publisher sections, which slows you down a bit and may be counterintuitive. But the only other option would be to update the alias directly, and that would be bad.

Q *I'm working on a video-conferencing solution that uses the video digitizer (vdig) incorporated in the Macintosh Quadra 840AV. I want to capture data from the system's built-in video hardware using the VDCompressOneFrame and VDCompressDone calls. I have the following questions about the vdig that supports the 840AV built-in video hardware:*

- *What's the header and data format for the captured video?*
- *What's the compressor type (cType) for this compression format?*
- *Does this compressor support more than one spatial compression setting and, if so, what are the data formats for the compression settings?*

A We can't provide information regarding the data format of the captured video. It's considered proprietary and confidential, except in cases where the codec in use is an industry standard like JPEG. Fortunately, you don't need to know the data format if you're using the correct QuickTime vdig and Image Compression Manager calls to manipulate the data.

We don't think you should use the vdig directly, but if you do, you can call VDGetCompressionType to determine the compression types it supports. You can select the compression type you want to use by calling VDSetCompression. Since the vdig uses standard codecs for compression, you don't need to know the data format; all you have to do is use the codec to decompress the image data when you want to draw it. Call VDGetImageDescription to get an image description handle, which you can pass to DecompressImage along with a pointer to the data, and the Image Compression Manager will take care of decompressing the data as long as the correct codec is available.

We don't recommend using vdigs directly because every one is different and supports different features. They can be pretty hard to work with because your code will require a lot of error handling and workarounds. The sequence grabber was written to provide a seamless interface between any vdig and applications, so you can use the sequence grabber as the engine for your video-conferencing system. It was designed with this kind of flexibility in mind. For more information about the sequence grabber, see Chapter 6, "Sequence Grabber Channel Components," in *Inside Macintosh: QuickTime Components*.

Using the sequence grabber with the right flags, you can get high-performance grabs, even over the network. You do this by supplying application-defined functions to the sequence grabber component. If you replace the grab function on the receiver side, you can use the sequence grabber to grab right off the network on that end. On the sender side, you can replace the data function so that you'll be able to write the frames out over the network, using whatever network protocol you like.

These answers are supplied by Apple's Developer Support Center. Special thanks to Brian Bechtel, Matt Deatherage, Godfrey DiGiorgi, Steve Falkenburg, Dave Hersey, Dave Johnson, Scott Kuechle, Joseph Maurer, Kevin Mellander, Jim Mensch, Martin Minow, Guillermo Ortiz, and Brigham Stevens for the material in this Q & A column. If you need more answers, take a look at the Macintosh Q & A Technical Notes on this issue's CD.•

apda
tools for developers

Your main source for Apple development products

Get easy access to *New Inside Macintosh* and over 200 other programming products, tools, technical resources, and information through APDA, Apple's worldwide source for Apple and third-party development products.

Ordering is easy, and APDA offers convenient payment and shipping options, including site licensing.

Call for additional information on APDA or recently announced products, or to request a complimentary copy of the *APDA Tools Catalog*.

Call APDA today.
United States 1-800-282-2732
Canada 1-800-637-0029
International (716) 871-6555

Newton Q & A: Ask the Llama

Q *Here's something that's been puzzling me a bit: I want to pop up something like a copyright message for ten seconds when my application starts up. So I drew up a layout called Presents with a protoFloater containing all the necessary text. In my main layout is a link to Presents called presentsLink. The following is the viewShowScript for the topmost view in my application:*

```
func()
begin
   presentsLink:open();
   AddDelayedAction(presentsLink:close(), nil, 10000)
end
```

This says to me: open the linked view, wait ten seconds, and close it. And that's exactly what it does, except that after the view is closed, there's an exception. What am I doing wrong and how do I fix it?

A The short answer is that the second argument to the AddDelayedAction function is of the wrong type. This argument is supposed to be an array of parameters to be passed to the delayed function, and nil is not an array. The proper syntax for no arguments is [] instead of nil.

But there's more: Although the closure you supplied in the first argument works in this case, you should get out of the habit of using that type of function call for delayed or deferred actions. You're better off providing a full closure and sending in the view you want closed, as in this:

```
func()
begin
   // Define a closure to use in the delayed action.
   local myClose := func(whichView)
      whichView:Close();

   presentsLink:Open();
   AddDelayedAction(myClose, [presentsLink], 10000);
end
```

Unfortunately, things do not end there. You also need to make sure that the myClose function is in internal RAM. The best way to do this is to use DefConst to define the function:

```
// In your ProjectData file:
DefConst('kDelayedClose, func(whichView) whichView:Close());

// This changes the function above:
func()
begin
   presentsLink:Open();
   AddDelayedAction(EnsureInternal(kDelayedClose),
      [presentsLink], 10000);
end
```

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Personal Interactive Electronics (PIE) division. Send your Newton-related questions to the PIE llama at AppleLink DR.LLAMA (on the Internet, dr.llama@applelink.apple.com). The first time we use a question from you, we'll send you a T-shirt.•

However, there is an easier solution. Instead of doing a delayed action you can use the view idle mechanism to do what you want. All you need to do is add a viewIdleScript and viewIdleFrequency to the presentsLink top-level view. The viewIdleScript simply sends a Close message. The viewIdleFrequency is set to the desired delay (10000 in this case).

The really short answer is that you can just use protoGlance. This proto has the show-and-disappear behavior built in. You can set the viewIdleFrequency to 10000 to get the behavior you want.

Q *I have an alphabetically sorted list of items in a protoTable and I want to use protoa2z to quickly move through the table (like the cardfile overview). How do I do it?*

A The first thing you need to do is find the index in the protoTable of the correct item (that is, find the right item in the def.tabValues array of the protoTable). How you do this will depend on what type of data you're representing. Then you can figure out how high each item in the protoTable is, set the VOrg slot of the table to the correct line, and force the protoTable to redraw. You probably want to make sure that you don't scroll off the bottom of the table. Below is a method you can add to your protoTable that will do what you want. You can then send the message from your a2zChanged method (make sure you send the message to the protoTable).

```
func(index)
begin
   // Figure out the height of an item in the table.
   local childHeight := if def.tabProtos.viewFont exists then
         fontHeight(def.tabProtos.viewFont)
      else
         fontHeight(viewFont);

   // Make sure that the table will not scroll off the end
   // by calculating the index of the bottommost item that
   // will be displayed.
   local largestIndex := def.tabDown - ((:LocalBox().bottom -
         :LocalBox().top) DIV childHeight) - 1;

   // Use the bottommost item (largestIndex) to make sure
   // the table has no empty space on the bottom.
   vOrg := MIN(index, largestIndex);

   // Now force the table to redraw.
   :RedoChildren();
end
```

Q *I would like to use the protoa2z sample code in my application, but I can't figure out how to set the highlighted letter when the user scrolls through my data.*

A It's easy once you realize that protoa2z is based on a protoPictIndexer. All you have to do is a SetValue of the currIndex slot to the correct index. This will change the highlighting of the protoa2z.

Note that using SetValue will not call the IndexClickScript, but this is probably what you want. If you do want it to be called, you'll have to manually unhighlight the current selection, set the currIndex slot, and then highlight the new item. The appropriate code would be

```
a2z:Unhilite();
a2z.currIndex := newIndex;
a2z:Hiliter(newIndex);
```

Q *What is your quest?*

A To answer the questions of those who develop for Newton.

Q *I tried to use a protoPictRadioButton in my application but the highlight rectangle isn't the right size. I know that I need to write some code to draw the correct-sized highlight, but where do I hook it in?*

A Minimally you need to override the viewDrawScript of the protoPictRadioButton. You may also want to change the viewFormat since it defaults to a thick rounded-rectangle border. Assuming that you wanted some sort of rectangle highlight around the selected button, you could use the following viewDrawScript:

```
func()
begin
   // If the button is selected, highlight it.
   if viewValue then
   begin
      // Get the bounds of the protoPictRadioButton.
      local b := :LocalBox();

      // Inset the bounds.
      b.top := b.top + 2;
      b.left := b.left + 2;
      b.bottom := b.bottom - 2;
      b.right := b.right - 2;

      // Now draw a rectangle.
      :DrawShape(MakeRect(b.left, b.top, b.right, b.bottom), nil);
   end;
end
```

Q *What is the AutoClose checkbox in the Newton Toolkit for? Why should I use it?*

A The AutoClose flag causes all other AutoClose applications to be closed when your application is clicked. The effect is that only one "auto-close" application can be open at one time. You should *always* make your application auto-close, to help conserve memory and other resources, unless it's providing special functionality to other applications (like the built-in Calculator or Styles application).

Q *How do I create my own class of binary object?*

A To get a binary object of your own class, you first need to create a binary object, then change its class to your own. The easiest way to do this is to create a string that's the same length as your intended binary object and then change (coerce) the class of this new object to your own class. You can use the string class as your basic binary object.

Suppose you wanted to have a binary class called CharID for an ID consisting of four ASCII characters. You could write a NewID function in your ProjectData file that would create the object and optionally initialize it, like this:

```
// Define a constant for a default CharID object. This constant
// can be cloned at run time. kDefaultCharIDObj will be a CharID
// object with 4 bytes that are set to 0x00.
DefConst('kDefaultCharIDObj, SetClass(SetLength("", 4), 'CharID));

// CharString is a string of 4 characters or nil.
NewCharID := func(CharString)
begin
   // Create a binary object of the correct length.
   local newObj := Clone(kDefaultCharIDObj);

   // Optionally initialize it.
   if CharString then
      for i := 0 to 3 do
         StuffChar(newObj, i, CharString[i]);

   // Return the new object.
   newObj;
end;
```

To see this code in action, you can type it into the Inspector window in the Newton Toolkit and evaluate it. Note that you cannot use DefConst in the Inspector since it's a compile-time function. Just substitute the second argument in the DefConst function for kDefaultCharIDObj in the function to evaluate it. Then you can try things like this:

```
x := :NewCharID(nil);
#440DD01  <CharID, length 4>
ExtractChar(x, 2);
#6        $\00
x := NewCharID("abcd");
#4410FC9  <CharID, length 4>
ExtractChar(x, 2);
#636      $c
ExtractByte(x, 2);
#18C     99
```

Q *What is your favorite color?*

A Llama fur beige.

Q *In my application I have a clPictureView that can display a variable number of pictures. Right now I create a bunch of picture slots in my application and then make another slot at run time that is an array of those items. There must be an easier way.*

A You're right; there is an easier way. You can use the GetPictAsBits function in your ProjectData file to read in the bitmaps. Note that you'll first have to open the resource file that contains the pictures.

```
// Open the resource file that contains the pictures.
// Assumes the file "Pictures" exists in the project folder.
r := OpenResFileX("Pictures");

// Get an array of pictures.
myPictures := [
   GetPictAsBits("TARDIS", nil),
   GetPictAsBits("Planet", nil)
];

// Now close the resource file.
CloseResFileX(r);
```

Once you have the myPictures array, you can create a slot of type Evaluate and just type "myPictures" in the editor for the slot. Then you can use SetValue to set the icon slot of the clPicture view to one of the elements of the array.

Q *How do I put my own default person in the fax information slip?*

A You can set up a default person in your SetupRoutingSlip method, which is called before the fax slip is shown. The argument to that method is used to set up the particular routing slip. In the case of a fax slip, there's a slot called "alternatives" which is an array of cardfile entries for the possible people to fax to. If there's just one entry, that's the person. The fax number will be set from the cardfile entry. So your SetupRoutingSlip method would look like this:

```
func(fields)
begin
   // Check for a fax.
   if fields.category = 'faxSlip then
      fields.alternatives := SmartCFQuery("Llama");
   // Do other stuff here, like put a title for the out box.
end
```

Note that the fields.category is set to the same value you set in the routeSlip slot of a frame in your application's entry in the global routing frame. The SmartCFQuery function returns an array of cardfile entries that have strings starting with the string passed in.

Q *I noticed that every soup entry has a _uniqueID slot. Just how unique is it?*

A The _uniqueID slot is only unique within that soup on that particular store. The ID will never be reused in that soup on that store. However, it's not necessarily unique across stores (say, RAM and a PCMCIA card).

Q *What is the ground velocity of an unladen llama climbing a 5% grade?*

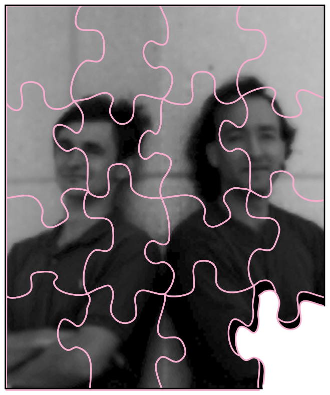
A What do you mean — Mexican or Venezuelan?

Thanks to Glen Raphael and our PIE Partners for the questions used in this column, and to jXopher, Todd Courtois, Bob Ebert, Mike Engber, Kent Sandvik, Maurice Sharp, and Scott ("Zz") Zimmerman for the answers.•

Have more questions? Need more answers? Take a look at PIE Developer Info on AppleLink.•

Monitor Madness

See if you can solve this programming puzzle, presented in the form of a dialog between Konstantin Othmer (KON) and Bruce Leak (BAL). The dialog gives clues to help you. Keep guessing until you're done; your score is the number to the left of the clue that gave you the correct answer. Even if you never run into the particular problems being solved here, you'll learn some valuable debugging techniques that will help you solve your own programming conundrums. And you'll also learn interesting Macintosh trivia.



KONSTANTIN OTHMER AND BRUCE LEAK

KON With all the talk of PowerPC, I should give you one of the really nasty PowerPC bugs I've been working on. But to be fair to those who aren't up to speed yet, namely one BAL, I'll give you a bug that's right up your alley.

BAL You're a saint.

KON Have you heard of the Display Enabler software that went out with the new 17-inch Multiple Scan display and is part of System 7.5? It lets you dynamically change your monitor configurations, such as your menu bar screen and resolution on Multiple Scan displays.

BAL You mean I can Option-drag the menu bar across screens in the Monitors control panel and it changes dynamically? Totally cool! I hate that "takes effect on restart" stuff. This means that I can help out all those multimedia applications that don't understand my second monitor is color.

KON And if you get one of the Multiple Scan displays, you can double-click on a screen in Monitors and change its resolution on the fly. Last time I saw someone try that on a Windows machine, it crashed!

BAL Give 'em ten more years, I'm sure they'll get it right.

KON With Display Enabler, the system automatically keeps windows on the screen when you move the menu bar. It even moves your icons to reasonable positions. If you're using the new scriptable Finder, the icon relocation is awesome. Sometimes I get the feeling it can read my mind.

KONSTANTIN OTHMER AND BRUCE LEAK absconded to the ski slopes immediately after writing this Puzzle Page, leaving *develop* without a bio. Fortunately, they had AppleLink hooked up to their cellular phone–based Newtons, so they could pen their bio remotely. Although *develop*'s editors repeatedly sent them mail requesting a bio, all they ever got back was a cryptic message about Crinoline Gopher and Brunch Creek. Go figure.•

BAL Wait. Are you running a Macintosh or a Newton?

KON Ha ha.

BAL OK, back to the puzzle. What's the bug? Was there trouble with applications that hide windows by moving them off the screen? Those windows wouldn't be able to be located after being moved along with the menu bar.

KON We handle that case for the most part, although that's a pretty cheesy way of hiding windows. Apparently some developers have trouble figuring out what ShowHide does. The bug I have for you is way better than that.

BAL Yeah, sure. It probably only happens on a particular machine, in one magic case that's really hard to reproduce, and only when MacsBug isn't loaded.

KON No fair. You've played this game before. Actually, someone reported that when you change the resolution of a monitor and then try to restart, the machine hangs on boot.

BAL Well, it's a good thing monitor reconfigurations take effect without restarting, since restarting doesn't work anymore! Does it happen if I don't have Display Enabler installed?

KON Without Display Enabler, you can't change the resolution of the monitor. Solve the problem.

BAL Only under protest. How far does he get when he tries to reboot?

KON Smiley Mac and that's it. It dies right about where you would expect the death chimes.

BAL And it only happens on this one machine?

KON When we tried to reproduce it on an identical configuration we couldn't.

BAL What's his configuration?

KON Macintosh Quadra 900, on-board video connected to a 17-inch Multiple Scan display, two hard drives, and System 7.1 with Display Enabler.

BAL How does he regain access to his machine?

KON He resets parameter RAM (PRAM) by holding down Command-Option-P-R until the machine chimes the second time. Then everything boots fine.

BAL But it boots into the default video mode, not the one he specified in Monitors.

100 KON Of course. Because the original Macintosh was floppy-based, PRAM was added to hold a lot of system configuration information, such as mouse speed, double-click time, and sound volume for each monitor. That way you could boot off of different floppies and still maintain all your system preferences.

BAL The Macintosh II extended the use of PRAM to add support for multiple video cards and slot devices. Since PRAM is in such short supply, there's also a 'scrn' resource that's maintained by Monitors that contains the relative locations of the different monitors, as well as each monitor's mode. The problem is that the 'scrn' resource and your

PRAM are out of sync, so the system gets confused and hangs. Go fix your bug. Over.

90 KON Nice theory, but wrong. First of all, we restart right after closing Monitors. The last thing Monitors does is update the 'scrn' resource and tell the driver to update its PRAM settings. So there's not much chance for them to get out of sync. Second, the 'scrn' resource is read and acted upon at extension loading time, and we're hanging way before that.

BAL So something that's getting written to PRAM must be causing all the fuss. On his machine, use the DPRAM dcmd to check the PRAM locations related to video before and after changing the mode. To factor Monitors out of the equation, boot and set the PRAM locations by hand and see if that's the only dependency.

KON Slick sleuthing, BAL! It turns out that PRAM location that holds the new mode for the on-board video (in your case, $49) gets changed. If you change it using the SPRAM dcmd rather than Monitors, the problem still occurs.

BAL Drat. I was hoping that Monitors was somehow trashing the startup drive PRAM location. I've always been looking for a good excuse to rewrite Monitors out of that Pascal morass. This certainly narrows it down.

KON If you say so. But you can only get it to happen on this one machine, and how could video PRAM be related to booting, anyway?

BAL It's going to be hard to get MacsBug involved since we're too early in the boot process.

KON You could get MacsBug-like capabilities at boot time using BootBug, but there's no card around.

BAL So what's special about this configuration? Is it a prototype 17-inch Multiple Scan display?

KON Nope. He claims to be using a production unit.

BAL What if you remove the extra hard drive?

85 KON It boots fine. No problem.

BAL You mean if I just turn off the external drive, the machine boots fine?

KON Yep.

BAL What if I set the system to boot off of the external drive?

KON There's no system on the external drive. But you can tell the system to set the startup disk there. It's not as if it checks for a valid boot volume or anything. I've heard you can even set it to boot off of a Sega CD.

BAL So I put a system on there and set it as the boot drive. What happens?

80 KON It boots fine.

BAL But if I set the boot drive back to the internal drive, I crash?

KON Nope. That seems to be working fine now too.

BAL Aha! I'll remove the system from the external drive and try to reboot. Now I crash, right?

75 KON Nope. The system seems to be booting fine now. Nice going, BAL! You fixed the only reproducible case! Could you recap the symptom

and the fix so that I can just add them to the Read Me file and be done with it?

BAL Yeah, yeah. Somehow something must have changed on that external disk. Maybe the Finder wrote out some new boot blocks or removed some stale boot blocks when I trashed the System file. Can we put the disk back to its original state?

70 KON We'll break into our hermetically sealed digital fiberoptic wireless remote personal information highway archive server and restore your disk image. The machine still boots fine.

BAL If it's not the disk, I must have changed something in PRAM. What's the startup drive set to now and what was it when I dumped PRAM earlier?

65 KON The long word that holds the boot drive (PRAM location $78–$7B) used to hold $0 and now it holds $FFFFFFDF (a driver refNum), which indicates that you're booting off the internal drive.

BAL What does $0 specify?

KON That's what PRAM gets set to when you don't select a boot drive in the Startup Disk control panel. It tells the system to go look for a valid boot drive and boot off of the first one it finds.

BAL When I set the boot drive to the internal drive using Startup Disk, the PRAM location was set. What happens if I set it back to $0?

60 KON You hang on boot, same as before.

BAL What if you put a System Folder on the external drive but leave the PRAM boot drive set to $0?

55 KON You don't hang anymore, and the system boots off of the external drive.

BAL And if I drag that System Folder to the trash?

50 KON The system hangs on boot.

BAL What if I use a newly initialized external drive with the same SCSI ID?

45 KON It boots fine.

BAL Let's recap for those just joining us. The machine hangs on boot on a two-drive system under these circumstances: the System Folder on the external drive has been deleted; there's no default boot drive selected; and the video mode of the on-board driver is set to something other than the default. Did I forget anything?

KON The solution.

BAL So how does the system go searching for boot drives?

40 KON If the device specified in PRAM doesn't exist or isn't bootable, the boot code starts with the device with the highest SCSI ID and looks for boot blocks. If they exist, it tries to boot off of that drive. If it works, great. If there's no System Folder or Finder, it will start over with the next highest SCSI ID.

BAL I bet if I put the same SCSI driver on both disks using Apple HD SC Setup, the problem goes away.

KON You got a theory here, or what?

BAL Early in the boot process, the system heap is really small. When you put the video driver into one of the new modes tickled by Display Enabler, it loads patches out of ROM into the system heap. These patches fight with the SCSI driver for the limited system heap RAM. When no boot drive is selected, the boot code trolls the SCSI bus looking for a valid boot device. It loads the SCSI driver off the first candidate, starts booting, and finds there's no valid System Folder present. So it goes to the next drive, sees that there's a different SCSI driver version on it, and tries to load that version. Because the video driver loaded its extended tables and forgot to grow the system heap, the load fails and the system hangs without a clue as to how to proceed.

35 KON Wow! Fabulous theory, but wrong. If you get BootBug — a wonderful product for anyone wanting to debug system startup, by the way — and watch the SCSI Manager allocate space for the driver, it succeeds. Furthermore, when the system fails to boot from the first drive it finds, it throws everything away by calling InitZone on the system heap zone and starts over with the next drive. Your story about the video driver patches is accurate, but there's still enough room in the system heap after they load. Your move.

BAL OK, I need some tools. What have you got? Do you expect me to debug this with my bare hands?

KON Well, we don't have an emulator handy, but we could probably call in a few favors and get a BootBug NuBus card.

BAL Now this should be easy! Where do I crash?

30 KON You crash in code that's monkeying around with the low-memory global at $DD8, UniversalInfoPtr. It's the table that tells you everything you ever wanted to know about this machine's configuration: the clock speed, all kinds of I/O stuff, the kind of sound hardware, SCSI hardware, on-board video, and memory controller, the number of NuBus slots . . .

BAL OK, OK. What's the problem?

KON Well, it dereferences $DD8, makes some calculations with the offsets, and ends up with a bogus address and a bus error.

BAL Where does $DD8 point?

25 KON Into RAM.

BAL That's strange. All that configuration information should be in ROM. I'll stop BootBug immediately and step spy on $DD8 to see who changes it.

20 KON By the time BootBug comes in, the location is already changed.

BAL Wait, BootBug loads first. It should come in before any other slots get called.

15 KON It works that way on the Macintosh Quadra 610, 650, and 800 models and later (including the AV models). But you're on a Quadra 900, and on-board video occupies the first NuBus slot. The video is already gray, so the primary INIT of the on-board video has already been called.

BAL If I don't set the new video mode, so that I boot successfully, where does $DD8 point?

KON Into ROM.

BAL Aha! The video driver is doing something different because it sees it needs one of the extended modes. It must patch $DD8 to change the configuration information for the video display.

10 KON So? What's wrong with patching? Why does it work when only one drive is involved?

BAL I got it! The video driver patches out $DD8 to replace the tables for the extended video modes. Then the boot code starts looking for a valid boot drive. It finds the external drive, which has valid boot blocks since it once had a system on it, and tries to boot off of that. When it realizes there's no valid System Folder on the disk, the machine performs a warm restart and tries the next drive.

5 KON Nothing new here yet.

BAL The problem is that when the system restarts, it reinitializes the system heap, throwing out the video patches, but doesn't reinitialize $DD8 since that's set very early by the boot code to describe the type of machine that was detected. Now $DD8 points to garbage. As soon as someone tries to reference $DD8 they get garbage, resulting in a bus error. The machine doesn't know what to do and locks up.

KON Why don't you get a system error or at least the death chimes?

BAL Since there's no way to draw to the screen until a video driver is successfully found and opened, the death chimes were designed to audibly indicate where in the boot process failure occurred. Once the video driver is successfully opened, the death chimes error handler is replaced with the standard system error handler. But when the external drive failed to boot, the video driver was thrown out and the error occurred before it was reopened, so no error message could be displayed.

KON Precisely. Since all the problems happen in ROM long before we can get control, unless we want to do one of those nasty Darin-changed-the-boot-blocks patches, we can't write those extended modes to PRAM. So we wait until the Display Enabler INIT loads to synchronize the display with the 'scrn' resource. Trying to debug things at boot time is hard enough, especially when they happen before BootBug loads.

BAL Nasty.

KON Yeah.

SCORING

75–100	If you're so smart, you write the next one.
50–70	You're qualified to rewrite the boot code. Please use C this time.
25–45	You're qualified to rewrite the boot code in Pascal.
5–20	Study the pipeline instruction architecture used by the 68000 versus the emulator to get a head start for next time.•

Thanks to Ian Hendry, Carl Hewitt, and Mike Puckett for reviewing this column.•

History of the Dogcow, Part 2

MARK ("THE RED") HARLAN

In Issue 17, we told part 1 of the history of the dogcow. We'll warn you again: If you don't know what or who the dogcow is, or you don't care for Apple cultural minutiae, you should just flip past this column.

DISTRIBUTION OF TECH NOTE #31

We left off at the point where the former Macintosh Technical Note #31, "The Dogcow," had been created. The question then was how to distribute it. Mark Johnson and I both thought that since it was an April Fool's joke anyway, the best thing would be to just include it in the April monthly mailing to Apple Partners and Associates; we'd drop it from the subsequent batches, with the direct intent of making it a curio. The idea was that the people who were currently in the Macintosh community would get it and everyone else wouldn't. We very intentionally were trying to build an aura around it. The April 1989 mailing is the only time this Tech Note was ever in print under the official auspices of Apple.

There was a bit of a lag time between the writing of the Note and the actual release; by the time it went out, I actually had forgotten about it. The response was immediate and intense. Internally I received a couple of vaguely threatening calls from people claiming false ownership, but the overwhelming majority of people thought it was great. One gentleman in the developer community took offense saying that "dogcow" was too close to "Dachau" and showed how the note had underpinnings of anti-Semitism. (I showed this one to my Jewish father-in-law, who had to be resuscitated, he was laughing so hard.)

Aside from that, it really struck a chord with the developer community like nothing I've seen before or since. I received about 40 pieces of fan mail that month. Developer Technical Support (DTS) must have gone for a year before there was a batch of e-mail that didn't have a dogcow reference in it. In fact, to this day people say to me, "Mark Harlan? I know your name from Tech Notes" — but it's the only one I ever wrote.

Then came the concept of a Developer CD as a vehicle for distributing Tech Notes electronically (along with sample code and more). I was overseeing that project, and immediately we had an interesting conundrum: We wanted all information in electronic format, yet what were we going to do with Tech Note #31? Merely slipping it into the Tech Notes stack seemed like disaster, but then it didn't really feel right to omit it.

Again, it was Mark Johnson who came to the rescue with the excellent idea of burying the Tech Note. So on the early CD, "Phil and Dave's Excellent CD," you have to go through a bizarre sequence of commands to bring it up. Even now, tradition requires that I not give the details, but it involves Shift-Option-clicking and typing "grazing off a cliff," and it emits "Moof!" and "Foom!" sounds. (For the "Moof!" sound we took a real cow and then Zz said "fff" into a MacRecorder; the "Foom!" is just the same sound played backwards.) It took a while for anyone to find the Note using any technique, and I've never heard of anyone doing it except through ResEdit.

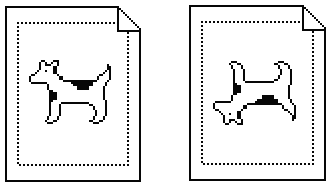
The Note stayed on the first few Developer CDs. The access technique changed from disc to disc, and not even I knew how to do it after the original "Phil and Dave." Somewhere along the line the Note was dropped from the CD altogether.

OTHER DOGCOW PARAPHERNALIA

Bootleg T-shirts started appearing. There was an apartment near Apple headquarters that started flying a dogcow flag. The stack version of the Note had a watermarked background that someone removed pixel

MARK ("THE RED") HARLAN went through extensive deprogramming after six years at Apple. Unfortunately, the therapy didn't hold and he has since joined yet another cult: General Magic. In a recent interview, Mark was asked if he had any words of wisdom on the dogcow. "Yeah. Warn everyone that both the dogcow logo and 'Moof!' are trademarks of Apple Computer. You don't *ever* want to be in the position of having to answer 'What are you in for?' with 'Bootleg T-shirts.'"•

Our friend in the LaserWriter Page Setup Options dialog, normal and flipped vertically:



by pixel before posting it to the Internet. Several developers were nearly thrown out of a movie theater at MacHack for "Moofing" before a movie.

In addition to the Tech Note there are three pins: green background, the most common; red background with Kanji (the word on the pin actually is pronounced "Moo-aann!" because Japanese dogs don't woof, they say something like "aann-aann"); and the super-rare red background with "Moof!", which are misprints of the Kanji batch. Also, there's a dogcow window sticker. All of these were given away in DTS labs, and all but the window sticker have been collected up a long time ago.

If you think of the dogcow fathers as being Zz Zimmerman, Mark Johnson, and me, there's only one dogcow shirt that received our supervision and approval: the black DTS sweatshirt with the small dogcow on the chest (designed by Toni Trujillo). I also designed the graphic for a DTS gift that was a shoulder bag with all incarnations of the dogcow on it (flipped, rotated, and inverted). Unfortunately the bag was incredibly cheap and most of them have self-destructed.

Chris Derossi and Mary Burke designed a dogcow mousepad and even went so far as to call Pepsi-Cola to get the exact color of Mountain Dew green for the background. They made 500 of these and I wrote an insert that went into the packaging. Aside from the original Tech Note, it's the only thing I've ever written about dogcattle — until these *develop* columns.

DOGCOW TRIVIA

Somewhere along the line I baptized the dogcow "Clarus." Of course she's a female, as are all cows; males would be referred to as dogbulls, but none exist because there are already bulldogs, and God doesn't like to have naming problems.

Now things are much bigger than they were then — both in number of developers and number of Apple employees. The dogcow regularly appears on documents that are no longer connected to DTS, or in some cases (such as Scott Knaster's books) not even from Apple. In a sense, the dogcow has become mainstream; people are copying it — and that's exactly what I was fighting against in the first place (not to mention that she, and her "Moof!" cry, are bona fide trademarks of Apple Computer). To put a stop to all this, I'm threatening to kill her off, but *develop*'s editor has become such a fan that she's not sure she'll accept a "Dogcow is Dead" column. Stay tuned!

How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to AppleLink DEVELOP or to:

Caroline Rose
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-4DP
Cupertino, CA 95014
AppleLink: CROSE
Internet: crose@applelink.apple.com
Fax: (408)253-8521

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-4DP
Cupertino, CA 95014
AppleLink: JOHNSON.DK
Internet: dkj@apple.com
CompuServe: 75300,715
Fax: (408)253-8521

Please direct all subscription-related queries to *develop*, P.O. Box 531, Mount Morris, IL 61054-7858 or AppleLink DEV.SUBS (or, on the Internet, dev.subs@applelink.apple.com). Or call 1-800-877-5548 in the U.S., (815)734-1116 outside the U.S., or (815)734-1127 for fax.

INDEX

For a cumulative index to all issues of *develop*, see this issue's CD.•