

develop



Issue 27 September 1996

The Speech Recognition Manager

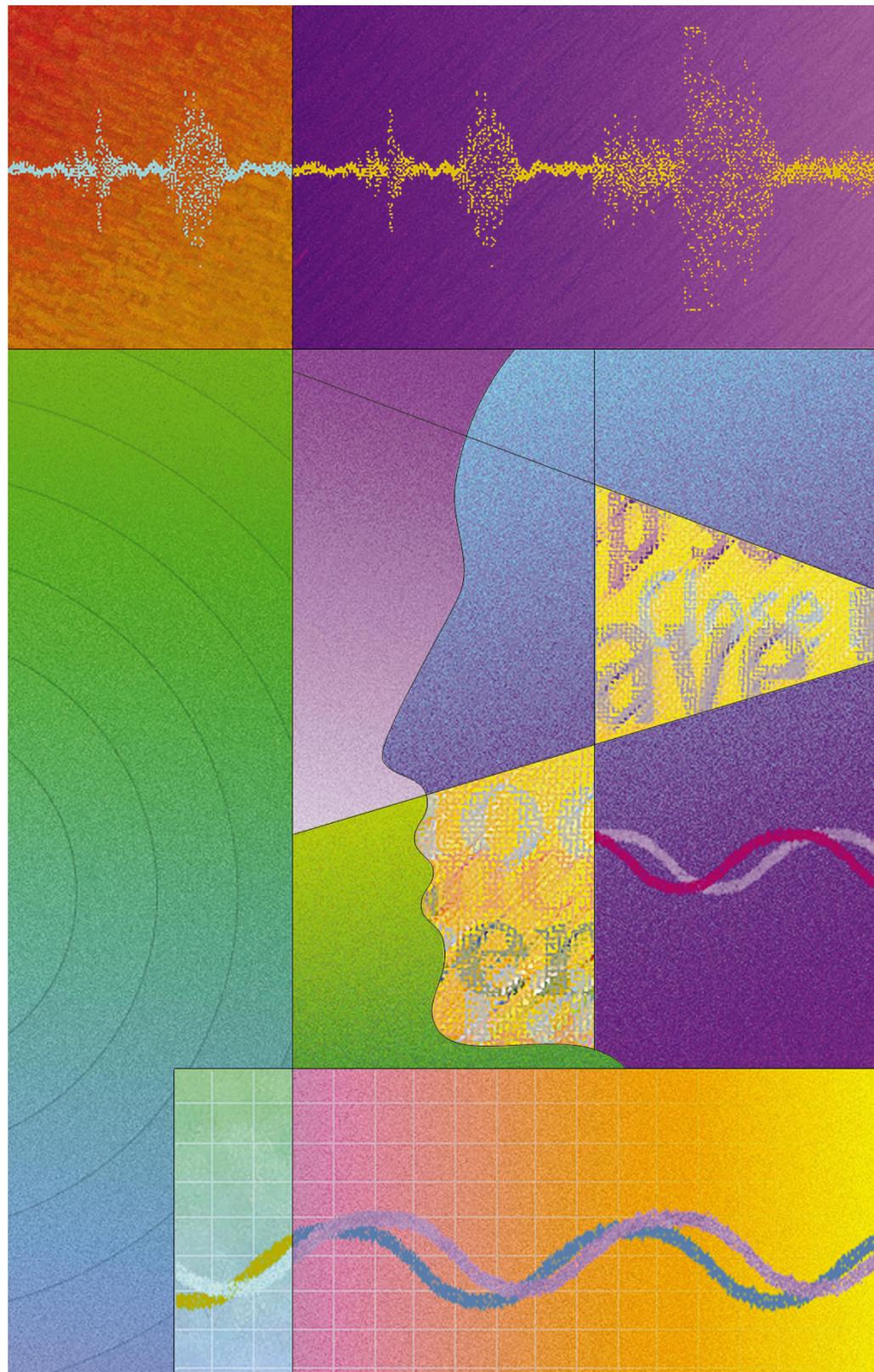
Adding Speech Recognition to an Application Framework

Working With OpenDoc Part Kinds

Using Apple Guide 2.1 With OpenDoc

Mac OS 8 Assistants in System 7 Applications

Game Controls for QuickDraw 3D





EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*
Editor-in-Cheek's Clothing (during
Caroline's sabbatical) *Lorraine Anderson*
Managing Editor *Toni Moccia*
Technical Buckstopper *Dave Johnson*
Bookmark CD Leader *Alex Dosber*
Able Assistants *Meredith Best, Beth Runciman*
Bosses *Mark Bloomquist, Garry Hornbuckle,*
Steve Strong
Review Board *Brian Bechtel, Dave Radcliffe,*
Quinn "The Eskimo!", Jim Reekes,
Bryan K. "Beaker" Ressler, Larry Rosenstein,
Nick Thompson, Gregg Williams
Contributing Editors *Lorraine Anderson,*
Linda Fogel, Toni Haskell, Tim Monroe,
Cheryl Potter, Erik Sea, George Truett
Indexer *Marc Savage*

ART & PRODUCTION

Art Direction *Lisa Ferdinandsen*
Technical Illustration *John Ryan*
Formatting *Forbes Mill Press*
Production *Diane Wilcox*
Photography *Sharon Beals, Lisa Ferdinandsen*
Cover Illustration *Grabam Metcalfe of*
Metcalfe/Shubert Design

ISSN #1047-0735. © 1996 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, AppleScript, AppleTalk, Bento, ColorSync, HyperCard, LaserWriter, MacApp, Macintosh, MacTCP, MessagePad, MPW, Newton, OpenDoc, PlainTalk, PowerBook, Power Macintosh, PowerTalk, QuickTime, and TrueType are trademarks of Apple Computer, Inc., registered in the U.S. and other countries. AOCE, A/ROSE, Balloon Help, Cyberdog, develop, Dylan, Finder, Mac, MacinTalk, NewtonScript, and QuickDraw are trademarks of Apple Computer, Inc. PostScript is a trademark of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions. PowerPC, SOM, and SOMObjects are trademarks of International Business Machines Corporation, used under license therefrom. All other trademarks are the property of their respective owners.

THINGS TO KNOW

develop, The Apple Technical Journal, a quarterly publication of Apple Computer's Developer Relations group, is published in March, June, September, and December. It provides developers of Apple-platform products with technical articles and code that have been reviewed for robustness by Apple engineers.

Electronic develop. This issue and all back issues of *develop*, along with the code they describe, can be found on the *develop Bookmark CD* or the Reference Library edition of the *Developer CD Series* (see below), or at <http://dev.info.apple.com/develop/> or ftp://ftpdev.info.apple.com/Developer_Services/Periodicals/develop/. The code is updated regularly, so always use the latest version.

This issue's CD. Subscription issues of *develop* are accompanied by the *develop Bookmark CD*. This CD contains a subset of the materials on the *Developer CD Series*, which is part of the Apple Developer Mailing available through the *Apple Developer Catalog*. The CD also contains Technotes, sample code, and other documentation and tools (these contents are subject to change). Items referred to as being on "this issue's CD" are located on either the *Bookmark CD* or the Reference Library or Tool Chest edition of the *Developer CD Series*. The *Bookmark CD* contents can also be accessed from <http://dev.info.apple.com/> or from ftp://ftpdev.info.apple.com/Developer_Services/.

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

CONTACTING US

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

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

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

Apple Developer Catalog. To order *develop* or other products through the catalog or to make subscription-related queries, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can send e-mail to order.adc@applelink.apple.com, use the catalog on the Web at <http://www.devcatalog.apple.com>, or write *Apple Developer Catalog*, Apple Computer, Inc., P.O. Box 319, Buffalo, NY 14207-0319.

For all subscription changes or queries, *please be sure to include your name, address, and account number as they appear on your mailing label.*



Printed on recycled paper by Stream International, USA

ARTICLES

- 6 The Speech Recognition Manager Revealed** by Matt Pallakoff and Arlo Reeves
- 22 Adding Speech Recognition to an Application Framework** by Tim Monroe
With these two articles, you'll have your application recognizing speech in no time. The first is an introduction to the long-awaited API for speech recognition, and the second is an example of adding basic speech recognition capabilities to a PowerPlant application. Listen up!
- 37 Working With OpenDoc Part Kinds** by Tantek Çelik and Dave Curbow
Part kinds are like file types, only more so, and the choices you make about which part kinds to support will have a profound effect on users' experiences with your part editor.
- 53 Using Apple Guide 2.1 With OpenDoc** by Peter Commons
Apple Guide 2.1 has been extended to work well in OpenDoc's brave new world of compound documents and processes within processes. Here's a look at the new features and how to take advantage of them.
- 72 Mac OS 8 Assistants in System 7 Applications** by José Arcellana and Arno Gourdol
Assistants will provide interview-based help in Mac OS 8, guiding users through complex tasks. This article gives some tips on designing an assistant and shows how you can implement one now, under System 7.
- 87 Game Controls for QuickDraw 3D** by Philip McBride
First-person 3D applications, whether games or 3D modeling systems, need to constantly move the camera to reflect the changing point of view of the player. You too can inflict vertigo on your users.

COLUMNS

- 34 PRINT HINTS**
The All-New LaserWriter Driver Version 8.4
by Dave Polaschek
The new version of the LaserWriter driver is way different. Will your application break?
- 69 THE OPENDOC ROAD**
Facilitating Part Editor Unloading
by Vincent Lo
Part editors are unloaded automatically when they're not needed, but your editor can help.
- 82 BALANCE OF POWER**
Stalking the Wild Defect
by Dave Evans
A tour through the dangerously twisted jungle of the Power Macintosh. Please keep your head and arms inside at all times.
- 98 GRAPHICAL TRUFFLES**
A Library for Traversing Paths
by Daniel I. Lipton
Parsing QuickDraw GX geometric shapes takes a bit of code, but it's already been written for you.
- 102 MACINTOSH Q & A**
Answers from Apple's Developer Support Center — lots on Open Transport this time.
- 111 THE VETERAN NEOPHYTE**
Your Friend the Drill Sergeant
by Dave Johnson
Learning to shoot pool isn't anything like learning to program computers. Right?
- 113 NEWTON Q & A: ASK THE LLAMA**
Answers to Newton-related development queries.
- 118 KON & BAL'S PUZZLE PAGE**
QuickTime Quandary
by Konstantin Othmer and Bruce Leak
More Macintosh madness from the MacsBug mentors. There's a possibility you might actually score on this one!

-
- 2 EDITOR'S NOTE**
3 LETTERS
123 INDEX

EDITOR'S NOTE



**DAVE JOHNSON
FOR CAROLINE ROSE**

As I write this our esteemed Editor-in-Cheek is off on sabbatical, indulging in a little global gallivanting and some well deserved (and completely unstructured) hanging around. Thus it falls to me to write this editorial, making this only the second issue of *develop* that has had *two* pictures of me in it (trivia question: which was the other?), and also marking the first time (and hopefully the last time) my signature has been aired in public. (Yes, I know it's illegible, and I confess: I never really learned to write cursive. So it's only a rough approximation. Even my printing is barely legible. Thank goodness for keyboards.)

Unaccustomed as I am to editorial speaking, I was having a hard time thinking of something to write about. Fortuitously, Apple's Worldwide Developers Conference occurred just about the time I needed to settle on a topic, and as always the developers I talked with at the conference brought up several issues about what we on the *develop* staff do and why we do it that way.

One issue that came up is conveniently editorial in nature. We're often applauded for the better-than-usual (at least for a technical journal) writing in our articles and columns. It's quite true that in addition to trying to make sure everything in the magazine is correct, we also put a *lot* of effort into making it read well. This is great for you, the reader, but as with any way of doing things there's a downside. In this case, it means more trouble and more work for those who generate the content of the journal. Occasionally an author thinks it's a hassle, all that fussing over finding the right word or phrase, all that questioning and worrying over something that's "off the topic" as far as they're concerned. For others, of course, it's a real blessing, having our highly trained team of crackerjack editors swarming over their work, nipping and tucking and polishing it until it's snug and smooth and gleaming. While I naturally tend to side with the latter, everybody's entitled to an opinion.

Those of us here at *develop* believe that it's absolutely worth it. It's a truism about technical writing that if it's done well no one notices it. That's our goal, and always has been, and I think it's a good and important one. If you have to read a sentence twice or three times to figure out what it means, or if you have to backtrack a page to make sense of something you just read, or if you can't find a constant in *Inside Macintosh* because it's spelled wrong in the article, then the writing *will* be noticed, because it's getting in your way. That's something we're proud to avoid more often than not, even though it takes longer, and even though it's a lot more work. We hope you agree.

So that's my editorial. An easy out, some might say, simply restating our editorial philosophy rather than coming up with new thoughts. But it's something that's often lost in all the noise, and I think it's good — both for us and for you — to be reminded once in a while why it is we do what we do.

A handwritten signature in black ink, appearing to read 'Dave Johnson'.

**Dave Johnson
Editorial Pretender**

DAVE JOHNSON (dkj@apple.com) and his wife Lisa have a tiny but thriving mask-making business in San Francisco, selling masks for a one-month period each year around Halloween. In 1994 they had sales of \$344 and gross profits were \$159. (There was a write-down of \$188

that year for retooling, resulting in a net loss of \$29.) In 1995 they had total profits of \$287 on sales of \$330. If this explosive profit growth keeps up, this small garage business could, in time, be worth literally hundreds of millions of dollars. Dave is rubbing his hands in anticipation. •

LETTERS

NURB CURVES TYPO

I'm an M.S. student in the Department of Industrial Engineering at Bogazici University, Istanbul, Turkey. My thesis is about pattern recognition using implicit polynomials in CAD applications. While I was surfing the Internet, I found the article "NURB Curves: A Guide for the Uninitiated" in *develop* Issue 25 and read it. It's a very good resource for those (like me) with minimal knowledge of NURB curves and representations, and I liked it a lot.

But in Figure 20 there's a mistake, I think. Control point B_2 has coordinates $\{2, 0, 0\}$, but I believe the last index (w) should be 1. Is that right?

— Ugur Murat Erdem

Yes, in fact you've found a typographical error in the figure you mention: B_2 should be $\{2, 0, 1\}$. We had a very good editor and several reviewers, but none of them caught this. I hope it doesn't mislead anyone too much. Thanks for pointing it out.

— Philip Schneider

TECHNICAL Q&A: WHERE?

In the Issue 25 Macintosh Q&A, you explain a new method for embedding GX pictures into QuickDraw PICT files. It says that we can find sample code in the Macintosh Technical Q&A "Embedding a GX Picture into a PICT" (GX 07). Unfortunately, I haven't found this file on any developer CD I have. Could you please help me locate this information?

— Michel Renon

That Q&A can be found on the Web at <http://dev.info.apple.com/techqa/qdgx/>

gx07.html. In general, the Web site <http://dev.info.apple.com/techqa/Main.html> has the latest and greatest Macintosh Technical Q&As. They sometimes take a while to find their way to the CDs, and that was the case here. That Q&A is now available on the CDs as well. Sorry for any inconvenience.

— Dave Johnson

TIME-SAVING TIPS

I really enjoyed Bo3b Johnson's Veteran Neophyte column in Issue 25 about ways to avoid wasting time. After programming the Mac for 10 years, I'm finally learning many of the things he talked about. It's funny looking back at all the mistakes I've made while thinking I was so smart.

I worked at Berkeley Systems on After Dark. One of the first things I did was write the Warp (or starfield) screen saver. I came up with a really cool assembly routine that, given an x and y , would draw the pixel on any monitor at any bit depth. It was a complicated routine (remember, I'm very smart) that used lots of shifts, multiplies, and divides. Even though I commented it, I still had to sit down and work through it each time I needed to make a change. Finally a coworker asked why I didn't just write a separate routine for each bit depth. I scoffed and said my routine was really cool. Needless to say, I rewrote it into separate routines; it was really easy, and is easy to maintain and change as well.

These days, instead of banging my head trying to come up with a "smart" way to do things, I just code in the most straightforward way I know how. I'm finding that I have better things to do than screw around with a triply linked list that looked good in *Dr. Dobbs* but

TELL US WHAT YOU THINK (PLEASE)

We welcome your letters to the editor, especially regarding articles published in *develop*. Write to Caroline Rose at crose@apple.com or, if technical *develop*-related questions, to Dave Johnson at dkj@apple.com. All letters should include your

name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did). Address subscription-related queries to order.adc@applelink.apple.com.

isn't really appropriate for my problem. I hate reliance on C-isms that aren't obvious: if you have to pull out the ANSI C book to figure it out, it isn't good code.

By the way, another good book is *The Psychology of Computer Programming* by Gerald M. Weinberg. It was written in 1971 but has some very interesting views on programming and programmers.

— Bruce Burkhalter

I'm not a windsurfer, but I *am* a Mac developer, so I read Bo3b Johnson's column in Issue 25 with great interest. My boss (Markus Fest, the programmer of Toast CD-ROM Pro) told me it was a Pflichtlektüre (something you just gotta read). He was right.

There's a book that should have been in your Recommended Reading section: *Code Complete* by Steve McConnell (Microsoft Press, 1993). It's worth checking out. Enjoy!

— Florian Dejako

It's amusing to look back and see how we learned the things we did, and how they've helped or hindered us. That introspection is actually what spawned the column: I realized that maybe others could avoid those mistakes if they read about them in advance.

To this day I run into arguments about using the full "power" of C/C++. I hate to see people writing code just to use some feature of C++ like operator overloading. If they can redirect that creative energy to figuring out a better algorithm, it's a total win.

I think restrictions can actually be liberating, by freeing you from having to think about everything. If the brace style in code were enforced, how many hours of wasted brain time would we get back? Having the meaningless stuff like brace style fixed in stone makes it easier to apply cleverness to the things that matter, like the quality of the software.

And Florian: thanks. I've never been called a Pflichtlektüre before, but I kinda like it.

— Bo3b Johnson



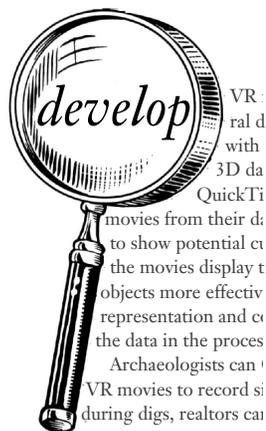
Learn how software components allow you to simplify the way you develop software.

Take Developer University's Creating OpenDoc Parts class and learn how to create parts using the OpenDoc Framework.

5 days, \$1500
September 16 - 20
October 28 - November 1
December 9 - 13

For more information, please visit Developer University's Web site at <http://dev.info.apple.com/du.html> or call our registrar at 408-974-4987. All courses offered in Cupertino, CA.

DEVELOPER
DU
UNIVERSITY



Looking to complete the set?

If you're looking for a complete *develop* collection, full-color, bound copies are available for \$13 per issue, including shipping and handling. (Back issues are also on the *develop Bookmark* CD and the *Developer CD Series* Reference Library edition, as well as on the Internet.) For more information about how to order printed back issues (and where to find them online), see the inside front cover of this issue.

Supplies are limited. Please allow 4 to 6 weeks for delivery.

Issue 1 Color; Palette Manager; Offscreen Worlds; PostScript; System 7; Debugging Declaration ROMs

Issue 2 C++ (Objects; Style Guide); Object Pascal; Memory Manager; MacApp; Object-Based Design

Issue 3 ISO 9660 and High Sierra; Accessing CD Audio Tracks; Comm Toolbox; 8•24 GC Card; PrGeneral

Issue 4 Device Driver in C++; Polymorphism in C++; A/ROSE; PostScript; Apple IIGS Printer Driver

Issue 5 (Volume 2, Issue 1) Asynchronous Background Networking; Palette Manager; Macintosh Common Lisp

Issue 6 Threads; CopyBits; MacTCP Cookbook

Issue 7 QuickTime 1.0; TrueType; Threads and Futures; C++ Objects in a World of Exceptions

Issue 8 Curves in QuickDraw; Date and Time Entry in MacApp; Debugging; Hybrid Applications for A/UX

Issue 9 Color on 1-Bit Devices; TextBox You've Always Wanted; Sound; Terminal Manager; Debugging Drivers

Issue 10 Apple Event Objects; Enhancements for the LaserWriter Font Utility; GWorlds; The Optimal Palette

Issue 11 Asynchronous Sound; Multibuffering Sounds; Exceptions; NetWork: Distributed Computing

Issue 12 Components; Time Bases; Apple Event Coding Through Objects; Globals in Standalone Code

Issue 13 Asynchronous Routines; QuickTime and Components; Debugging; Color Printing; DeviceLoop

Issue 14 Localizable Applications; 3-D Rotation; QuickTime (Video Digitizing; Making Better Movies)

Issue 15 QuickDraw GX; Component Registration; Floating Windows; Working in the Third Dimension

Issue 16 Making the Leap to PowerPC; PowerTalk; Drag and Drop From the Finder; Color Matching With QuickDraw GX; International Number Formatting

Issue 17 Newton Proto Templates; PowerPC (Standalone Code; Debugging); Thread Manager; Window Zooming

Issue 18 Apple Guide; Open Scripting Architecture; Graphics Speed on the Power Macintosh; Displaying Hierarchical Lists; Preferences Files

Issue 19 OpenDoc Part Handlers; PowerPC Memory Usage; Designing for the Power Macintosh; QuickDraw GX (Printing; Bitmaps); Inheritance in Scripts

Issue 20 AOCE; Make Your Own Sound Components; Scripting the Finder; NetWare on PowerPC

Issue 21 OpenDoc Graphics; Designing a Scripting Implementation; Dylan; Object-Oriented Hierarchical Lists

Issue 22 QuickDraw 3D; Copland; PCI Device Drivers; Custom Color Search Procedures; The OpenDoc User Experience; Futures

Issue 23 QuickTime Music Architecture; QuickDraw 3D Geometries; Internet Config; Multipane Dialogs; Document Synchronization; ColorSync 2.0

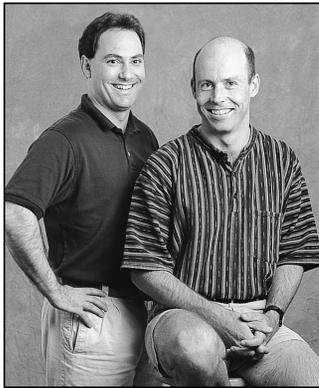
Issue 24 Speeding Up *whose* Clause Resolution; OpenDoc Storage; Sound; Alert Guidelines; Printing Faster With Data Compression; The New Device Drivers

Issue 25 QuickTime VR Movies From QuickDraw 3D; Flicker-Free Drawing With QuickDraw GX; NURB Curves; C++ Exceptions in C; Localized Strings for Newton

Issue 26 Mac OS 8 Compatibility; QuickTime Conferencing; OpenDoc Parts and SOM Dynamic Inheritance; Adding Custom Data to QuickDraw 3D Objects; 64-Bit Integer Math on 680x0 Machines

The Speech Recognition Manager Revealed

As any Star Trek fan knows, the computer of the future will talk and listen. Macintosh computers have already been talking for a decade, using speech synthesis technologies such as MacinTalk or the Speech Synthesis Manager. Now any Power Macintosh application can use Apple's new Speech Recognition Manager to recognize and respond to spoken commands as well. We'll show you how easy it is to add speech recognition to your application.



**MATT PALLAKOFF AND
ARLO REEVES**

Speech recognition technology has improved significantly in the last few years. It may still be a long while before you'll be able to carry on arbitrary conversations with your computer. But if you understand the capabilities and limitations of the new Speech Recognition Manager, you'll find it easy to create speech recognition applications that are fast, accurate, and robust.

With code samples from a simple speech recognition application, SRSample, this article shows you how to quickly get started using the Speech Recognition Manager. You'll also get some tips on how to make your application's use of speech recognition compelling, intuitive, and reliable. For everything you need in order to use the Speech Recognition Manager in your application (including SRSample and detailed documentation), see this issue's CD or Apple's speech technology Web site at <http://www.speech.apple.com>.

WHAT THE SPEECH RECOGNITION MANAGER CAN AND CANNOT DO

The Speech Recognition Manager consists of an API and a recognition engine. Under System 7.5, these are packaged together in version 1.5 or later of the Speech Recognition extension. (This packaging may change in future OS versions.)

The Speech Recognition Manager runs only on Power Macintosh computers with 16-bit sound input. Speech recognition is simply too computation-intensive to run

MATT PALLAKOFF (mattp@apple.com), Apple's Speech Recognition engineering manager, likes to talk to inanimate objects. He has spent the last several years helping Apple's speech group pull speech recognition technology kicking and screaming over a threshold of usability that (as of PlainTalk 1.4) finally allows Power Macintosh users to leave speech recognition on and use it in simple ways every day. He denies ever having worked in the field of Artificial Intelligence. •

ARLO REEVES (arlo@apple.com) has had a varied employment history that includes babysitting young Peregrine falcons in Yosemite, studying variable stars from Nantucket, and adding two-dimensional FFT capabilities to NIH Image. Lately he's been helping Matt and the speech team at Apple bring the Speech Recognition Manager into existence. Arlo lives in Santa Cruz, California, where he enjoys spending his free time out of doors with his friends. •

well on most 680x0 systems. The installed base of Power Macs is growing by about five million a year, however, so plenty of machines — including the latest PowerPC™ processor-based PowerBooks — can run speech recognition.

The current version of the Speech Recognition Manager has the following capabilities and limitations:

- It's speaker independent, meaning that users don't need to train it before they can use it.
- It recognizes continuous speech, so users can speak *naturally*, without — pausing — between — words.
- It's designed for North American adult speakers of English. It's not localized yet, and in general it won't work as well for children.
- It supports command-and-control recognition, not dictation. It works well when your application asks it to listen for at most a few dozen phrases at a time; however, it can't recognize arbitrary sentences and its accuracy decreases substantially if the number of utterances it's asked to listen for grows too large. For example, it won't accurately recognize one name out of a list of five thousand names.

OVERVIEW OF THE SPEECH RECOGNITION MANAGER API

To use the Speech Recognition Manager, you must first open a *recognition system*, which loads and initializes the recognition toolbox. You then allocate a *recognizer*, which listens to a *speech source* for sound input. A recognizer might also display a *feedback window* that shows the user when to speak and what the recognizer thinks was said.

To define the spoken utterances that the recognizer should listen for, you build a *language model* and pass it to the recognizer. A language model is a flexible network of words and phrases that defines a large number of possible utterances in a compact and efficient way. The Speech Recognition Manager lets your application rapidly change the *active* language model, so that at different times your application can listen for different things.

After the recognizer is told to start listening, it sends your application a *recognition result* whenever it hears the user speak an utterance contained in the current language model. A recognition result contains the part of the language model that was recognized and is typically sent to your application via Apple events. (Alternatively, you can request notification using callbacks if you cannot support Apple events.) Your application then processes the recognition result to examine what the user said and responds appropriately.

Figure 1 shows how the Speech Recognition Manager works. Note that the telephone speech source is not supported in version 1.5 of the Speech Recognition extension.

SPEECH OBJECTS

The recognition system, recognizer, speech source, language models, and recognition results are all objects belonging to classes derived from the `SRSpeechObject` class, in accordance with object-oriented design principles. These and other objects are arranged into the class hierarchy shown in Figure 2. The class hierarchy gives the Speech Recognition Manager API the flexibility of polymorphism. For example, you can call the routine `SRReleaseObject` to dispose of any `SRSpeechObject`.

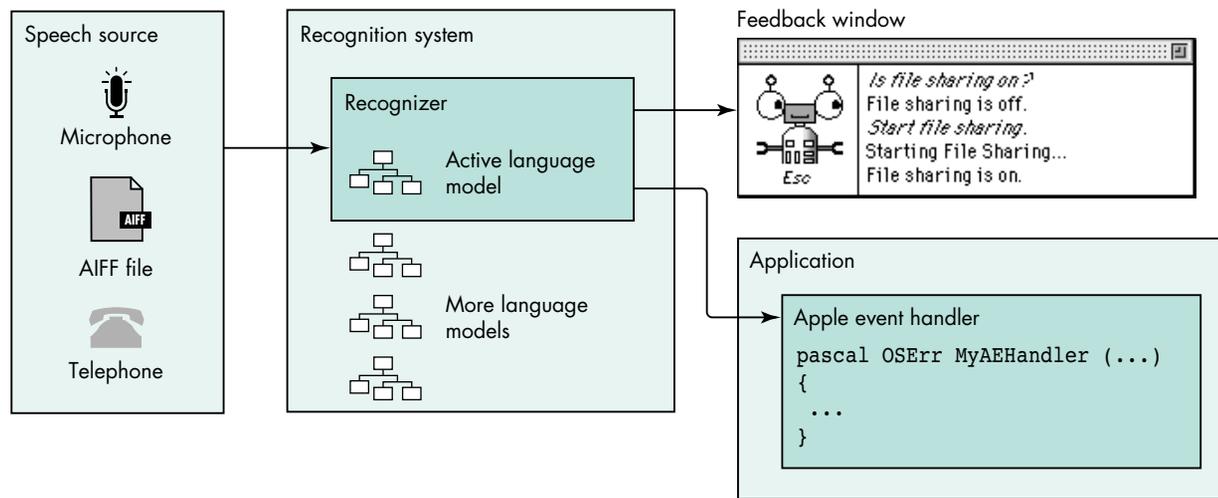


Figure 1. How the Speech Recognition Manager works

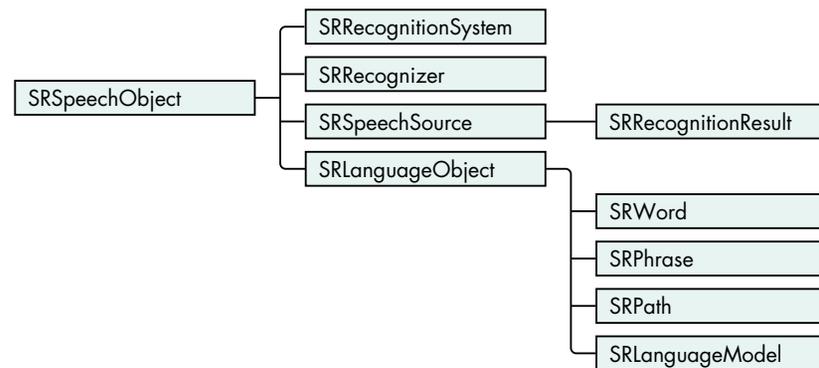


Figure 2. The speech object class hierarchy

The most important speech objects are as follows:

- **SRRecognitionSystem** — An application typically opens one of these at startup (by calling `SROpenRecognitionSystem`) and closes it at shutdown (by calling `SRCloseRecognitionSystem`). Applications allocate other kinds of objects by calling routines like `SRNewWord`, which typically take the `SRRecognitionSystem` object as their first argument.
- **SRRecognizer** — An application gets an `SRRecognizer` from an `SRRecognitionSystem` by calling `SRNewRecognizer`. The `SRRecognizer` does the work of recognizing utterances and sending recognition results back to the application. It begins doing this whenever the application calls `SRStartListening` and stops whenever the application calls `SRStopListening`.
- **SRLanguageModel, SRPath, SRPhrase, SRWord** — An application builds its language models from these object types, which are all subclasses of `SRLanguageObject`. (A *phrase* is a sequence of one or more words, and a *path* is a sequence of words, phrases, and language models.) A language model, in turn, describes what a user can say at any given moment. For example, if an application displayed ten animals and wanted to allow the user to say any of the animals' names, it might build a language model containing ten phrases, each corresponding to an animal's name.

-
- `SRRecognitionResult` — When an utterance is recognized, an `SRRecognitionResult` object is sent (using either an Apple event or a callback routine, whichever the application prefers) to the application that was listening for that utterance. The `SRRecognitionResult` object describes what was recognized. An application can then look at the result in several forms: as text, as `SRWords` and `SRPhrases`, or as an `SRLanguageModel`, which can assist in quickly interpreting the uttered phrase.

Each class of speech object has a number of properties that define how the objects behave. For example, all descendants of `SRLanguageObject` have a `kSRSpelling` property that shows how they're spelled. Your application uses the `SRSetProperty` and `SRGetProperty` routines to set and get the various properties of each these objects.

RELEASING OBJECT REFERENCES

You create objects by calling routines like `SRNewRecognizer` and `SRNewWord`. When you've finished using them, you dispose of them by calling `SRReleaseObject`. You can also acquire references to existing objects by calling routines like `SRGetIndexedItem` (for example, to get the second word in a phrase of several words).

The Speech Recognition Manager maintains a reference count for each object. An object's reference count is incremented by `SRNew...` and `SRGet...` calls, and is decremented by calls to `SRReleaseObject`. An object gets disposed of only when its reference count is decremented to 0. Thus, to avoid memory leaks, your application *must* balance every `SRNew...` or `SRGet...` call with a call to `SRReleaseObject`.

A SIMPLE SPEECH RECOGNITION EXAMPLE

It's easy to add simple speech recognition capabilities to your application. All you need to do is perform a small number of operations in sequence:

1. Initialize speech recognition by determining whether a valid version of the Speech Recognition Manager is installed, opening an `SRRecognitionSystem`, allocating an `SRRecognizer`, and installing an Apple event handler to handle recognition result notifications.
2. Build a language model that specifies the utterances your application is listening for.
3. Set the recognizer's active language model to the one you built and call `SRStartListening` to start listening and processing recognition result notifications.

We'll describe each of these operations in more detail.

INITIALIZING SPEECH RECOGNITION

First, you must verify that a valid version of the Speech Recognition Manager is installed on the target machine. Listing 1 shows how to do this. Note that only versions 1.5 and later of the Speech Recognition Manager adhere to the API used in this article.

Listing 2 shows how to open an `SRRecognitionSystem`, allocate an `SRRecognizer`, and install your Apple event handler. All of this happens when your application starts up. The Apple event handler `HandleRecognitionDoneAE` is shown later (in Listing 4).

Listing 1. Determining the Speech Recognition Manager version

```
Boolean HasValidSpeechRecognitionVersion (void)
{
    OSErr          status;
    long           theVersion;
    Boolean        validVersion          = false;
    const unsigned long kMinimumRequiredSRMVersion = 0x00000150;

    status = Gestalt(gestaltSpeechRecognitionVersion, &theVersion);
    if (!status)
        if (theVersion >= kMinimumRequiredSRMVersion)
            validVersion = true;

    return validVersion;
}
```

Listing 2. Initializing the Speech Recognition Manager

```
/* Our global variables */
SRRecognitionSystem gRecognitionSystem = NULL;
SRRecognizer        gRecognizer       = NULL;
SRLanguageModel    gTopLanguageModel = NULL;
AEEEventHandlerUPP gAERoutineDescriptor = NULL;

OSErr InitSpeechRecognition (void)
{
    OSErr status = kBadSRMVersion;

    /* Ensure that the Speech Recognition Manager is available. */
    if (HasValidSpeechRecognitionVersion()) {
        /* Open the default recognition system. */
        status = SROpenRecognitionSystem(&gRecognitionSystem,
                                         kSRDefaultRecognitionSystemID);

        /* Use standard feedback window and listening modes. */
        if (!status) {
            short feedbackNeeded = kSRHasFeedbackHasListenModes;

            status = SRSetProperty(gRecognitionSystem,
                                   kSRFeedbackAndListeningModes, &feedbackNeeded,
                                   sizeof(feedbackNeeded));
        }

        /* Create a new recognizer. */
        if (!status)
            status = SRNewRecognizer(gRecognitionSystem, &gRecognizer,
                                     kSRDefaultSpeechSource);

        /* Install our Apple event handler for recognition results. */
        if (!status) {
            status = memFullErr;
        }
    }
}
```

(continued on next page)

Listing 2. Initializing the Speech Recognition Manager (*continued*)

```
gAERoutineDescriptor =
    NewAEEEventHandlerProc(HandleRecognitionDoneAE);
if (gAERoutineDescriptor)
    status = AEInstallEventHandler(kAESpeechSuite, kAESpeechDone,
        gAERoutineDescriptor, 0, false);
}
}

return status;
}
```

Notice in Listing 2 how we call `SRSetProperty` to request Apple’s standard feedback and listening modes for the recognizer. To have a successful experience with speech recognition, users need good feedback indicating when the recognizer is ready for them to talk and what utterances the recognizer has recognized (for more on giving feedback, see “Speech Recognition Tips”). In addition, because of the recognizer’s tendency to misinterpret background conversation and noises as speech, it’s usually a good idea to let the user tell the recognizer when to listen by pressing a predefined key (the “push-to-talk” key). Your application can get all of this important behavior for free, simply by setting the `kSRFeedbackAndListeningModes` property.

With Apple’s Speech control panel (which comes bundled on new Macintoshes and with system updates), users can tailor this behavior to suit their needs, choosing preferred feedback characters (that is, the cartoon faces displayed in the feedback window) and preferred push-to-talk keys.

BUILDING A SIMPLE LANGUAGE MODEL

Your application needs to build a language model — `gTopLanguageModel` in our sample code — that specifies what the recognizer is listening for. The routine in Listing 3 shows how your application can create a simple language model. (We’ll discuss fancier language models later in this article.) Even simple language models should avoid using phrases that sound similar to one another; just like a human listener, the recognizer may have a hard time distinguishing between similar-sounding phrases.

A recognizer returns a special speech object, called the *rejection word*, if it hears an utterance but cannot recognize it. Listing 3 sets the reference constant of the top-level language model to a predefined value to be able to distinguish that model from the rejection word.

Note in Listing 3 that we add the phrases “Hello,” “Goodbye,” and “What time is it?” to our `gTopLanguageModel` using the call `SRAddText`, a convenient shortcut for the sequence of calls `SRNewPhrase`, `SRAddLanguageObject`, and `SRReleaseObject`. `SRAddText` also sets the `kSRRefCon` property of each added phrase. We’ll use this reference constant when we examine the recognition result to help determine what was said.

HANDLING RECOGNITION RESULT NOTIFICATIONS

Now let’s look at how your application would process result notifications given this simple language model.

SPEECH RECOGNITION TIPS

Speech recognition is a completely new input mode, and using it properly isn't always as straightforward as it might seem. While we don't yet have a complete set of human interface guidelines to guarantee a consistent and intuitive speech recognition user experience, there are a few simple rules that all speech recognition applications should follow.

GIVE FEEDBACK

Your application must always provide feedback to let users know when they can speak, when their utterance has been recognized, and how it was interpreted. The feedback services in the Speech Recognition Manager perform this for you, using the standard feedback window shown in Figure 3. (The user's recognized utterances are shown in italics, and the displayed feedback is in plain text. The string under the feedback character's face indicates the push-to-talk key.) All you need to do is set the `kSRFeedbackAndListeningModes` property as shown in Listing 2.



Figure 3. Standard feedback window

Your application should use this standard feedback behavior unless you have a very good reason to provide your own feedback and custom push-to-talk options. (Fast action games that take over the entire screen and don't call `WaitNextEvent` are examples of applications that wouldn't use the standard feedback.) Not only will users enjoy the benefits of consistent behavior, but as Apple improves the feedback components, your speech

recognition applications will automatically inherit this improved behavior without having to be recompiled.

SHOW WHAT CAN BE SAID

Successful speech recognition applications always let the user know what he or she can say. The way they achieve this depends on the application, but one good example is a Web browser that makes all visible hyperlinks speakable. This lets the user know what can be said while restricting the size of the language model to improve recognition accuracy.

CONSTRAIN THE LANGUAGE MODEL

The recognition technology currently used by the Speech Recognition Manager works best when it's listening for a small number of distinct utterances. The longer an utterance is, the more easily it can be distinguished from other utterances. For example, distinguishing the isolated words *hot*, *cut*, and *quit* is difficult and error prone. Recognition performance also decreases as the language model grows. The larger the language model, the more time the recognizer must spend searching for a matching utterance and the larger the likelihood of two utterances in the language model sounding similar. For best results, limit the size of the language model to fewer than a hundred phrases at any time and avoid including phrases that are easily confused when spoken, like "wreck a nice beach" and "recognize speech."

DO SOMETHING DIFFERENT

Compelling applications of speech recognition are often novel ones. Instead of simply paralleling an application's graphical user interface with a spoken one (making all menu items speakable, for example), do something different — something that takes advantage of the unique properties of speech. Combine speech synthesis with speech recognition to engage the user in a brief dialog. Use efficient language models to allow a single utterance to trigger a series of commands that might otherwise require interaction with dialog boxes. Let the power of speech recognition augment the graphical interface your users are already familiar with. Use your imagination!

In Listing 4, `HandleRecognitionDoneAE`, our Apple event handler, uses the routine `AEGetParamPtr` to extract the status of the result as well as the recognizer and recognition result objects from the Apple event.

At this point, the Apple event handler could easily get the text of what was heard by getting the `kSRTEXTFormat` property of the recognition result. But a more useful form of the result is the `kSRLanguageModelFormat`. This language model parallels the language model `gTopLanguageModel`, but instead of containing all of the phrases "Hello," "Goodbye," and "What time is it?" it contains only a copy of the phrase

Listing 3. Building a simple language model

```
OSErr BuildLanguageModel (void)
{
    OSErr      status;
    const char  kLMName[]  = "<Top LM>";

    /* First, allocate the gTopLanguageModel language model. */
    status = SRNewLanguageModel(gRecognitionSystem, &gTopLanguageModel, kLMName, strlen(kLMName));
    if (!status) {
        long refcon = kTopLMRefcon;

        /* Set the reference constant of our top language model so that when we process our */
        /* recognition result, we'll be able to distinguish it from the rejection word, "???". */
        status = SRSetProperty(gTopLanguageModel, kSRRefCon, &refcon, sizeof(refcon));
        if (!status) {
            const char *kSimpleStr[] = { "Hello", "Goodbye", "What time is it?", NULL };
            char **currentStr = (char **) kSimpleStr;
            long refcon = kHelloRefCon;

            /* Add each of the strings in kSimpleStr to the language model, and set the refcon to */
            /* the index of the string in the kSimpleStr array. */
            while (*currentStr && !status) {
                status = SRAddText(gTopLanguageModel, *currentStr, strlen(*currentStr), refcon++);
                ++currentStr;
            }

            /* Augment this simple language model with a fancier one. */
            if (!status)
                status = AddFancierLanguageModel(gTopLanguageModel);
        }
    }
    return status;
}
```

Listing 4. Handling the recognition-done Apple event

```
pascal OSErr HandleRecognitionDoneAE (AppleEvent *theAEvt, AppleEvent *reply, long refcon)
{
    OSErr      recognitionStatus = 0, status;
    long      actualSize;
    DescType   actualType;

    /* Get recognition result status. */
    status = AEGGetParamPtr(theAEvt, keySRSpeechStatus, typeShortInteger, &actualType,
        (Ptr) &recognitionStatus, sizeof(recognitionStatus), &actualSize);

    /* Get the SRRecognizer. */
    if (!status && !recognitionStatus) {
        SRRecognizer recognizer;
        status = AEGGetParamPtr(theAEvt, keySRRecognizer, typeSRRecognizer, &actualType,
            (Ptr) &recognizer, sizeof(recognizer), &actualSize);
    }
}
```

(continued on next page)

Listing 4. Handling the recognition-done Apple event (*continued*)

```
/* Get the SRRecognitionResult. */
if (!status) {
    SRRecognitionResult recResult;
    status = AEGgetParamPtr(theAEvt, keySRSpeechResult, typeSRSpeechResult, &actualType,
        (Ptr) &recResult, sizeof(recResult), &actualSize);

    /* Extract the language model from the result. */
    if (!status) {
        SRLanguageModel resultLM;
        long propertySize = sizeof(resultLM);

        status = SRGetProperty(recResult, kSRLanguageModelFormat, &resultLM, &propertySize);

        /* Process the language model. */
        if (!status) {
            status = ProcessRecognitionResult(resultLM, recognizer);

            /* What we SRGot we must SRRelease! */
            SRReleaseObject(resultLM);
        }
        /* Also release the recognition result. */
        SRReleaseObject(recResult);
    }
}
return noErr;
}
```

that was recognized. For example, if the user said “Goodbye,” the language model returned in the `kSRLanguageModelFormat` property would contain one phrase, which would have a `kSRSpelling` property of “Goodbye” and a `kSRRefCon` property of 1 (the value passed for that phrase in the `SRAAddText` call in Listing 3). The `ProcessRecognitionResult` routine (Listing 5) uses the language model to determine what was said by getting the `kSRRefCon` property of the spoken phrase and responding appropriately.

This example uses the `SRSpeakAndDrawText` routine to respond to recognition events. The Speech Recognition Manager uses the Speech Synthesis Manager to speak the string, and the animated feedback character (displayed in Apple’s standard feedback window) lip-synchs with the synthesized text. The Speech Recognition Manager also displays the response text in the feedback window. (You can use other routines to simply speak a string through the feedback window without displaying it, or to display a string without speaking it.)

SETTING THE ACTIVE LANGUAGE MODEL AND STARTING TO LISTEN

All we need to do now is make the language model we’ve built, `gTopLanguageModel`, the *active* language model and tell our recognizer to start listening. First we call the `SRSetLanguageModel` function, which associates `gTopLanguageModel` with the `SRRecognizer` we’ve allocated, `gRecognizer`:

```
OSErr status = SRSetLanguageModel(gRecognizer, gTopLanguageModel);
```

Listing 5. Processing a recognition result

```
OSErr ProcessRecognitionResult (SRLanguageModel resultLM, SRRecognizer recognizer)
{
    OSErr    status = noErr;

    if (resultLM && recognizer) {
        long    refcon;
        long    propertySize = sizeof(refcon);

        /* Get the refcon of the root object */
        status = SRGetProperty(resultLM, kSRRefCon, &refcon, &propertySize);

        /* Is the resultLM a subset of our top language model or is it the rejection word, "???"? */
        if (!status && refcon == kTopLMRefcon) {
            SRLanguageObject languageObject;
            propertySize = sizeof(languageObject);

            /* The resultLM contains either an SRPhrase or an SRPath. We use the refcon property */
            /* set in our language model building routine to distinguish between the results. */

            /* Get the phrase or path. */
            status = SRGetIndexedItem(resultLM, &languageObject, 0);
            if (!status) {
                long refcon;
                propertySize = sizeof(refcon);

                /* Get the refcon of the language object. */
                status = SRGetProperty(languageObject, kSRRefCon, &refcon, &propertySize);
                if (!status) switch (refcon) {
                    case kHelloRefCon:
                    case kGoodbyeRefCon:
                    case kWhatTimeIsItRefCon:
                        {
                            const char *kResponses[] = { "Hi there!", "Don't leave now!",
                                                            "It's time to use the Speech Recognition Manager!"
                                                        };

                            /* Speak and display our response using the feedback character. */
                            /* Use the refcon as an index into our response array. */
                            status = SRSpeakAndDrawText(recognizer, kResponses[refcon],
                                                            strlen(kResponses[refcon]));
                        }
                            break;
                    case kCompanyRefCon:
                        status = ProcessFancierLanguageModel(languageObject, recognizer);
                        break;
                }
                /* Always SRRelease what we SRGot. */
                status = SRReleaseObject(languageObject);
            }
        }
    }
    return status;
}
```

You can build as many language models as you like, but there is always just one that's active. You can make another language model active (and thereby deactivate the one that was previously active), or you can enable and disable parts of the active language model. Typically this is done in response to a speech-detected Apple event, sent to the application when recognition is about to begin.

For a good example of making your language model dynamically conform to the context of your application, see the article "Adding Speech Recognition to an Application Framework" in this issue of *develop*.

Once we've set the active language model, we start the recognition process by calling `SRStartListening`, as follows:

```
if (!status)
    status = SRStartListening(gRecognizer);
```

Now we can start speaking to our application. When an utterance is recognized as belonging to our language model, our Apple event handler, `HandleRecognitionDoneAE`, will be called and the recognition result will be processed. It's that easy!

CLEANING UP

Listing 6 shows how to clean up when your application quits. In general, you should release the speech objects in the order shown.

BUILDING FANCIER LANGUAGE MODELS

The Speech Recognition Manager provides several routines that your application can use to create and manipulate fancier language models than the one created earlier in Listing 3. For example, suppose you wanted to create an application that responds to users when they say, "Tell me the price of <company> stock," where <company> is one of several company names.

To create a language model like this, your application needs to create an `SRPath` object that consists of the phrase "Tell me the price of" followed by an *embedded* language model representing the company names, followed by the word "stock." The `AddFancierLanguageModel` function creates this path and adds it to the language model created in Listing 3. (Note that the embedded company language model is simply a list of phrases, just like the language model we created in Listing 3.)

Figure 4 shows the structure of the entire language model. We've limited the number of companies to three here for simplicity. The top half of each box shows the spelling and `refcon` properties of each object; the lower half indicates the object type.

Take a look at the `AddFancierLanguageModel` function (not shown, but included with our sample code) to see how to build the fancier language model. (Don't worry if this routine seems like a lot of code just to add the command "Tell me the price of <company> stock"; below we'll describe the `SRLanguageModeler` tool, which makes the creation of complicated static language models very easy.) Listing 7 shows how your application would process results given this fancier language model.

Speech recognition applications that support utterances like "Tell me the price of <company> stock" or "Call <name>," while limiting <company> or <name> to a few dozen items, can be more compelling than those that just respond to simple phrases. They're nicely limited in scope, yet they allow the user to invoke actions more easily than would be possible with a graphical user interface. What other technology does that?

Listing 6. Terminating speech recognition

```
void TerminateSpeechRecognition (void)
{
    OSErr status = noErr;

    /* If we have an active language model, release it. */
    if (gTopLanguageModel) {
        status = SRReleaseObject(gTopLanguageModel);
        gTopLanguageModel = NULL;
    }

    /* If we have a recognizer, release it. */
    if (gRecognizer) {
        status = SRStopListening(gRecognizer);
        status = SRReleaseObject(gRecognizer);
        gRecognizer = NULL;
    }

    /* If we have a recognition system, close it. */
    if (gRecognitionSystem) {
        status = SRCloseRecognitionSystem(gRecognitionSystem);
        gRecognitionSystem = NULL;
    }

    /* Remove our Apple event handler and dispose of the handler's */
    /* routine descriptor. */
    if (gAERoutineDescriptor) {
        status = AERemoveEventHandler(kAESpeechSuite, kAESpeechDone,
                                     gAERoutineDescriptor, false);
        DisposeRoutineDescriptor(gAERoutineDescriptor);
        gAERoutineDescriptor = NULL;
    }
}
```

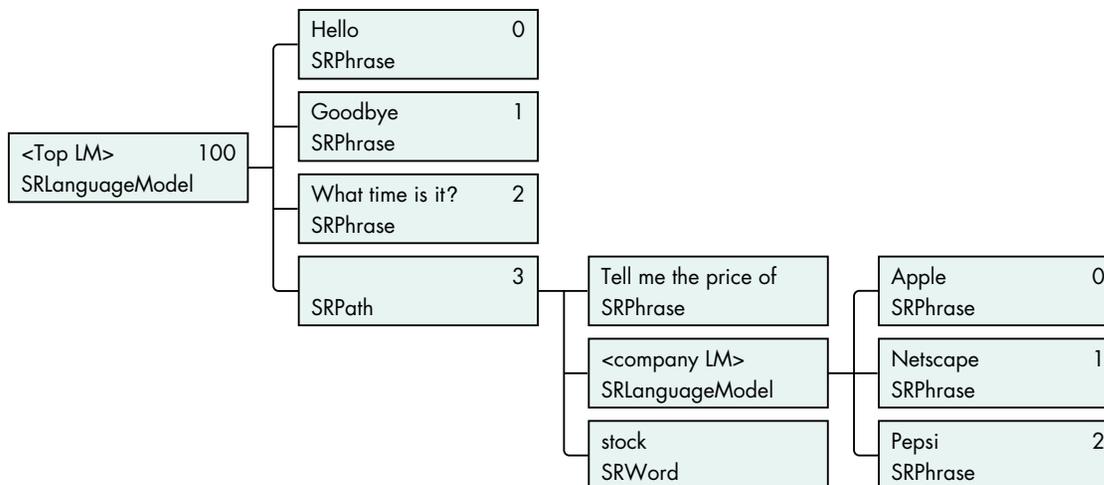


Figure 4. Language model built by calling BuildLanguageModel

Listing 7. Processing a recognition result given a fancier language model

```
OSErr ProcessFancierLanguageModel (SRPath resultPath, SRRecognizer recognizer)
{
    OSErr    status = noErr;

    if (resultPath && recognizer) {
        SRLanguageModel companyLM;

        /* Get the second item in the path -- it's the company language model. */
        status = SRGetIndexedItem(resultPath, &companyLM, 1);
        if (!status && companyLM) {
            SRPhrase companyName;

            /* In the result language model, the company language model contains just one phrase. */
            status = SRGetIndexedItem(companyLM, &companyName, 0);
            if (!status) {
                long refcon;
                long propertySize = sizeof(refcon);

                /* Get the refcon from the company name. It's our index into the response array. */
                status = SRGetProperty(companyName, kSRRefCon, &refcon, &propertySize);
                if (!status) {
                    const char *kResponses[] =
                        { "Apple stock is priced to move!",
                          "Netscape is trading at fifty dollars.",
                          "Why would you want to know that?"
                        };
                    status = SRSpeakAndDrawText(recognizer, kResponses[refcon],
                                                strlen(kResponses[refcon]));
                }
                /* What we SRGot we must SRRelease. */
                status = SRReleaseObject(companyName);
            }
            status = SRReleaseObject(companyLM);
        }
    }
    return status;
}
```

MANIPULATING LANGUAGE MODELS

The Speech Recognition Manager contains several more routines and properties for manipulating language models. We'll look at a few of them here.

Your application can create a large language model and then use the `SRSetProperty` function to disable and enable parts of it quickly on the fly, as shown in Listing 8. By enabling only parts of a language model, you can minimize the number of utterances that the recognizer is listening for.

Your application can change, clear, or rebuild parts of a language model dynamically to reflect the current context of your program. Listing 9 clears and then rebuilds the company language model that was originally built by the `AddFancierLanguageModel` function.

Listing 8. Disabling a part of a language model

```
/* Disable the stockPath part of the gTopLanguageModel. */
/* The stock path is the fourth item in this language model. */

SRPath  stockPath;
OSErr   status = SRGetIndexedItem(gTopLanguageModel, &stockPath, 3);

if (!status) {
    Boolean enabled = false;
    status = SRSetProperty(stockPath, kSREnabled, &enabled,
                          sizeof(enabled));

    /* Balance SRGet call. */
    status = SRReleaseObject(stockPath);
}
```

Listing 9. Emptying and refilling the company language model

```
/* Empty and refill the embedded company language model. */
/* Assume that stockPath has already been initialized. */

/* The companyLM is the second item in the stock path. */
SRLanguageModel  companyLM;
OSErr             status = SRGetIndexedItem(stockPath, &companyLM, 1);

if (!status) {
    /* This releases each phrase in the company language model. */
    status = SREmptyLanguageObject(companyLM);

    /* Now rebuild the company language model with new companies. */
    if (!status) {
        const char  *kNewCompanies[] = { "I B M", "Motorola",
                                          "Coca-Cola", NULL };

        char        **company        = (char **) kNewCompanies;
        long         refcon           = 0;

        while (*company && !status) {
            status = SRAddText(companyLM, *company, strlen(*company),
                              refcon++);

            ++company;
        }
    }
    SRReleaseObject(companyLM);
}
```

At any given moment, the active language model should be relatively small, but your application can change the set of active phrases at any time. For example, if a Web browser application made its links speakable, at any given moment there would only be a few dozen visible links, so there would only be a few dozen phrases active. But if you spent a couple of hours surfing the Web with that browser, you would have seen many thousands of links throughout the session, and you could have spoken any one of them while it was visible.

In addition to enabling and disabling parts of your language model, the `SRSetProperty` function allows your application to make words, phrases, paths, or language models repeatable (so that the user can say that item one or more times in a row) or rejectable (so that if the user says something else for that item, the recognizer will fill it in with a special rejection word with a spelling of “???”).

Your application can also make any word, phrase, path, or language model optional by setting the corresponding object’s `kSROptional` property to true. In `AddFancierLanguageModel`, we’ve set the `kSROptional` property of the `SRWord` “stock” to true, so the recognizer is ready for the user to say, “Tell me the price of Apple” as well as “Tell me the price of Apple stock.”

Your application doesn’t have to build language models from scratch each time it runs. The Speech Recognition Manager provides routines for saving and loading language objects (for example, the `SRPutLanguageObjectIntoHandle` and `SRNewLanguageObjectFromDataFile` routines). Listing 10 shows an example.

Listing 10. Saving a language model into a resource

```
/* Allocate a handle of size 0 to store our language model in; */
/* SRPutLanguageObjectIntoHandle will resize it as needed. */
Handle  lmHandle = NewHandle(0);
OSErr   status   = MemError();

if (!status) {
    status = SRPutLanguageObjectIntoHandle(gTopLanguageModel, lmHandle);
    if (!status) {
        /* Save the language model as a resource in the current */
        /* resource file. Pick a reasonable resource type and ID. */
        AddResource(lmHandle, 'LMDL', 100, "\pTop Language Model");

        /* Make sure it gets written to disk. */
        if (!(status = ResError())) {
            WriteResource(lmHandle);
            DetachResource(lmHandle);
        }
    }

    DisposeHandle(lmHandle);
}
```

Apple provides a very handy developer tool, called `SRLanguageModeler`, that you can use to quickly create, test, and save language models into resources or data files. You can find this tool, and documentation for it, with the other Speech Recognition Manager developer information on this issue’s CD and on the speech technology Web site. `SRLanguageModeler` lets you write out a language model in a relatively simple text form and then try it out to see how well its phrases can be recognized and discriminated from one another. It lets you save the language models into a binary resource or file format that you can ship with your application. Your application can load the language model at run time with `SRNewLanguageObjectFromHandle` or `SRNewLanguageObjectFromDataFile`. `SRLanguageModeler` will eliminate a lot of the code you would otherwise have to write to construct the static parts of your language models.

SPEECH: THE FINAL FRONTIER

If you've understood this article, you'll have no problem making practical use of speech recognition in your application. From the basics of checking for the proper version of the Speech Recognition Manager to some of the finer details of building language models, we've shown you everything you need to know to get started. Be sure to take a look at the SRSample application, which uses many of the listings in this article.

To dig even deeper, check out the Speech Recognition Manager documentation and the SRLanguageModeler tool. For tips on using the Speech Recognition Manager within an application framework and dynamically changing your language model, see the article "Adding Speech Recognition to an Application Framework" in this issue of *develop*. Then have fun turning your application into a good listener.

RELATED READING

- "Speech Recognition Manager," on this issue's CD and on Apple's speech technology Web site, <http://www.speech.apple.com>.
- "Adding Speech Recognition to an Application Framework" by Tim Monroe, in this issue of *develop*.

Thanks to our technical reviewers Mike Dilts, Eric "Braz" Ford, Tim Monroe, and Guillermo Ortiz. •

PlainTalk™

Make your products stand out from the crowd.

Add a new dimension to your products with Apple's speech technologies.

Apple's Speech Development Kits are online and **free!** Create speech-savvy applications that engage your customers and draw them into rich, immersive environments. Apple's Speech APIs let you incorporate speech recognition and synthesis into your applications quickly and easily. Master the power of Apple's speech technology today. Download the free Speech Development Kits at <http://www.speech.apple.com>.

The Apple Speech Development Kits include the Speech Recognition Manager, the Speech Synthesis Manager, APIs, extensions, sample code, libraries, and documentation. (Online service and computer not included.)

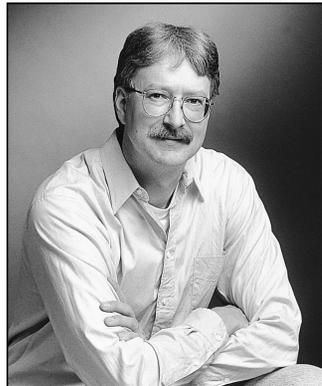


 Apple Computer, Inc.

©1996 Apple Computer, Inc. All rights reserved. Apple, the Apple logo, PlainTalk and Power Macintosh are registered trademarks of Apple Computer, Inc. Mac is a trademark of Apple Computer, Inc.

Adding Speech Recognition to an Application Framework

It's easy to add speech recognition capabilities to an application built with an object-oriented framework, with minimal disruption to your existing code. To illustrate the process, this article shows one way to add basic speech recognition capabilities to an application built with PowerPlant, Metrowerks' popular C++-based application framework. You can use the same strategy with other application frameworks as well.



TIM MONROE

Speech recognition capabilities, such as those provided by Apple's Speech Recognition Manager, promise to revolutionize the way people use computers. The reason for this is simple: it's often a lot easier to say what you want done than to actually do it, even in the "user-friendly" environment provided by the Macintosh graphical user interface. So the time you spend making your application speakable is time *very* well spent. Happily, if you've built your application with a framework such as PowerPlant or MacApp, you can add basic speech recognition capabilities quickly and easily.

To show how to add speech recognition to an application built with a framework, we'll modify the PowerPlant DocDemo sample provided with the CodeWarrior 8 release to add speech support for the File menu commands. Of course, there's nothing special about DocDemo: you should be able to drop the code we provide into any PowerPlant application. Moreover, although this code is specific to PowerPlant, you should be able to use similar techniques with other application frameworks as well.

Before reading this article, you should be familiar with the basic operations of the Speech Recognition Manager and with the PowerPlant application framework. For an overview of the Speech Recognition Manager, see the article "The Speech Recognition Manager Revealed" in this issue of *develop*. As mentioned in that article, you'll find everything you need to use the Speech Recognition Manager — including detailed documentation (written by yours truly) — on this issue's CD and on Apple's speech technology Web site at <http://www.speech.apple.com>. For basic information about PowerPlant, see *The PowerPlant Book* or other Metrowerks documentation.

THE BASIC STRATEGY

We want to add speech support for the File menu commands in the DocDemo application. This isn't the highest or best use of speech recognition capabilities (see

TIM MONROE (monroe@apple.com) is a technical writer for Apple's Developer Relations group. He's written more *Inside Macintosh* books and chapters than he cares to remember and is currently working with the QuickDraw 3D and QuickTime VR teams, as well as the speech recognition team, to bring the excitement of

interactive media to Macintosh applications everywhere. He's rumored to have an office in Cupertino but prefers to spend his time in his converted garage in Oakland living the quiet life of a telecommuting "cybermonk." That way, he's never too far from his wife, his kids, or his model train layout. •

“Speakable Menus?”), but it makes a simple example for us to focus on. In a nutshell, we’ll define a custom C++ class and create a single instance of that class to handle all the required speech recognition processing (such as installing a language model and responding to recognition results sent to it via Apple events). Here are the steps we’ll follow:

1. Add a few lines of code to the main application source code file, CDocDemoApp.cp. In part, this code creates a single instance of our custom class CDocSpeech.
2. Design a set of language models that describe the words and phrases we want to listen for.
3. Add resources containing string representations of those words and phrases to the application’s resource file.
4. Write Apple event handlers for the two speech recognition events.

The following sections explain these steps in detail, though not strictly in this order. All the code provided here is also included on this issue’s CD.

HOOKING UP WITH THE MAIN APPLICATION

All the speech recognition processing for our PowerPlant-based application will be handled by a single custom object of type CDocSpeech. The main application code needs only to create (and later delete) that custom object. We’ll start by adding these lines of code to the beginning of the main application source code file, CDocDemoApp.cp:

```
#include "CDocSpeech.h"
extern CDocSpeech *gDocSpeechObj;
Boolean          gHasSpeechRecog;
```

The external reference is to an instance of the CDocSpeech class, and the Boolean global variable indicates whether the Speech Recognition Manager is available in the current operating environment. To set that variable and create our custom object, we add the code in Listing 1 to the constructor CDocDemoApp::CDocDemoApp.

We’ll also need to delete gDocSpeechObj when our application quits. We do this by adding the following code to the destructor CDocDemoApp::~CDocDemoApp:

SPEAKABLE MENUS?

While it’s fairly easy to make your application’s menus speakable, this isn’t necessarily the best use of speech recognition technology and it’s definitely not what Apple’s speech engineers would like to see you focus your attention on. Most File and Edit menu commands are just too short to be easily distinguished by the recognizer (“quit” sounds a lot like “cut,” for example).

In addition, since menus can’t be seen without pulling them down, novice users probably won’t know which menu commands are available until they click in the menu bar; at that point, they may as well just use the menu.

However, there *is* some value in knowing how to make menus speakable. For one thing, the techniques used in this article can easily be extended to handle more complex utterances that have nothing to do with menus. Also, there is real value in making tool palettes — which are really just graphical menus that happen to float on the desktop — speakable; for an example, see the demo program PlacMac on this issue’s CD.

So the moral is: make your menus speakable if you think there is value for the user, but don’t *just* make your menus speakable. Do something creative and compelling with speech recognition.

```
// Shut down speech recognition, if it's running.
if (gHasSpeechRecog)
    delete gDocSpeechObj;
```

Those are all the modifications we need to make to our existing source code! The rest of the speech processing is handled by the custom speech recognition object created by our main application code.

DEFINING A SPEECH RECOGNITION CLASS

The header file CDocSpeech.h, shown in Listing 2, defines a number of constants specifying the 'STR#' resources (and indices within those resources) that contain the names of the language models we want to create and the actual words or phrases we want to listen for. We'll use these constants later, when we create the various language models.

Listing 1. Creating a custom speech recognition object

```
// Determine whether the Speech Recognition Manager is available;
// if it's available, create a custom speech recognition object.
long    theVersion;
OSErr   theErr;

gHasSpeechRecog = false;
theErr = ::Gestalt(gestaltSpeechRecognitionVersion, &theVersion);
// Version must be at least 1.5.0 to support API used here.
if (!theErr)
    if (theVersion >= 0x00000150) {
        gHasSpeechRecog = true;
        gDocSpeechObj = new CDocSpeech();
    }
```

Listing 2. Specifying 'STR#' resources and declaring CDocSpeech

```
#include "SpeechRecognition.h"

// Language model names
const ResIDT  rSTR_LMNames      = 400; // ID of STR# resource
const short   kStr_GApplLM      = 1;   // Indices within resource
const short   kStr_GUnivLM      = 2;
const short   kStr_GDocuLM      = 3;
const short   kStr_UFileLM      = 4;
const short   kStr_DFileLM      = 5;

// Universal file command phrases
const ResIDT  kSTR_UFileCmds    = 500; // ID of STR# resource
const short   kStr_New          = 1;   // Indices within resource
const short   kStr_Open        = 2;
const short   kStr_PageSetup    = 3;
const short   kStr_Quit        = 4;
```

(continued on next page)

Listing 2. Specifying 'STR#' resources and declaring CDocSpeech (*continued*)

```
// Document file command phrases
const ResIDT  kSTR_DFileCmds  = 501; // ID of STR# resource
const short   kStr_Close      = 1;   // Indices within resource
const short   kStr_Save       = 2;
const short   kStr_SaveAs     = 3;
const short   kStr_Revert     = 4;
const short   kStr_Print      = 5;
const short   kStr_PrintOne   = 6;

// Apple menu command phrases
const ResIDT  kSTR_UApplCmds  = 503; // ID of STR# resource
const short   kStr_About      = 1;   // Indices within resource

#define kEnableObj           true
#define kDisableObj          false

class CDocSpeech {
public:
    CDocSpeech();
    ~CDocSpeech();
    virtual
    static pascal OSErr HandleSpeechBegunAppleEvent (AppleEvent
        *theAEvt, AppleEvent *reply, long refcon);
    static pascal OSErr HandleSpeechDoneAppleEvent (AppleEvent
        *theAEvt, AppleEvent *reply, long refcon);
private:
    OSErr MakeLanguageModels (void);
};
```

CDocSpeech.h also contains the declaration of the custom CDocSpeech class. CDocSpeech is extremely simple: it contains a constructor, a destructor, and two Apple event handlers. It also defines a private method, MakeLanguageModels, that creates the language models used by DocDemo. MakeLanguageModels is called by the constructor when an instance of the CDocSpeech class is created.

All the remaining code is found in the file CDocSpeech.cp. Listing 3 shows the beginning of that file, which declares all the global variables and function prototypes.

Listing 3. Declaring global variables and function prototypes

```
#include "CDocSpeech.h"

// Global variables
SRRecognitionSystem  gSystem;
SRRecognizer         gRecognizer;
SRLanguageModel      gGApplLM, gGDocuLM;
SRPhrase             gRevert;
CDocSpeech           *gDocSpeechObj = nil;

// Function prototypes
void SetLanguageObjectState (SRLanguageObject inObj, Boolean isEnabled);
```

The constructor method, shown in Listing 4, performs all the necessary startup associated with speech recognition. Much of this code should already be familiar to you from the article “The Speech Recognition Manager Revealed.”

Now we just need to write the `MakeLanguageModels` function called by the `CDocSpeech` constructor, and the two Apple event handlers.

Listing 4. Starting up speech recognition

```
CDocSpeech::CDocSpeech()
{
    OSErr    theErr = noErr;

    // Open a recognition system.
    theErr = ::SROpenRecognitionSystem(&gSystem, kSRDefaultRecognitionSystemID);

    // Set recognition system properties to user-selected feedback and listening modes.
    if (!theErr) {
        short theModes = kSRHasFeedbackHasListenModes;
        theErr = ::SRSetProperty(gSystem, kSRFeedbackAndListeningModes, &theModes, sizeof(theModes));
    }

    // Create a recognizer with default speech source.
    if (!theErr)
        theErr = ::SRNewRecognizer(gSystem, &gRecognizer, kSRDefaultSpeechSource);

    // Set recognizer properties. We want to receive notifications when recognition begins and ends.
    if (!theErr) {
        unsigned long theParam = kSRNotifyRecognitionBeginning | kSRNotifyRecognitionDone;
        theErr = ::SRSetProperty(gRecognizer, kSRNotificationParam, &theParam, sizeof(theParam));
    }

    // Install Apple event handlers.
    if (!theErr) {
        theErr = ::AEInstallEventHandler(kAESpeechSuite, kAESpeechDetected,
                                         NewAEEEventHandlerProc(HandleSpeechBegunAppleEvent), 0, false);
        theErr = ::AEInstallEventHandler(kAESpeechSuite, kAESpeechDone,
                                         NewAEEEventHandlerProc(HandleSpeechDoneAppleEvent), 0, false);
    }

    // Make our language models.
    if (!theErr)
        theErr = MakeLanguageModels();

    // Install initial language model and release our reference to it.
    if (!theErr) {
        theErr = ::SRSetLanguageModel(gRecognizer, gGApplLM);
        ::SRReleaseObject(gGApplLM);
    }

    // Have the recognizer start processing sound.
    if (!theErr)
        theErr = ::SRStartListening(gRecognizer);
    }
}
```

CONSTRUCTING THE LANGUAGE MODELS

Probably the most time-consuming part of adding speech recognition to an application is defining the language models that describe the words and phrases you want to listen for. The process is straightforward, but it requires careful attention to the various states your application can be in. This is because you want the active language model to include only utterances that make sense at any given time. For instance, if no document window is open, it makes no sense to listen for the Close or Save command. Similarly, if a document isn't dirty (that is, if it hasn't changed since it was most recently saved), you probably don't want the user to be able to execute the Revert command.

This should remind you, of course, of the context-specific menu enabling and disabling that's a standard part of any good Macintosh application. For our demonstration application, we'll handle context sensitivity by creating a number of embedded language models that we'll enable or disable according to context.

The commands in the File menu fall into two main categories: those that can be issued at any time (such as New or Open) and those that apply to a specific document (such as Save or Close). Accordingly, we'll construct two language models, one for each type of command. Let's call the first variety *universal file commands* and the second variety *document file commands*. In addition, we want to make the About DocDemo command utterable. Here's a Backus-Naur Form (BNF) diagram of our top-level language model:

```
<Menu Commands> = <Universal Commands> | <Document Commands>;
<Universal Commands> = <Universal File Commands> | About DocDemo;
<Universal File Commands> = New | Open | Page Setup | Quit;
<Document Commands> = <Document File Commands>;
<Document File Commands> = Close | Save | Save As | Revert | Print |
                          Print One;
```

As you can see, the top-level language model Menu Commands consists of two embedded language models, one for commands that can be issued at any time and one for commands that require a document window to be open. Each of these embedded language models contains other language objects. The Universal Commands language model contains the phrase "About DocDemo" and the language model that contains the universal file commands. The Document Commands language model contains only the language model that contains the document file commands; you would add other document-specific models here (for instance, document-specific editing commands). In all, we'll create five language models. (Note that the Page Setup command is in the universal file commands language model; that's because DocDemo allows you to choose that command even if no document window is open.)

Listing 5 shows the code defining the MakeLanguageModels function (error checking has been removed for the sake of readability). Apple provides a utility, SRLanguageModeler, that you can use to build and test language models described with BNF diagrams like that shown above. SRLanguageModeler can also save those language models into resources or files, from which your application can load the models at run time. Here, however, we build the language models on the fly to demonstrate the Speech Recognition Manager routines for doing so.

MakeLanguageModels begins by calling SRNewLanguageModel five times to create the five new, empty language models. (As indicated earlier, the names of the language models are read from the application's resource fork.) Then MakeLanguageModels creates a language object for the single word *revert*, as follows:

```
::GetIndString(theStr, kSTR_DFileCmds, kStr_Revert);
::SRNewPhrase(gSystem, &gRevert, &theStr[1], theStr[0]);
```

Listing 5. Creating the language models

```
OSErr CDocSpeech::MakeLanguageModels (void)
{
    OSErr          theErr = noErr;
    Str255         theStr;
    SRLanguageModel myGUnivLM, myUFileLM, myDFileLM;

    // Make the language models (which are initially empty).
    ::GetIndString(theStr, rSTR_LMNames, kStr_GApplLM);
    ::SRNewLanguageModel(gSystem, &gGApplLM, &theStr[1], theStr[0]);
    ::GetIndString(theStr, rSTR_LMNames, kStr_GUnivLM);
    ::SRNewLanguageModel(gSystem, &myGUnivLM, &theStr[1], theStr[0]);
    ::GetIndString(theStr, rSTR_LMNames, kStr_UFileLM);
    ::SRNewLanguageModel(gSystem, &myUFileLM, &theStr[1], theStr[0]);
    ::GetIndString(theStr, rSTR_LMNames, kStr_GDocuLM);
    ::SRNewLanguageModel(gSystem, &gGDocuLM, &theStr[1], theStr[0]);
    ::GetIndString(theStr, rSTR_LMNames, kStr_DFileLM);
    ::SRNewLanguageModel(gSystem, &myDFileLM, &theStr[1], theStr[0]);

    // Make any other language objects we'll need.
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_Revert);
    ::SRNewPhrase(gSystem, &gRevert, &theStr[1], theStr[0]);

    // ****<Universal File Commands>****
    ::GetIndString(theStr, kSTR_UFileCmds, kStr_New);
    ::SRAddText(myUFileLM, &theStr[1], theStr[0], cmd_New);
    ::GetIndString(theStr, kSTR_UFileCmds, kStr_Open);
    ::SRAddText(myUFileLM, &theStr[1], theStr[0], cmd_Open);
    ::GetIndString(theStr, kSTR_UFileCmds, kStr_PageSetup);
    ::SRAddText(myUFileLM, &theStr[1], theStr[0], cmd_PageSetup);
    ::GetIndString(theStr, kSTR_UFileCmds, kStr_Quit);
    ::SRAddText(myUFileLM, &theStr[1], theStr[0], cmd_Quit);

    // ****<Document File Commands>****
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_Close);
    ::SRAddText(myDFileLM, &theStr[1], theStr[0], cmd_Close);
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_Save);
    ::SRAddText(myDFileLM, &theStr[1], theStr[0], cmd_Save);
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_SaveAs);
    ::SRAddText(myDFileLM, &theStr[1], theStr[0], cmd_SaveAs);
    unsigned long theRefCon = cmd_Revert;
    ::SRSetProperty(gRevert, kSRRefCon, &theRefCon, sizeof(theRefCon));
    ::SRAddLanguageObject(myDFileLM, gRevert);
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_Print);
    ::SRAddText(myDFileLM, &theStr[1], theStr[0], cmd_Print);
    ::GetIndString(theStr, kSTR_DFileCmds, kStr_PrintOne);
    ::SRAddText(myDFileLM, &theStr[1], theStr[0], cmd_PrintOne);

    // ****<Document Commands>****
    ::SRAddLanguageObject(gGDocuLM, myDFileLM);
}
```

(continued on next page)

Listing 5. Creating the language models (*continued*)

```
// ****<Universal Commands>****
::SRAddLanguageObject(myGUnivLM, myUFileLM);
::GetIndString(theStr, kSTR_UApplCmds, kStr_About);
::SRAddText(myGUnivLM, &theStr[1], theStr[0], cmd_About);

// ****<Menu Commands>****
::SRAddLanguageObject(gGAppLLM, myGUnivLM);
::SRAddLanguageObject(gGAppLLM, gGDocuLM);

// Release any embedded language models we won't need later.
::SRReleaseObject(myDFileLM);
::SRReleaseObject(myUFileLM);
::SRReleaseObject(myGUnivLM);

return theErr;
}
```

We treat the Revert command specially because we want to listen for it only when an open document has a file associated with it (and, of course, when the document is dirty). Even when the Document Commands language model is active, the Revert command might need to be disabled.

Next, `MakeLanguageModels` builds the two language models Universal File Commands and Document File Commands. In both cases, it simply adds the relevant words or phrases, read from resources, to the language model, like this:

```
::GetIndString(theStr, kSTR_UFileCmds, kStr_New);
::SRAddText(myUFileLM, &theStr[1], theStr[0], cmd_New);
```

`SRAddText` sets the reference constant property of the specified language object to the value passed in its fourth parameter. In this example, the reference constant for the New command is set to the value `cmd_New`, which is a constant defined by PowerPlant. As you'll see later, we'll use that value to get PowerPlant to react appropriately to the user's utterances. If you don't use `SRAddText`, you need to explicitly set an object's reference constant property, as is done for the Revert command:

```
unsigned long theRefCon = cmd_Revert;
::SRSetProperty(gRevert, kSRRefCon, &theRefCon, sizeof(theRefCon));
::SRAddLanguageObject(myDFileLM, gRevert);
```

Once the two main language models have been created, the hierarchy displayed in the BNF diagram is established by a series of calls to `SRAddLanguageObject`.

ENABLING AND DISABLING THE LANGUAGE MODELS

When a user begins speaking, your application is notified via a speech-detected Apple event. In general, your speech-detected event handler should determine what state your application is in and set the active language model accordingly. As we've mentioned, we'll use this opportunity to enable or disable embedded language models (or even single words) to limit the recognizable utterances to those that make sense at the time. Listing 6 shows our speech-detected Apple event handler.

Listing 6. Handling speech-detected Apple events

```
pascal OSErr CDocSpeech::HandleSpeechDetectedAppleEvent
    (AppleEvent *theAEvt, AppleEvent *reply, long refcon)
{
#pragma unused(reply, refcon)
    long        actualSize;
    DescType    actualType;
    OSErr       theErr = 0, recStatus = 0;
    SRRecognizer theRec;
    LWindow     *theWindow;

    // Get status and recognizer.
    theErr = ::AEGetParamPtr(theAEvt, keySRSpeechStatus,
        typeShortInteger, &actualType, (Ptr)&recStatus,
        sizeof(recStatus), &actualSize);
    if (!theErr && !recStatus)
        theErr = ::AEGetParamPtr(theAEvt, keySRRecognizer,
            typeSRRecognizer, &actualType, (Ptr)&theRec,
            sizeof(theRec), &actualSize);
    if (theErr)
        if (!theRec)
            return theErr;

    // Figure out what state we're in; then enable or disable the
    // appropriate language models.
    theWindow = UDesktop::FetchTopRegular(); // Look for a doc window.
    if (theWindow != nil) { // There is a doc window.
        SetLanguageObjectState(gGDocuLM, kEnableObj);

        // Turn off "Revert" if there's no file or it isn't dirty.
        Boolean    isEnabled, outUsesMark;
        Char16     outMark;
        Str255     outName;

        LCommander::GetTarget()->FindCommandStatus(cmd_Revert, isEnabled,
            outUsesMark, outMark, outName);
        SetLanguageObjectState(gRevert, isEnabled);
    } else // There is no doc window.
        SetLanguageObjectState(gGDocuLM, kDisableObj);

    // Now tell the recognizer to continue.
    theErr = ::SRContinueRecognition(theRec);
    return theErr;
}
```

The event handler, `HandleSpeechDetectedAppleEvent`, calls the PowerPlant utility function `UDesktop::FetchTopRegular` to get the top document window. If there's an open document window, `HandleSpeechDetectedAppleEvent` calls the application-defined function `SetLanguageObjectState` to enable the Document Commands language model. Otherwise, if no document window is open, the event handler calls `SetLanguageObjectState` to disable that language model. Listing 7 shows the simple function `SetLanguageObjectState`.

Listing 7. Enabling or disabling a language object

```
void SetLanguageObjectState (SRLanguageObject inObj, Boolean isEnabled)
{
    Boolean    theState = isEnabled;

    ::SRSetProperty(inObj, kSREnabled, &theState, sizeof(theState));
}
```

Notice that if a document window is open, we need to determine whether to enable the Revert command. `HandleSpeechDetectedAppleEvent` cleverly calls the document window's `FindCommandStatus` function to determine this.

Instead of disabling the Revert command when it isn't relevant, we could just let the recognizer keep listening for it but ignore it when the frontmost document, if any, isn't dirty or has no file. This alternate strategy has some advantages. In particular, if the user says "revert" but we aren't listening for that command, the recognizer might think the user has uttered some other command (like "quit" or "print"). These misfires are much less likely to occur if the recognizer is listening for "revert" in addition to the other document file commands.

If you think that a user is apt to utter a particular command at an inappropriate time, it's probably better to ignore it than to disable it. On the other hand, we don't want to make the active language model too big, and one way to keep its size manageable is to enable or disable parts of it according to context. That's the strategy we've adopted for this article. Our sample application doesn't listen for the Revert command unless it's appropriate, to illustrate how to enable and disable language objects.

HANDLING RECOGNITION RESULTS

So far, we've defined our language models and set up the mechanism by which relevant parts of the language models are enabled or disabled according to context. All that remains is to do the right thing when the recognizer recognizes an utterance. Our application is informed of successful recognitions via recognition-done Apple events. Listing 8 shows the `DocDemo` recognition-done event handler.

Listing 8. Handling recognition-done Apple events

```
pascal OSErr CDocSpeech::HandleRecognitionDoneAppleEvent
    (AppleEvent *theAEvt, AppleEvent *reply, long refcon)
{
    #pragma unused(reply, refcon)
    long          actualSize;
    DescType      actualType;
    OSErr         theErr = 0, recStatus = 0;
    SRRecognitionResult recResult = nil;
    Size          theLen;
    SRPath        thePath;
    SRSpeechObject theItem;
    long          theRefCon;    // Reference constant of item
```

(continued on next page)

Listing 8. Handling recognition-done Apple events (*continued*)

```
// Get status.
theErr = ::AEGetParamPtr(theAEvt, keySRSpeechStatus,
                        typeShortInteger, &actualType, (Ptr)&recStatus,
                        sizeof(recStatus), &actualSize);

// Get result.
if (!theErr && !recStatus)
    theErr = ::AEGetParamPtr(theAEvt, keySRSpeechResult,
                            typeSRSpeechResult, &actualType, (Ptr)&recResult,
                            sizeof(recResult), &actualSize);

// Get command from result by reading the reference constant
// of the relevant object.
if (!theErr && !recStatus) {
    ::SRGetProperty(recResult, kSRPathFormat, &thePath, &theLen);
    theErr = ::SRGetIndexedItem(thePath, &theItem, 0);
    if (!theErr) {
        theLen = sizeof(theRefCon);
        ::SRGetProperty(theItem, kSRRefCon, &theRefCon, &theLen);
        ::SRReleaseObject(theItem);
    }
    // Release recognition result, since we're done with it.
    ::SRReleaseObject(recResult);
    ::SRReleaseObject(thePath);
}

// Send the reference constant up the chain of command.
LCommander::GetTarget()->ObeyCommand((MessageT)theRefCon, nil);

return theErr;
}
```

The interesting thing in this event handler is how utterly simple the important code is: all it does is extract the reference constant value of the recognized utterance and send that value up the PowerPlant chain of command. For example, if the recognized utterance is the word *new*, the reference constant is the value `cmd_New`, which is sent to a commander. In this case, the DocDemo application creates a new document. In effect, the CDocSpeech object does its work by calling code already in the DocDemo application.

THE LAST WORD

As you've seen, it's easy to add basic speech recognition for File menu commands to a PowerPlant application, largely because our custom speech object can simply issue the same commands that would be issued in response to a menu choice. You should now be able to add speech support for Edit menu commands and for any other menu commands supported by your application.

Only one method remains to discuss, the destructor for the CDocSpeech class. The destructor simply stops recognizing utterances and closes down the recognition system opened by the constructor, as shown in Listing 9.

Listing 9. Shutting down speech recognition

```
CDocSpeech::~CDocSpeech()
{
    ::SRStopListening(gRecognizer);
    ::SRReleaseObject(gRecognizer);
    ::SRReleaseObject(gDocuLM);
    ::SRReleaseObject(gRevert);
    ::SRCloseRecognitionSystem(gSystem);
}
```

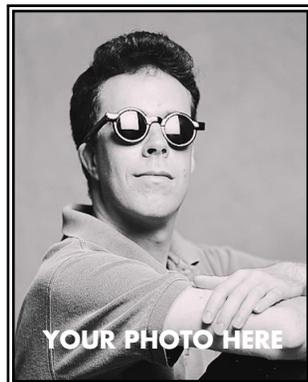
’Nuff said.

RELATED READING

- “The Speech Recognition Manager Revealed” by Matt Pallakoff and Arlo Reeves, in this issue of *develop*.
- “Speech Recognition Manager,” on this issue’s CD and on Apple’s speech technology Web site, <http://www.speech.apple.com>.
- *The PowerPlant Book* by Jim Trudeau, in *Inside PowerPlant for CW8* (Metrowerks, 1995).

Thanks to our technical reviewers Mike Dilts, Guillermo Ortiz, Matt Pallakoff, Arlo Reeves, and Brent Schorsch. •

Want to Show off your cool code?



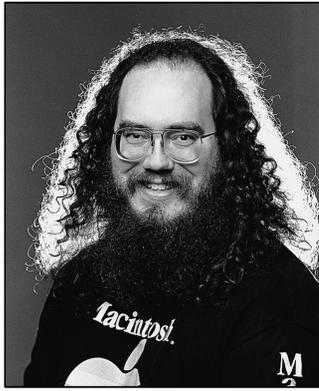
YOUR NAME HERE

Do you have code that solves a problem other Macintosh developers might be having? Why not show it off by writing about it in *develop*?

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

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

To receive our Author’s Guidelines, editorial schedule, and information about our incentive program, please send a message to develop@apple.com, or write Caroline Rose, Apple Computer, Inc., 1 Infinite Loop, Cupertino, CA 95014.



DAVE POLASCHEK

PRINT HINTS

The All-New LaserWriter Driver Version 8.4

By the time you read this, version 8.4 of the LaserWriter 8 printer driver will have shipped. This driver — LaserWriter version 8.4, for short — is not the same old LaserWriter driver: it has new features that developers have been asking for, sports a new user interface, and beats earlier versions of the driver in the quarter mile.

Here I'll outline some of the changes — a few minor, a few major — that you need to be aware of for compatibility reasons. Even if you don't want to take advantage of any of the great new features, you'll at least need to address compatibility issues if any of the changes cause problems with your application.

To help you implement the new features, this column is backed up with detailed documentation on this issue's CD.

THE EXTENDED PRINT RECORD

The 120-byte print record in the previous driver version doesn't have as many free bits available as some programmers would like. So to let you save all possible printing information about a document, Apple decided to allow for extensible print records.

If all you want to do is maintain compatibility with the new driver version, you shouldn't need to change your application at all. But if you want to take advantage of the extended print record — and implement attractive features such as access to a larger number of paper sizes, tray handling that works with the `PrJobMerge` function, and the ability to reliably save more user settings from the Page Setup dialog — you do need to make some minor changes, along the following lines:

- Because the extended print record can be any size larger than 120 bytes, your application must not make any assumptions about the record's size.
- Although the locations of fields that are currently defined within the `TPrint` structure won't change, you should use `PrGeneral` with the extended print record opcodes (described below) to access any additional fields.
- When using an extended print record, you'll need to call the `extendPrDefault` and `extendPrValidate` functions where you previously would have called the functions `PrintDefault` and `PrValidate`. (The new extend functions really just call `PrGeneral` with specific opcodes, but are more convenient to use than `PrGeneral` itself.) See "Extending the Print Record" on this issue's CD for more information on the extend functions and how they use the new `PrGeneral` opcodes.

Those who break the rules might need to make more changes. See the Print Hints column in *develop* Issue 26 ("The Top 10 Printing Crimes Revisited") for more information.

NEW PRGENERAL OPCODES

LaserWriter version 8.4 adds three new `PrGeneral` opcodes for dealing with the extended print record: `kExtendPrintRecOp` (which extends the print record), `kGetExtendedPrintRecOp`, and `kSetExtendedPrintRecOp`.

Table 1 gives a complete list of all the `PrGeneral` opcodes as of June 1996 (but be aware that printing in Mac OS 8 might not implement all of these). These opcodes are all planned to be supported by LaserWriter version 8.4, except for the ones that aren't used by LaserWriter 8 (as noted in the table). Refer to the article "Meet `PrGeneral`" in *develop* Issue 3 for more information about `PrGeneral`.

NEW PRINT DIALOGS

The print dialogs have been completely redesigned for LaserWriter version 8.4. Applications that use the approved method of extending the print dialogs will continue to function. But if your application uses a nonstandard method of extending the print dialogs, it's in trouble. The definitive source about how to extend a print dialog is `PDlog Expand`, available as sample code on this issue's CD and included with the Macintosh Technical Note "Print Dialogs: Adding Items" (PR 09).

DAVE POLASCHEK (dpolasch@apple.com) continues to be confused by California. There's nice weather when it isn't baseball season, the earth moves even when he's alone, and it's easier to

find good wine than good beer. Dave works in Developer Technical Support (DTS) at Apple. If you'd like more details, look at <http://www.best.com/~davep/>.

Table 1. The PrGeneral opcodes

Opcode	Operation
4	getRslDataOp
5	setRslOp
6	draftBitsOp
7	noDraftBitsOp
8	getRotnOp
9	NoGrayScl (not used by LaserWriter 8)
10	getPSInfoOp
11	PSIntentionsOp
12	enableColorMatchingOp
13	registerProfileOp (ColorSync 1 only; not used by LaserWriter 8)
14	PSAdobeOp
15	PSPrimaryPPDOp
16	kLoadCommProcsOp
17	kUnloadCommProcsOp
18	kExtendPrintRecOp (LaserWriter version 8.4 and later only)
19	kGetExtendedPrintRecOp (LaserWriter version 8.4 and later only)
20	kPrinterDirectOpCode (not used by any LaserWriter driver)
21	kSetExtendedPrintRecOp (LaserWriter version 8.4 and later only)

The new print dialogs have a pop-up menu that lets the user select between multiple panes of the dialog. In Figure 1, the General pane has been selected from the

pop-up menu. When an application adds items to the print dialog, they're added to a pane that has the name of the application. Because of this new multipane dialog, applications that extend the print dialogs in a nonstandard manner will cause many problems, such as dialog items appearing in the wrong locations, standard items being overwritten within the dialog, and standard items being drawn incorrectly.

Applications also shouldn't assume that the print dialog's foreground color is black or that the background color is white. Furthermore, when applications exit their CDEFs or user items, they should be careful to leave the foreground and background colors as they found them. Other items in the dialog rely on these colors, so if you change them the standard controls in the print dialog could take on unusual colors.

ONE-PASS PRINTING

With LaserWriter version 8.4, when background printing is disabled, printing is one-pass. This means that there are no longer any big spool files to fill up your hard drive, and the first printed page comes out of the printer more quickly (because it doesn't have to wait for the entire document to spool). The downside is that because the LaserWriter driver isn't making two passes over the data to be printed, it might not be able to perform the same optimizations on the PostScript™ code as when background printing is enabled. As a result, jobs printed with background printing disabled might print more slowly, and in a few cases the final quality could suffer.

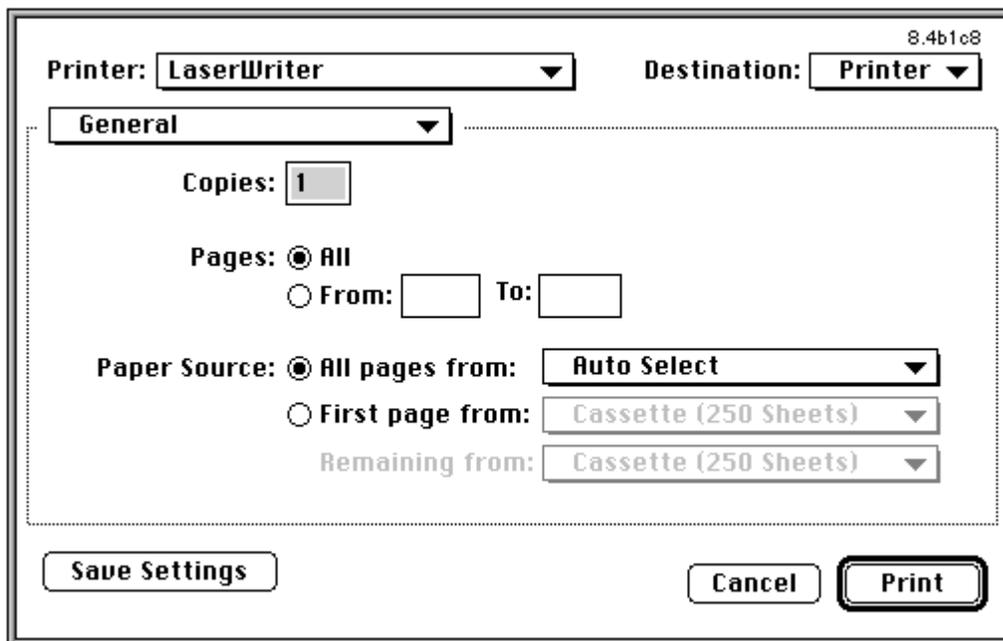


Figure 1. The new multipane print dialog

With the advent of one-pass printing, if your application has its own PostScript LaserPrep dictionary, it should use the PREC 103 mechanism for this dictionary. With this mechanism, the driver downloads to the printer the PostScript dictionary contained in the PREC 103 resource before it's needed by application-generated PostScript code. If the application doesn't do this and defines its own PostScript procedures at the page level, these procedures will be undefined as part of the one-pass font-handling mechanism and you'll get PostScript errors (mostly undefined operators, because the operators you defined aren't there).

PORTIONS OF THE CODE IN SHARED LIBRARIES

Some of the functionality of LaserWriter version 8.4 has been broken out into shared libraries, including the following:

- Converter library — generation of PostScript code
- PPD library — parsing of the PostScript printer description file
- Preferences and Collection libraries — storage and retrieval of preferences file data
- Downloader library — downloading of PostScript and EPS files to a printer
- PostScript Utilities library — PostScript utility functions
- Communications library — communications

In the future, Apple may provide APIs to these shared libraries for third parties.

CHANGES TO PARSING AND HANDLING OF PPD FILES

If you're a printer developer, you should know that the way PPD files are parsed and handled has changed in LaserWriter version 8.4. Previous versions of the driver would supply a "Printer's Default" choice so that the user could choose not to decide about a certain feature and accept the default setting of the printer. With version 8.4, the driver will no longer provide this option. If PPD creators want to continue to have a Printer's Default option for a user interface feature (called UIFeature in the PPD specification), they'll have to add it to the PPD file in the list of options for that feature.

Also, common features available through the PPD file will be added to the correct pane of the print dialog.

Features that aren't recognized or that are vendor-specific will be placed in their own pane. This can cause problems: if you use a nonstandard naming convention for a common feature, it will be placed with all other unknown features, and if you use a standard name for a nonstandard feature, it will probably end up in the wrong location.

One other change is that you can specify the graphic elements you'd like to use for UI features specified within the PPD file. See "LaserWriter 8.4 PPDs" on the CD for information about how to design your own pane for use with LaserWriter version 8.4. The latest Apple PPD files are the best examples of how to implement the new features.

NEW ERROR CODES

LaserWriter 8 introduced a number of new error codes, but they haven't been documented — until now, that is. See the unofficial documentation "LaserWriter 8 Errors" on the CD. Future versions of this document will be released as Technotes.

These error codes are provided for debugging purposes. Be aware that they may change in the future, so you probably don't want your application to depend on them.

WRAPPING IT UP

That's a quick rundown of the new features of the newest version of the LaserWriter driver. These features should make printing a better experience for the user, should give the developer more flexibility, and should require no changes to most applications. And to top it all off, they're *cool!*

RELATED READING

- "Print Hints: The Top Ten Printing Crimes Revisited" by Dave Polaschek, *develop* Issue 26.
- "Meet PrGeneral, the Trap that Makes the Most of the Printing Manager" by Pete "Luke" Alexander, *develop* Issue 3.
- Macintosh Technical Note "Print Dialogs: Adding Items" (PR 09).
- "Extending the Print Record," "LaserWriter 8.4 PPDs," and "LaserWriter 8 Errors," all on this issue's CD.

Thanks to Rich Blanchard, Paul Danbold, Ingrid Kelly, Dan Lipton, and Steve Simon for reviewing this column. Special thanks to Matt Deatherage. •

Working With OpenDoc Part Kinds

If you're ready to create your first full-featured OpenDoc part editor but have some questions about part kinds and how to work with them, you'll find the answers here. We explain how your choice of part kinds will affect whether users will be able to read your content with different part editors and even across different platforms. We also discuss some human interface principles and describe how to handle the most common user actions having to do with part kinds.



**TANTEK ÇELİK AND
DAVE CURBOW**

We imagine that every computer user on earth has had the experience of trying to open a document created by someone else but not being able to because the application it was created with is missing. In the context of OpenDoc, users can run into this when the part editor that created a part is missing. OpenDoc provides several ways to mitigate this “missing editor” problem. One way is for developers to create and freely distribute part viewers for all the kinds of parts that they support; a *part viewer* is a subset of its corresponding editor’s code that displays and prints a part’s contents but can’t be used to create or edit a part.

But suppose a user doesn’t have either an editor or a viewer for a particular part. That’s where part kinds come in. A *part kind* is a data format in which a part’s intrinsic content is stored, analogous to a file type in a traditional application. OpenDoc allows a part editor to support multiple part kinds — that is, to store the same content in multiple data formats — to increase the probability that a user will be able to see and copy the contents of a part. A user who doesn’t have the same part editor that created a part may have a different part editor that can read at least one of the data formats in which that part is stored. Alternatively, one or more of the data formats can perhaps be translated into a part kind for which the user has an editor or viewer.

What this means to you is that your choice of part kinds to support is a crucial step in developing a part editor. This article discusses how to choose which part kinds to support — standard (to Macintosh or across platforms) or proprietary — and whether

TANTEK ÇELİK (tantek@6prime.com) was until recently an OpenDoc technical lead at Apple. After shipping OpenDoc 1.0 and determining that it was good, he helped found 6prime corporation (<http://www.6prime.com>), an OpenDoc software consulting firm. TanteK prides himself on his multiple modes of alternative transportation, including inline skating, bicycling, and motorcycling. He likes to occasionally spend time writing applications in HyperCard, night skating in San Francisco, or turning a profit shorting Microsoft options. •

DAVE CURBOW (curbow@apple.com) was until recently the OpenDoc human interface lead. He has transferred to Apple Research Labs, where he’s working on something really neat — but he can’t tell you about it yet. When he isn’t toiling away at Apple or planning his next trip to Europe, Dave likes to work in his Japanese-style garden. All he needs now is a book that clearly explains how to twist black pines into interesting shapes. •

to support one or multiple part kinds. We also discuss how to decide which category your part kinds fit into, some human interface principles having to do with part kinds, and what to do in a few key situations in which user actions cause your editor to have to deal with part kinds.

If you're not already familiar with the OpenDoc human interface, you should first read "The OpenDoc User Experience" in *develop* Issue 22 to get up to speed. This article also requires you to know something about OpenDoc storage and how to use the `ODStorageUnit` class. "Getting Started With OpenDoc Storage" in *develop* Issue 24 is a good introduction; further details can be found in the *OpenDoc Programmer's Guide for the Mac OS* and its accompanying *OpenDoc Class Reference CD*.

CHOOSING YOUR PART KINDS AND CATEGORY

In developing your part editor, you first need to decide which part kind or kinds to support. This choice is worthy of careful consideration. The decision you make about whether to support standard vs. proprietary part kinds and how many part kinds to support will affect the number of users able to read your content across documents and platforms. We'll look at the tradeoffs here. We'll also give you the information you need in order to decide which category or categories your part kinds fit into.

STANDARD VS. PROPRIETARY PART KINDS

First, you need to decide whether to support standard or proprietary part kinds, or some combination of each. Standard part kinds are those data formats that, either through an official decree or by some de facto means, have become widely used and accepted. There are industry-standard part kinds, which are standard across more than one platform, and standard Macintosh part kinds.

Because new data formats are being created all the time, we can't give you a complete list, but here's a sample:

- industry standards — ASCII, TIFF, GIF, JPEG, MPEG
- Macintosh standards — TEXT, PICT, stxt, MOOV, 3DMF

Part kinds are usually specified as ISO strings (null-terminated ASCII strings using 7-bit characters) for manipulation by OpenDoc. As you can see from our list, standard Macintosh part kinds are actually today's standard Macintosh file types, except that instead of being file-type signatures they're ISO strings, which can be derived by using methods of the class `ODTranslation`. (See the Data Interchange recipes on the *OpenDoc Class Reference CD* for more details on how to properly support a standard Macintosh part kind based on a standard Macintosh file type.) Your part editor needs to provide user-readable names for part kinds in a name-mapping resource; more on this later.

The ASCII standard is actually pretty loosely defined. It doesn't specify whether you should use 7- or 8-bit encoding, nor does it say whether you should use LF, CR, or CRLF for line separators. In the near future, Unicode, which OpenDoc uses internally, is likely to become the standard. In the meantime, your part may need to be prepared to handle several variants on the ASCII standard without failure. •

If the part kind you choose to support is an industry standard, users will benefit because they'll be more likely to avoid the missing editor problem mentioned earlier. Furthermore, supporting standard part kinds enables your part editor to support more of the content that's already out there. Let's face it — data formats don't live forever, but the standard ones have a much better chance of being long-lived than any proprietary kinds you create.

On the other hand, if there's no standard for the content your part editor creates, or if the standard won't suffice to capture the functionality your part editor offers, you'll need to create a proprietary part kind. You must weigh the advantages of using a proprietary part kind against the disadvantage of users possibly not being able to read your part's content.

In any case, don't redefine an existing standard. For example, the TEXT part kind should be used only for plain text, not for some data format that uses text as part of its definition, such as PostScript, HTML, or BinHex. These data formats should be part kinds in their own right. Otherwise, there will be confusion when OpenDoc needs to find a substitute part editor for a part that claims to be TEXT but is in fact another kind such as HTML. The user won't be happy with the result.

If you decide to use an industry-standard part kind, the Bento container suite (part of the storage system in OpenDoc 1.0) can help you solve internal byte-ordering problems and ensure that a document written on any OpenDoc platform can be read and written on any other OpenDoc platform. However, your part editor is responsible for proper byte ordering of the values in the content property of your storage unit. (Data formats typically specify byte ordering, so OpenDoc stays out of your way here.) The Standard Type I/O utilities (see the file StdTypIO.h and the functions declared there) solve the byte-ordering problem for a variety of simple data formats. These utilities can be used in combination to build up more complex data formats.

SUPPORTING MULTIPLE PART KINDS

As we've said, your editor can support one or more part kinds. If it supports more than one part kind, one of these will be the *preferred kind*. Users implicitly indicate the preferred kind when they choose a stationery pad or cut and paste content. They can also change the preferred kind in the Part Info dialog if they desire; more on this later.

Supporting multiple part kinds increases the probability that other users can see the contents of a part created with your editor, even if they don't have your part editor (see "Editor Substitution Explained" for why this is so). Your choice of part kinds to support comes into play both when the user saves a document with parts created by your editor and when the user transfers data with a paste or drop operation.

When deciding how many part kinds to support when your editor is saving its parts of a document, you'll want to consider the tradeoff between portability and the space required to store your part as multiple kinds. The most transportable part kind (that is, the standard one) may not be the most compact or the one that will represent the underlying contents with the greatest fidelity. Typically, you'll want to store only the one preferred part kind, or the preferred kind and one standard part kind. If there isn't a standard kind that's roughly equivalent to your preferred kind, consider also storing a TEXT or PICT representation, simply to maximize the chances that the user will be able to see something for your part. For example, if your part's preferred kind is 3DMF, there isn't an equivalent standard kind, so you should also store a PICT representation. You might want to present the user with a Settings or Preferences dialog giving a choice of part kinds to store in addition to the preferred kind. See pages 476 and 479 of the *OpenDoc Programmer's Guide* for implementation details.

When your editor is providing data for a data transfer operation (such as a copy to the Clipboard), you may want to write out a greater number of standard part kinds than during a save operation. This is because during data transfer it's more likely that the user is trying to move content to a different editor or application. Providing standard part kinds in this situation is therefore even more important. On the other hand, remember that the user can use the Paste As command to get more options, including translation, so you needn't go overboard in supporting lots of kinds.

EDITOR SUBSTITUTION EXPLAINED

When a user tries to open a document or edit a part and the editor that created it is missing, OpenDoc searches for a substitute. This occurs as part of OpenDoc's *binding* process — the process of assigning the correct part editor to a given part. When a document is opened, the OpenDoc binding subsystem binds editors to all parts that need to be displayed. During execution, OpenDoc binds editors to part data when a part is read in or when its editor is explicitly changed.

Let's look at a simplified example of editor substitution. Suppose we've created a text editor named SurfWriter that stores its content in three formats: a proprietary part kind (SurfWriter Text) and two standard part kinds (RTF and TEXT). And suppose that SurfWriter Text is the preferred kind. When OpenDoc tries to display the part, its binding subsystem looks first for SurfWriter — the last editor that was used. If that isn't found, the binding subsystem looks for an editor that can read SurfWriter Text — the preferred kind. If that can't be found, it looks for one that can read RTF or TEXT. Thus, storing multiple part kinds increases the probability that users will be able to read your content with different part editors and across different platforms.

Now let's look at editor substitution in a little more detail. When attempting to find an editor to bind to a part, OpenDoc looks first for the editor that last edited the part, specified in the `kODPropPreferredEditor` property in the

part's storage unit. If this editor isn't present on the user's system, the binding subsystem examines each of the part kinds in the stored part and the list of kinds supported by the editor or editors installed on the user's system, looking for a match. For each supported kind, there's a default editor. The user can inspect and modify the list of default editors in the Editor Setup control panel (Figure 1).

During the matching process, the binding subsystem looks first for the default editor for the preferred kind. If this editor isn't present, it looks for the default editor for the preferred kind's category, and finally for any editor that can read the preferred kind. If such an editor can't be found, the binding subsystem repeats the whole process for each of the remaining part kinds in the part, from highest fidelity to lowest.

If no editor for any of the part kinds is installed on the user's machine, the part remains unviewable and uneditable. But OpenDoc still binds an editor to the part — the "editor of last resort." This editor is always available and represents the part as an icon within the document, so that there's never a blank spot in the document where a part can't be displayed. The user can examine the part's kind in the Part Info dialog, which gives a clue as to which editor or viewer should be installed, although if there's no editor for the part, there's probably no user string for the preferred kind. The user can also decide to translate the part to another part kind.

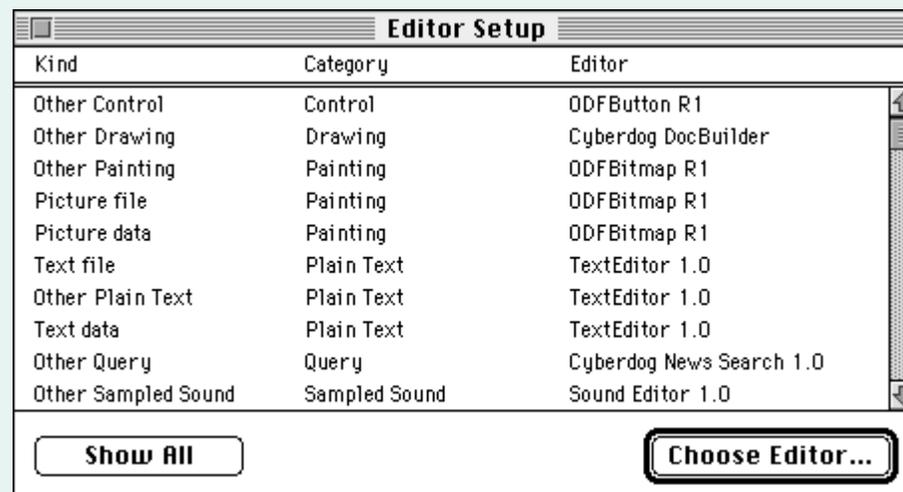


Figure 1. The Editor Setup control panel

CATEGORY CONSIDERATIONS

After you've chosen the part kinds to support, you need to determine which category or categories these belong to. A *part category* is a set of part kinds that are conceptually similar. You might think of it as a generic term for several “brand name” variants. For example, the `kODCategoryStyledText` category might include the part kinds `SurfWriter Text 3.0`, `SurfWriter Text 2.0`, and others.

OpenDoc looks at a part's category to decide which part editors or part viewers can be substituted if an editor is missing and whether to merge or embed data when content is copied from one part into another. Categories are specified by your editor in a name-mapping resource and can't be changed by the user.

Categories for existing part kinds have already been determined and should be adhered to; this set of categories is broad enough to include most new part kinds as well. A list of the predefined categories is given in Table 1. This list can be found in the *OpenDoc Programmer's Guide* on pages 477–478, but note that a new category has been added since the publication of the book: `kODCategoryArchive`.

The majority of the entries in the list (such as `kODCategoryPlainText` and `kODCategoryStyledText`) are self-explanatory, but a few need some clarification.

Table 1. Predefined part categories

Part category	Explanation
<code>kODCategoryPlainText</code>	Plain ASCII text
<code>kODCategoryStyledText</code>	Styled text
<code>kODCategoryDrawing</code>	Object-based graphics
<code>kODCategory3DGraphic</code>	3D object-based graphics
<code>kODCategoryPainting</code>	Pixel-based graphics
<code>kODCategoryMovie</code>	Movies or animations
<code>kODCategorySampledSound</code>	Simple sampled sounds
<code>kODCategoryStructuredSound</code>	Sampled sounds with additional information
<code>kODCategoryChart</code>	Chart data
<code>kODCategoryFormula</code>	Formula or equation data
<code>kODCategorySpreadsheet</code>	Spreadsheet data
<code>kODCategoryTable</code>	Tabular data
<code>kODCategoryDatabase</code>	Database information
<code>kODCategoryQuery</code>	Stored database queries
<code>kODCategoryConnection</code>	Network-connection information
<code>kODCategoryScript</code>	User scripts
<code>kODCategoryOutline</code>	Outlines created by an outliner program
<code>kODCategoryPageLayout</code>	Page layouts
<code>kODCategoryPresentation</code>	Slide shows or other presentations
<code>kODCategoryCalendar</code>	Calendar data
<code>kODCategoryForm</code>	Forms created by a forms generator
<code>kODCategoryExecutable</code>	Stored executable code
<code>kODCategoryCompressed</code>	Compressed data
<code>kODCategoryControlPanel</code>	Data stored by a control panel
<code>kODCategoryControl</code>	Data stored by a control, such as a button
<code>kODCategoryPersonalInfo</code>	Data stored by a personal information manager
<code>kODCategorySpace</code>	Stored server, disk, or subdirectory data
<code>kODCategoryProject</code>	Project-management data
<code>kODCategorySignature</code>	Digital signatures
<code>kODCategoryKey</code>	Passwords or keys
<code>kODCategoryUtility</code>	Data stored by a utility function
<code>kODCategoryMailingLabel</code>	Mailing labels
<code>kODCategoryLocator</code>	Locators or addresses, such as URLs
<code>kODCategoryPrinter</code>	Stored printer data
<code>kODCategoryTime</code>	Stored clock data
<code>kODCategoryArchive</code>	Archive or partial archive data such as TAR

- `kODCategoryOutline` — Use this category when your part's content has some hierarchy — that is, when the content is assigned to different nested levels. For example, the Cyberdog Notebook, an excerpt from which is shown in Figure 2, presents a collection of URLs in hierarchical form and thus is an outline.
- `kODCategorySpace` — Use this category when the content has no intrinsic order, as in the case of server, disk, or subdirectory (folder) data. For example, in a pre-System 7 Finder folder, the order of the contents depends entirely on the settings of the View menu. A part with content like this would belong to this category.
- `kODCategoryPersonalInfo` — Use this category for the various kinds of information represented in personal information management (PIM) applications.
- `kODCategoryPageLayout` — Use this instead of `kODCategoryDrawing` when the part contains only embedded content. In contrast, the category `kODCategoryDrawing` is for a drawing that has intrinsic content, such as circles and rectangles.

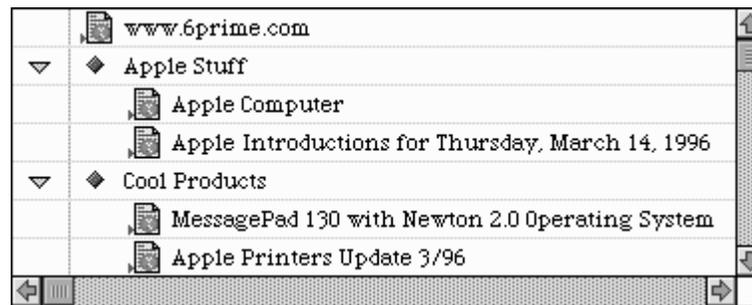


Figure 2. Example of an outline from the Cyberdog Notebook

Some of the categories seem as though they could be subsets of other categories — for instance, `kODCategoryPlainText` could be a subset of `kODCategoryStyledText`, and `kODCategory3DGraphic` could be a subset of `kODCategoryDrawing`. But categories aren't hierarchical — that is, one category can't include others.

When you're considering which category or categories your part kinds should belong to, ask yourself the following question for each of the categories: If users pasted my kind of data into a part belonging to this category, would they expect the content to be merged, or embedded as a separate part? If they would expect the content to be merged, that's a category your part kind should belong to. (Note that whether a part kind supports embedding doesn't affect which category it's in.)

For example, if users pasted a slide (a part belonging to the `kODCategoryPresentation` category) into some text (a part belonging to the `kODCategoryStyledText` category), they would expect the slide to be embedded within the text because the operations on slides and text are very different. But if they pasted one slide into another slide, they would expect the contents of the first slide to be merged into the destination slide; thus, the two parts should belong to the same category.

Consider another example. If users pasted a picture from a Web page into a part belonging to the category `kODCategoryPainting`, they would probably expect it to be merged. But if they pasted the picture into a part belonging to the category `kODCategoryDrawing` or `kODCategory3DGraphic`, they would probably expect it

to be embedded, because the operations available in a painting part are usually very different from those in a drawing part. Thus, the picture should belong to the category `kODCategoryPainting`.

You need to choose one or more categories for each of the part kinds that your part editor supports. A part kind can be in multiple categories; for example, a part that can shift its view from table to chart should have a preferred kind that's a member of both categories. The same category can be specified for a part kind that represents a single object and a part kind that represents a collection of those objects; for example, you can specify `kODCategoryDatabase` for a part kind that represents a single database record and for a part kind that represents a collection of such records.

As mentioned earlier, when your part editor provides content to the Clipboard or a drag and drop object, you may want to write out a greater number of standard part kinds than during a save operation, to increase the probability of being able to interchange data with other parts. In fact, it will help if you support kinds in more than one category. Here's an example: Suppose a user copies some spreadsheet cells and pastes them into a chart. Because the operations on cells and charts are different, the user will expect the spreadsheet cells to be embedded. However, if the spreadsheet provides its copied data in a format that the chart is prepared to merge, the user gets a higher level of interoperability. If the spreadsheet and the chart both support kinds that are in the `kODCategoryPlainText` category, for instance, the chart can take the content of the spreadsheet and chart it instead of embedding the spreadsheet.

Here are some more examples of part kinds and the categories they fit into:

- **PostScript** — This page description language is used to define images in a structured fashion. The PostScript format might fit into either `kODCategoryPageLayout` or `kODCategoryDrawing`. We recommend `kODCategoryDrawing` because a part in PostScript format has intrinsic content like a drawing, such as arcs and clip shapes.
- **HTML** — Hypertext Markup Language (HTML) is similar to PostScript in that it defines a page layout. However, when HTML is displayed it typically looks more like styled text than like a drawing. Therefore, the appropriate category for HTML is `kODCategoryStyledText`.
- **BinHex** — Like many other formats that claim to be text but are only making use of text to define some richer format, BinHex is actually an archive format. Hence BinHex belongs in `kODCategoryArchive`.
- **URL** — Another kind that uses text to define some richer format, a URL should be in `kODCategoryLocator`.

If your part kinds don't appear to fit in any of the predefined categories, you can request a new category. The list of predefined categories is maintained by CI Labs, a consortium that coordinates cross-platform OpenDoc development. See the CI Labs Web page <http://www.cilabs.org/categories> for instructions on how to request a new category.

RESOURCES REQUIRED

Both part kinds and part categories are assigned in your part editor's name-mapping ('nmap') resources. You can learn how to construct these resources by looking at the Dynamic Binding recipes on the *OpenDoc Class Reference* CD. These resources are required:

- **EditorKinds** — lists every part kind your editor supports, *except* standard Macintosh part kinds

- `EditorPlatformKind` — lists the standard Macintosh part kinds your editor supports
- `KindCategories` — lists the category or categories your part kinds belong to
- `KindUserString` — lists the part kind user strings

If you request a new category and CI Labs approves your request, you'll also need a `CategoryUserString` resource listing your category user strings. OpenDoc already contains user strings for predefined categories.

Listing 1 shows an `EditorPlatformKind` resource indicating that your editor supports TEXT files and TEXT scrap data. Listing 2 demonstrates how a part editor would declare two part kinds that are in the same category in a `KindCategories` resource.

At run time, if you need to convert a Mac OS file type such as 'TEXT' to an ISO type, first get the translation object from the session:

```
ODTranslation* translation = session->GetTranslation(ev);
```

Then call the translation object to convert the Mac OS file type, or what we call the *platform kind* (a platform-neutral term), to an ISO type:

```
ODValueType valueType =
    translation->GetISOTypeFromPlatformType('TEXT', kODPlatformFileType);
```

Listing 1. An example `EditorPlatformKind` resource

```
resource kODNameMappings (kPlatformEditorKindMapId)
{
    kODEditorPlatformKind,
    { /* array KeyList: 1 element */
        /* [1] */
        kYourEditorID,
        kODIsPltfrmTypeSpac
        { /* array PltfrmTypeSpacList: 2 elements */
            {
                /* [1] */
                kODPlatformFileType,
                'TEXT',
                smRoman,
                langEnglish,
                "Plain Text",
                kODCategoryPlainText,
                /* [2] */
                kODPlatformDataType,
                'TEXT',
                smRoman,
                langEnglish,
                "Plain Text",
                kODCategoryPlainText,
            }
        }
    }
};
```

Listing 2. An example KindCategories resource

```
resource kODNameMappings (kKindCategoryMapId)
{
  kODKind,
  { /* array kinds: 2 elements */
    /* [1] */
    kStyledTextKind1,
    kODIsAnISOStringList
    {
      { /* array categories: 1 element */
        /* [1] */
        kODCategoryStyledText
      }
    },
    /* [2] */
    kStyledTextKind2,
    kODIsAnISOStringList
    {
      { /* array categories: 1 element */
        /* [1] */
        kODCategoryStyledText
      }
    }
  }
};
```

You'll find `kODPlatformFileType` defined in `StdDefs.xh`, `ODTranslation` defined in `Translt.xh`, and `ODSession` defined in `ODSessn.xh`. Use `kODPlatformDataType` instead of `kODPlatformFileType` if you're converting a *scrap* type from the Clipboard as opposed to a *file* type from the file system.

SOME HUMAN INTERFACE PRINCIPLES

There are some important human interface principles regarding part kinds that you should incorporate in the design of your part editor. They boil down to maintaining the fidelity of parts as they pass through various operations.

One key principle of the user model is that editors shouldn't change the part kind of content without warning, because the translation may cause information to be lost. Only the user should be able to change the preferred kind of a part, and then only through an explicit action. This supports the concept that content copied into or out of an OpenDoc document should retain its fidelity. For example, when the user drags a drawing document from the desktop into an OpenDoc document and then back to the desktop, the initial document and the final document should be identical, as far as the user is concerned. The final document should have the same part kind as the original document, unless the user elects to change the part kind. (See the recipe for promising a non-OpenDoc file on the *OpenDoc Class Reference* CD for more details.)

Users can change the preferred kind of a part with the Part Info (or Document Info) command in the Edit menu. This command brings up a dialog like the one shown in Figure 3. A pop-up menu offers a list of part kinds supported by the current editor, plus the possibility of translating to a different format with the "Translate to" command.

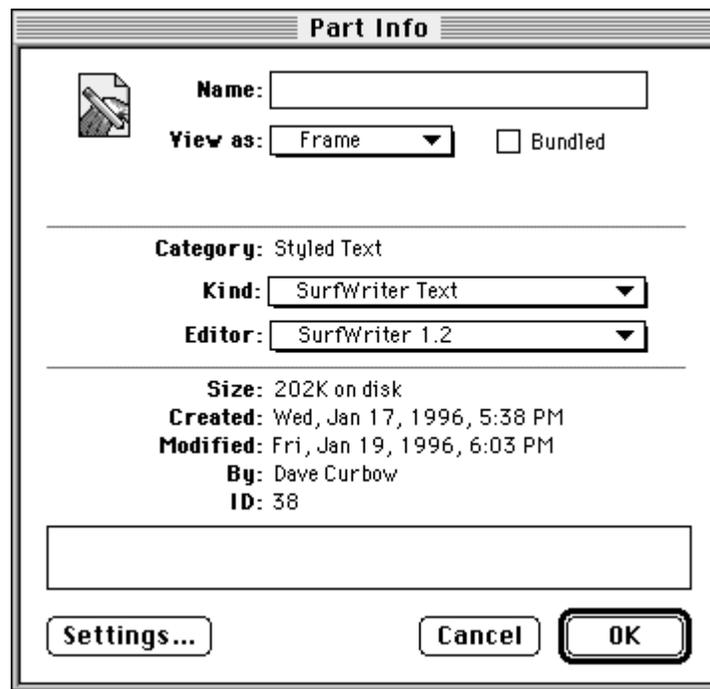


Figure 3. The Part Info dialog

When the user wants to save a document, your editor should write it out in the format of the preferred kind. The highest-fidelity kind that your editor writes should be the preferred kind. Don't change the preferred kind, because that would be implicit translation, or translating formats behind the user's back — not a good idea, although some applications behave this way today. Perhaps you've seen this: the application claims to read or write a particular data format, but when a document of that file type is opened with that application and then saved, the application converts the document to its own proprietary format. Users are left wondering why their documents can't stick with the format they were created with.

To maximize interchange between OpenDoc, traditional applications, and system software, OpenDoc does *not* arbitrarily promote platform kinds (which, remember, is our platform-neutral term for Mac OS file types) to OpenDoc part kinds. •

In today's applications, this unexpected format change is also often associated with the creation of a new document named "Untitled *x*" or "FooDocument - converted." In OpenDoc, parts don't have control over the name of the document, so this errant behavior is prevented. The name of the document, just like the preferred kind of a part, should be considered a user setting. Editors shouldn't tamper with user settings.

There *are* situations where it's appropriate for the editor to query the user about changing a part kind. If the user tries some operation, or tries to add some content, that's not supported in the current kind but is supported in another kind that the editor understands, it's appropriate to suggest changing to the kind with more functionality.

As an example of the first situation, suppose the user is editing a plain-text document with an editor that supports styled text. If the user selects some text and tries to change it to bold, the part editor must allow this change but should warn the user that the operation will require a change in the part kind — and the user must be allowed to veto this operation before it's done. In this situation the part editor should display an alert like the one shown in Figure 4.

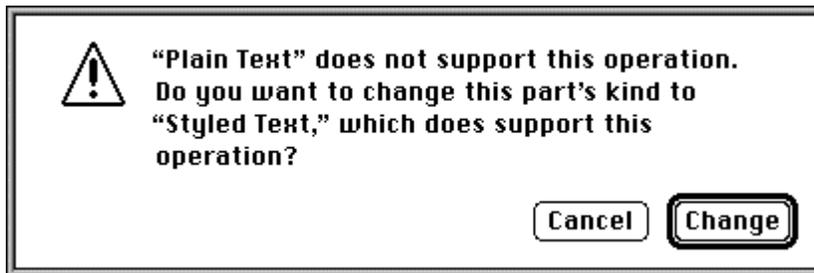


Figure 4. Warning the user that an operation requires the part kind to be changed

As an example of the second situation, suppose the user now pastes some text that includes a page break and an indentation, which isn't supported in styled text but is supported in a proprietary format the part editor uses. The part editor should allow this change but present an alert (see Figure 5) and let the user veto the change.

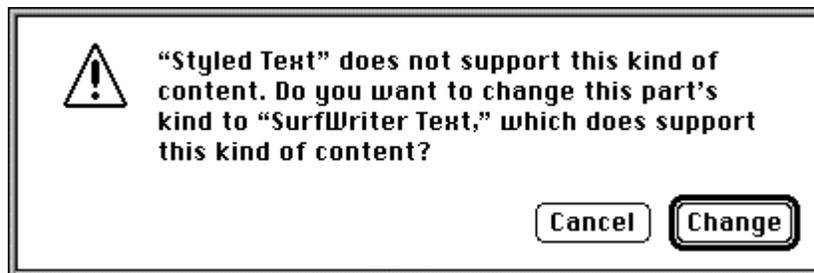


Figure 5. Warning the user that adding content requires the part kind to be changed

HANDLING USER ACTIONS

A number of user actions require your editor to deal with part kinds and categories, though in most cases this interaction is transparent to the user. For example, when a user pastes content into a part, the editor of the part where the content is about to be pasted examines the part kinds and categories of the content being pasted. The editor decides which, if any, of the multiple part kinds available will be pasted. In this case, as in many others, the user doesn't realize what's going on behind the scenes with part kinds and categories.

We'll discuss in detail what your editor should do with part kinds and categories in response to each of the following user actions:

- creating a document
- opening a document
- saving a document
- transferring data
- changing the preferred kind
- translating or converting a part

CREATING A DOCUMENT

To create a document, the user double-clicks on a stationery pad that you supply with your part editor. You must provide at least one stationery pad for each part category that your editor supports. For example, if your editor supports the "styled text"

category and the SurfWriter Text, AcmeWriter Text, and RTF part kinds, you must supply (and your product's installer must install on the user's system) a stationery pad for at least one of these part kinds. Typically, you'll install a stationery pad for the highest-fidelity part kind that you support.

You can optionally provide more than one stationery pad. When users decide to double-click on one stationery pad instead of another, they've made an explicit decision about the preferred kind of the document they want to be created.

Rules and conventions for installing part editors and stationery pads can be found in the *OpenDoc Programmer's Guide*, Appendix C, "Installing OpenDoc Software and Parts."

OPENING A DOCUMENT

Whenever a user opens a document containing one of your parts, your part must be reconstituted from external storage by your `InitPartFromStorage` method, described in detail in the article "Getting Started With OpenDoc Storage" in *develop* Issue 24. Your editor needs to find out the preferred kind and read in the content data accordingly.

If your editor supports any platform kinds (Mac OS file types), you should first check for the `HFSFlavor` value type in the content property (`kODPropContents`) of the part's storage unit. If it's there, you've been bound to an empty storage unit that's pointing to a file that you should use to internalize from. This binding can happen in one of two ways: the user may have dragged and dropped a traditional Macintosh file onto an OpenDoc document and your part editor was bound to the drop, or the user may have opened a traditional Macintosh file with the OpenDoc launcher application. For detailed information on how to make this work, see the Drag and Drop Recipes on the *OpenDoc Class Reference* CD, specifically the section "Incorporating Data From a Non-OpenDoc Document." Also, see the section "Accepting Non-OpenDoc Data" on page 371 of the *OpenDoc Programmer's Guide*.

If your editor doesn't support any platform kinds, follow these steps:

1. Get the preferred kind — that is, read the value from the `kODPropPreferredKind` property of the part's storage unit. If this property doesn't exist, the editor can assume that the preferred kind of the part is the value type of the first value in the content property. Keep the preferred kind in a field, as shown in the following example using utility functions from `StdTypIO` and `TempObj`:

```
#include <StdTypIO.h>
#include <TempObj.h>
...
// The following code belongs in your InitPartFromStorage method.
ODStorageUnit* su = self->GetStorageUnit(ev);
TempISOStr preferredKind = ODGetISOStrProp(ev, su, kODPropPreferredKind,
    kODISOStr, kODNULL);
if (preferredKind == kODNULL) {
    su->Focus(ev, kODPropContents, kODPosUndefined, kODNULL, 1,
        kODPosUndefined);
    preferredKind = su->GetType(ev);
}
```

2. Focus your part's storage unit to the value of the content property whose value type is the preferred kind.

```
self->GetStorageUnit(ev)->Focus(ev, kODPropContents, kODPosUndefined,
    preferredKind, 0, kODPosUndefined);
```

-
3. Read the contents of that value and create the in-memory data structures necessary to represent that content. Use the `ODStorageUnit` method `GetValue` to accomplish this step.

Note that it's possible for your editor to be bound to a part that previously had a different editor, as described earlier in “Editor Substitution Explained.” In this case, the OpenDoc binding subsystem will automatically notify the user. If your editor doesn't support the preferred kind, use the highest-fidelity kind in the content property that your editor does support as the de facto preferred kind. *Do not* update the preferred kind property until `Externalize` or `ChangeKind` is called on your part.

SAVING A DOCUMENT

Whenever a user saves the document, your part must be written out to the storage unit, or externalized, by your `Externalize` method, described in detail in the article “Getting Started With OpenDoc Storage” in *develop* Issue 24. Your editor should write out the preferred kind, at a minimum; you may also decide to write out one or more alternate part kinds, as discussed earlier under “Supporting Multiple Part Kinds.”

The first two steps that are required have to do with preparing the storage unit for clean externalization from your part editor, also known as “prepping the storage unit.” You should only have to do this the first time `Externalize` is called on your part.

1. Clean up the storage unit by removing any values that you won't be updating. This means calling the `Remove` method for any values in the content property that have value types (part kinds) that your editor doesn't support or that your editor won't externalize.
2. Add values if necessary. Use `AddValue` to create or recreate the value types that you want to externalize in proper fidelity order (from highest fidelity to lowest fidelity). Fidelity ordering is important because OpenDoc looks at it to determine which editor would best edit any given part.
3. Externalize your content in the format of the preferred kind that your editor kept track of in a local field.
4. Optionally, write out alternative part kinds. As mentioned earlier, the typical part editor should by default write out only the one preferred kind, or the preferred kind and one standard part kind. If you present users with a Settings or Preferences dialog to indicate a set of alternative part kinds to store, write out the alternative kinds indicated there.

Your `Externalize` method may be called at times other than when the user saves a document. For example, depending on the Save model of the current document and the idle-time optimizations that may or may not be present, your part may be told to externalize only when the user saves a document or as often as every minute. Therefore, your editor shouldn't have preconceived notions about why it's asked to write out your part. As an optimization, your editor should keep an `fDirty` flag that's set whenever the user changes the part's content and cleared whenever externalization is completed. If your `fDirty` flag is clear, your `Externalize` method should be a no-op.

TRANSFERRING DATA

Whenever the user transfers data with Cut, Copy, Paste, Paste As, or drag and drop, your `CloneInto` method is called. See the section “The `CloneInto` Method of Your Part Editor” on pages 327–329 of the *OpenDoc Programmer's Guide* for the precise details of implementing the `CloneInto` method.

For the purposes of multiple part kind support, however, your editor should do the following:

1. Write the same part kinds you would if you were externalizing, plus any standard part kinds you support. As explained earlier, it's more important to write out standard part kinds during CloneInto than Externalize because the user is more likely to be trying to move content to a different editor or application.
2. Call SetPromiseValue for each part kind if you're using promises (explained in the *OpenDoc Programmer's Guide*).

If your part editor is a container, it's important for it to treat pasted content appropriately. When your container receives a Paste command or is the destination of a drag and drop, it should check the preferred kind of the incoming content to decide whether to merge or embed that content. If the category of the preferred kind of the incoming content is the same as the category of your content's kind, merge the incoming content; otherwise, embed the incoming content into a new part.

As mentioned earlier, it's possible for kinds to belong to more than one category. If the incoming content's kind or your content's kind belongs to multiple categories, or if both do, as long as they share at least one category they can be said to be of the same category. If the incoming content isn't an OpenDoc part, simply use the data type that's closest to your own content kind as the de facto preferred kind for the incoming content.

If the user drops a part onto your part that you determine should be merged, and you find there's no content when you try to merge it, the operation will appear to be a no-op, which is very confusing to the user. You may want to actually embed an empty part in this case rather than merging nothing, so that the user at least receives some feedback.

You also should be aware of a concern about format fidelity that arises if the user attempts a paste or drop operation with your editor that involves content with other content embedded. Some data or formatting may be lost if one or more of the part kinds supported by your part editor is of lesser fidelity and can't handle embedded content, and at the time of the paste or drop the destination part editor can work only with the lower-fidelity kind. In this case, the destination part editor can't know that it's losing the embedded content.

What can you do to minimize these cases, or at least make them easier on the user? We *strongly* recommend that your part editor support embedding. If it doesn't, it shouldn't claim to support a kind that includes embedding. For example, the part editor that's the destination for a paste or drop shouldn't strip embedded content or links out of the data format. If your editor can't preserve the fidelity of the paste or drop, it must choose a lower-fidelity part kind; if there are no other kinds present that your editor supports, it shouldn't allow the paste or accept the drop. The only exception to this is when a plain-text editor receives a paste of styled text; in this case, it can use only the text and ignore the style information. Because text is so ubiquitous, it's handled differently from other kinds of content.

If your part editor supports embedding, it should allow the user to embed any content that can't be merged; it shouldn't restrict the kinds that can be embedded. •

Remember that if your part editor supports data interchange, it must completely support Undo, so that if data or formatting is lost in a transfer operation, the user can undo and recover what was lost. Although most of today's applications don't alert

the user when data or formatting is lost, users seem to recognize with ease when they've experienced such a loss and need to choose Undo to recover. With the multiple-level Undo support in OpenDoc, recovering from a loss of data or formatting is much easier.

CHANGING THE PREFERRED KIND

Whenever the user changes the preferred kind of a part, your ChangeKind method is called. This is usually done from the Part Info (or Document Info) dialog shown earlier, but you shouldn't assume that that will be the only user interface that can cause this method to be called.

Your editor should do the following:

1. Externalize the part in the new preferred data format. Make sure that the fidelity order of the values in your content property is maintained by creating the values for the supported part kinds in the right order. You may need to prep your storage unit again and recreate the values to ensure that they're in the proper fidelity order. It's up to your part editor whether you keep the previous preferred kind or not.
2. Write the new preferred kind into the preferred kind property of the part, as shown in the following example using utility functions from StdTypIO and TempObj:

```
#include <StdTypIO.h>
#include <TempObj.h>
...
// The following code belongs in your ChangeKind method; the kind
// that the user selected is passed in the changeKind parameter.
ODStorageUnit* su = self->GetStorageUnit(ev);
ODSetISOStrProp(ev, su, kODPropPreferredKind, kODISOStr, changeKind);
```

TRANSLATING OR CONVERTING A PART

The user can force translation of a part with the Part Info (or Document Info) command in the Edit menu, which brings up a dialog like the one shown earlier in Figure 3. The part kind pop-up menu in the dialog, in addition to listing part kinds supported by the current editor, offers the possibility of choosing "Translate to" and then choosing a part kind from the Translate To dialog. The part kind pop-up menu shown in Figure 6 illustrates a number of different ways that picture data can be stored on the Macintosh, including standard MIME types, standard Macintosh file types, and standard Macintosh data types. Of course, most part editors won't support this many different kinds.

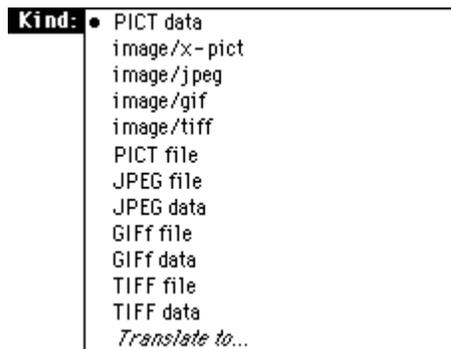


Figure 6. The part kind pop-up menu

There are also data interchange utilities, such as converters and grinders, that convert parts or entire documents to different part kinds. This operation involves asking each part in the original document to externalize itself in a set of standard part kinds. The user may initiate this action by dropping a document on a converter or grinder icon (like the one shown in Figure 7) on the desktop. Your `ExternalizeKinds` method is called in response.



Figure 7. A converter icon

`ExternalizeKinds` is passed a list of kinds to externalize. Your part editor doesn't need to write other values it might ordinarily write in addition to the preferred kind. Your editor should do the following in its `ExternalizeKinds` method:

1. Externalize the set of part kinds specified. Make sure that the fidelity order of the values in your content property is maintained by creating the values for these part kinds in the right order. You may need to prep your storage unit again and recreate the values to ensure that they're in the proper fidelity order. Be sure to write out these kinds in addition to the preferred kind, not *instead of* the preferred kind.
2. Ignore any part kinds in the set that you don't support.

PARTING WORDS

By now you should have a good idea of all the ramifications of choosing the part kinds to support with your part editor. We hope that by spelling out what the tradeoffs are and suggesting how your part editor should respond to various user actions related to part kinds, we're helping to promote a consistent approach to working with part kinds. This is bound to result in more portable parts and happier users.

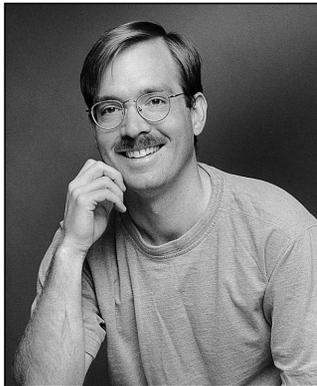
RELATED READING

- *OpenDoc Programmer's Guide for the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995). This book is accompanied by the *OpenDoc Class Reference* CD and includes the OpenDoc human interface guidelines.
- *OpenDoc Cookbook for the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995).
- "The OpenDoc User Experience" by Dave Curbow and Elizabeth Dykstra-Erickson, *develop* Issue 22.
- "Getting Started With OpenDoc Storage" by Vincent Lo, *develop* Issue 24.
- *Byte Guide to OpenDoc* by Andrew MacBride and Joshua Susser (Osborne McGraw-Hill, 1996), <http://www.splash.net/books/opendoc>.
- The OpenDoc World Wide Web pages. Apple's page is at <http://www.opendoc.apple.com>, and the CI Labs page is at <http://www.cilabs.org/opendoc.html>. These include updated recipes, technical notes, and the like.

Thanks to our technical reviewers Craig Carper, Elizabeth Dykstra-Erickson, and Kurt Piersol. •

Using Apple Guide 2.1 With OpenDoc

You've helped create an Apple Guide guide for your standalone application. Now your company is writing an OpenDoc part editor, but it's not obvious how Apple Guide can be used in the context of parts and compound documents. The answer is Apple Guide 2.1, which extends the original version to allow easy use of Apple Guide features, including Help menu management, coachmarks, and context checks, in the world of OpenDoc. This article introduces the new features of Apple Guide 2.1 and explains in detail how to use them with OpenDoc.



PETER COMMONS

The world of OpenDoc isn't like the world of the standalone "behemoth" application — a fact that hasn't escaped the notice of developers who want to provide online help. An application developer can easily provide complete online help since the application is a single self-contained unit that will run in its own window. On the other hand, the developer of an OpenDoc part editor never knows what other part editors will be running along with it during an OpenDoc session. Users can have any number of part editors running in any number of windows at the same time. All the editors might belong to a package written by one company, or, more likely, it might be a conglomeration written by a number of different companies. Under these circumstances, providing help gets a little more complicated.

Since Apple Guide is such a powerful help system, wouldn't it be nice if it could be applied to the new world of OpenDoc? Enter Apple Guide 2.1 (released right on the heels of Cyberdog, Apple's new integrated suite of OpenDoc part editors for the Internet). The *raison d'être* of Apple Guide 2.1 is to support Apple Guide for OpenDoc, with particular emphasis on Cyberdog. This version includes a number of new features, some of which are specific to OpenDoc and some of which aren't. Among these new features are the following:

- a combined Full Access window (known as a Merged Access window) that contains all the guide topic areas and index terms for OpenDoc part editors currently running
- a similar ability to merge guides in conventional applications
- the ability to specify a *list* of applications for which a particular guide is intended
- support for a whole new series of OpenDoc-related context checks

PETER COMMONS (commons@guideworks.com) is the vice president of engineering at guideWorks, LLC. He lives happily with his wife, Claire, their dog, Chet, and their cats, "fat cat" Clyde and

"brat cat" Oliver, in Sunnyvale, California, and wonders if he'll ever finish writing updates to Spaceward Ho!•

In this article, I'll describe how Apple Guide behavior has changed in the world of OpenDoc, and I'll also tell you about some new features of Apple Guide 2.1 that you can use in conventional applications. I'll go over the simple steps you *must* take to add a guide to an OpenDoc part editor, and then I'll cover some of the things you *could* do, depending on the kinds of help you want to provide for your part editor. You'll find samples of the resources mentioned in the article on this issue's CD.

If you haven't worked with Apple Guide before, you might want to read "Giving Users Help With Apple Guide" in *develop* Issue 18 before tackling the new concepts presented here. *Apple Guide Complete* is the definitive reference, though the current (1995) edition doesn't cover Apple Guide 2.1.

HOW APPLE GUIDE BUILDS THE HELP MENU

From the beginning, one of the strengths of Apple Guide has been that it enables guide authors to create guides without requiring modification of the application being guided. While most other new Toolbox managers were saying, "To support me, just add a `NewManagerIdle` call in your main event loop," Apple Guide said simply, "I work just fine without any modifications to your application at all!" One of the key elements enabling Apple Guide to work without requiring changes to any code is Apple Guide's automatic population of the Help menu.

The Help menu has also been called the Guide menu at certain times in its history, but both names refer to the same menu (the one labeled with a question mark). •

The algorithm Apple Guide uses to determine how to populate the Help menu, although simple on the surface, has a number of subtleties. With each new release, the algorithm has been extended somewhat. To understand how the Help menu is populated in Apple Guide 2.1, let's look at how the population algorithm has developed over time. You Apple Guide experts might even discover some little-known features.

Table 1 presents a summary of how the different versions of Apple Guide populate the Help menu; details follow.

Table 1. How populating the Help menu has evolved

Apple Guide version	Candidates	Exclusions	Placement in menu
Original Apple Guide	Guide files in application's folder.	Based on <App Creator>, <Gestalt>, 'QLfy'.	Based on type (About, Tutorial, Help, Shortcuts, Other). There can be only one guide file of each type except Other.
Apple Guide 2.0	Add guide files in Global Guide Files folder.	No changes.	Guide files in application's folder take precedence over those in Global Guide Files folder.
Apple Guide 2.1	For OpenDoc shell documents, change application's folder to mean document's folder.	Add an exclusion check based on resources of type 'prts' (for OpenDoc) and 'apsg' (for applications).	Multiprocess guide files ('prts' for OpenDoc, 'mkti' for others), if present, are accessed through the "Process Name Guide" item in the Help type menu position. Guide files with the 'apsg' resource appear in the Help menus of multiple applications.

Note: All candidates are determined when the Help menu is built. Exclusions are applied at the time the menu is built, except for the 'prts' resource test, which is applied when Apple Guide is launched.

“ORIGINAL” APPLE GUIDE

The general process of populating the Help menu hasn’t changed since Apple Guide was first introduced. At application launch time, Apple Guide does the following:

1. creates a list of possible guide file candidates
2. excludes any candidates that don’t match required criteria
3. puts the names of all remaining guide files in their requested positions in the Help menu

For the original version of Apple Guide (any version before Apple Guide 2.0), the list of candidates is created by searching for all guide files that are in the same folder as the application being launched and that aren’t Mixin guide files (see *Apple Guide Complete*, page 2-14, for details about Mixin guide files). A guide file with an alias in the application’s folder would also be added to the list of candidates.

Apple Guide then sees if any candidates should be excluded by subjecting them to these tests:

1. If the guide file contains an <App Creator> command specifying a value that doesn’t match the signature of the current application, it’s excluded (see *Apple Guide Complete*, page 10-8).
2. If the guide file specifies one or more <Gestalt> checks and no <Gestalt> selector returns its required value, the guide file is excluded (see *Apple Guide Complete*, page 10-10).
3. If the guide file specifies exactly one <Gestalt> check whose selector is 'QLfy', Apple Guide looks in the guide file’s resource fork for a resource of type 'QLfy' with a resource ID equal to the requiredValue parameter of the <Gestalt> command. If it finds such a resource, it assumes it’s a 680x0 code resource that takes no parameters and returns a short in register D0 (standard C calling conventions). It calls the resource code and, if the result is 0, the guide file is excluded (see *develop Issue 18*, page 19).

Apple Guide then tries to place the name of each remaining candidate in the position it requested with the helpType parameter of the <Help Menu> command (see *Apple Guide Complete*, page 10-14). Placement of the names of different types of guide files is shown in Figure 1. If two or more final candidate guide files have the same type and that type isn’t Other, Apple Guide includes the name of the first guide file of that type (by alphabetical order) and excludes the others. The names of all guide files of type Other appear in the Help menu.

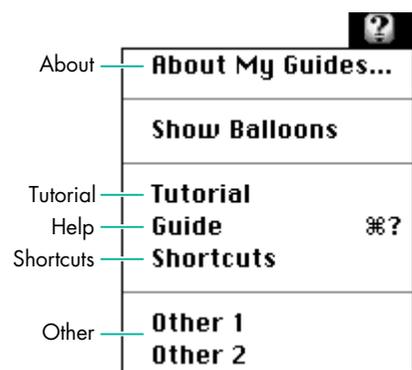


Figure 1. Placement of Help menu items by type in original Apple Guide

APPLE GUIDE 2.0

Apple Guide 2.0 (an update to System 7.5 but backward-compatible with System 7 and 7.1) added some logic to the Help menu population algorithm that hasn't been widely documented. The crux was adding a new place to look for candidate guide files, called the Global Guide Files folder. The Global Guide Files folder resides in the Extensions folder and, as its name suggests, is a place where you can put guide files to make them available to all applications.

When Apple Guide 2.0 is creating its list of candidate guide files for an application, it looks both in the folder containing the application *and* in the Global Guide Files folder. When looking to exclude candidates, Apple Guide 2.0 works almost exactly like the original Apple Guide — it excludes any guide files from its candidate list that don't pass the <App Creator>, <Gestalt>, and 'QLfy' tests.

The only difference is in how Apple Guide 2.0 selects an item for the Help menu if there are multiple candidates. In the original Apple Guide, if two guide files passed all the tests and were of the same type (aside from type Other), the one sorting first alphabetically would be chosen for inclusion. In Apple Guide 2.0, if there are two or more guide files of the same type, the first one alphabetically is still chosen for inclusion, but any guide file in the application's folder is chosen over any guide file in the Global Guide Files folder. So, for example, if there are Tutorial guide files in the application's folder and also in the Global Guide Files folder, those in the Global Guide Files folder will be ignored, and the one that's first alphabetically in the application's folder will be chosen for inclusion. Names of guide files of type Other are added from both the application folder and the Global Guide Files folder.

APPLE GUIDE 2.1

Apple Guide 2.1 adds two new mechanisms to the process of building the Help menu:

- the ability to define multiprocess guide files (with an 'mlti' or a 'prts' resource) and to access these files as a group through a single menu item that presents the user with a combined Full Access window (available in both OpenDoc part editors and conventional applications)
- the ability to specify a list of application signatures that your guide supports (with an 'apsg' resource), so that the name of your guide will appear in the Help menu for each of those applications

We'll look at these new mechanisms in more detail in the next section.

In addition, Apple Guide 2.1 supports document-specific help. Recall that when creating a list of candidate guide files, Apple Guide 2.0 looks in the same folder as the application and in the Global Guide Files folder. Apple Guide 2.1 behaves exactly the same way for conventional applications, but for OpenDoc documents launched via the OpenDoc shell application, Apple Guide 2.1 treats the document's folder as the "application's folder," so it looks in the same folder as the *document* for guide files, rather than in the same folder as the OpenDoc shell application (besides searching the Global Guide Files folder). As a result, the names of guide files in the same folder as the OpenDoc shell application *never* appear in the Help menu; the names of guide files in the same folder as an OpenDoc document can appear in the Help menu for that document.

A CLOSER LOOK AT APPLE GUIDE 2.1

Apple Guide 2.1 introduces a new concept: the *multiprocess guide file*. A multiprocess guide file is specified by including in the guide file an 'mlti' resource for conventional applications or a 'prts' resource for OpenDoc part editors. Because these two are so

similar, I'll discuss them together. I'll also tell you more about the new 'apsg' resource, which when added to a guide file means that the guide file's name will appear in the Help menus of multiple applications.

USING MULTIPROCESS GUIDE FILES

Before multiprocess guide files, the name of every guide file that met Apple Guide's criteria for inclusion in the Help menu appeared as its own menu item. Multiprocess guide files are different. All multiprocess guide files that are candidate guide files and that aren't excluded for any reason (other than that there's more than one of them) get grouped together with the Help guide file (if there is one). This group is accessed through a single Help menu item labeled "*Process Name* Guide," where *Process Name* is the document name for OpenDoc and the application name for conventional applications. This item is placed in the menu position that the name of the Help guide file would otherwise occupy. Figure 2 shows a Help menu with a multiprocess guide item for a Cyberdog document called Peter's Notebook.

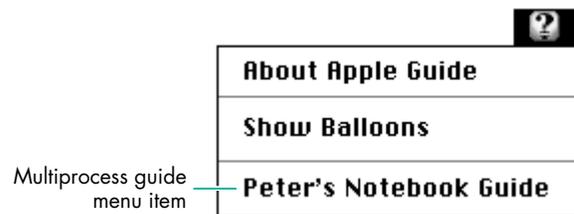


Figure 2. A Help menu with a multiprocess guide item

When users choose the multiprocess guide menu item from the Help menu in a conventional application, they get a combined Full Access window, known as a *Merged Access window*, with these features:

- Topic areas for each multiprocess guide and for the Help guide, listed under the guide file's name in the order specified in the guide's topic area list (see Figure 3).

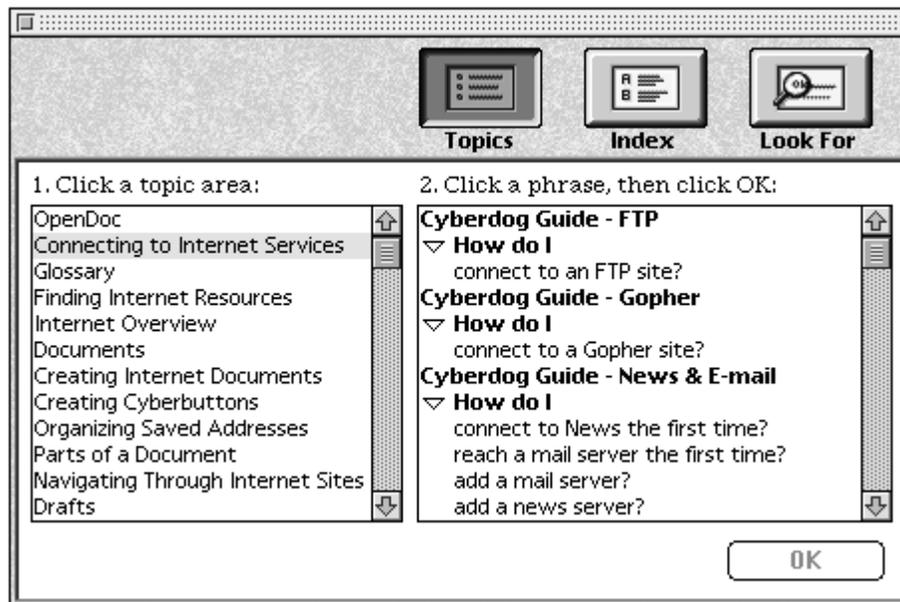


Figure 3. Merged Access window with topic areas

- Index terms for each multiprocess guide, combined, alphabetized, and with duplicates merged. When an index term is selected in the left pane, all topics associated with the term are listed on the right for all guides (Figure 4).
- “Look For” search capability, which can search all multiprocess guide files independently and return a list combining all the topics that match in each guide file (Figure 5).

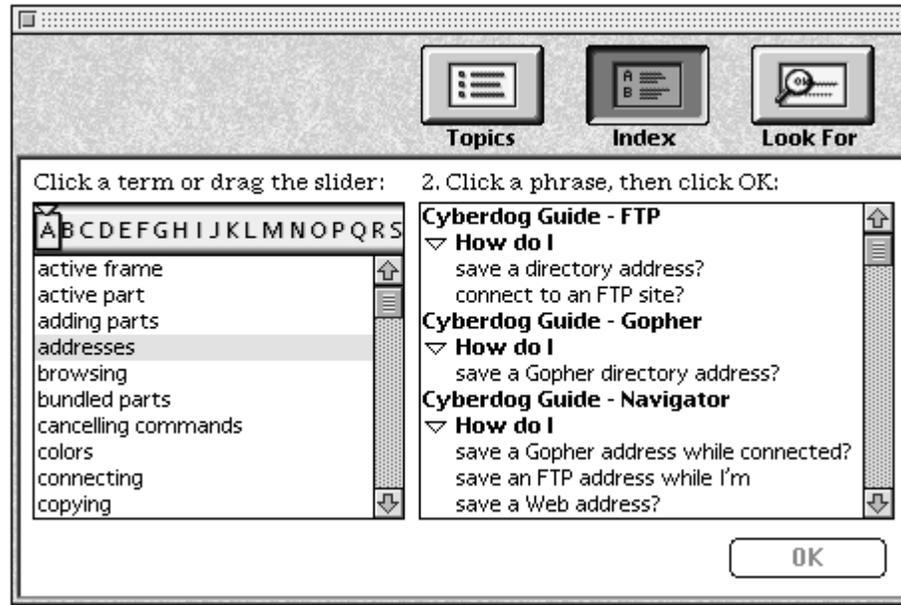


Figure 4. Merged Access window with index terms

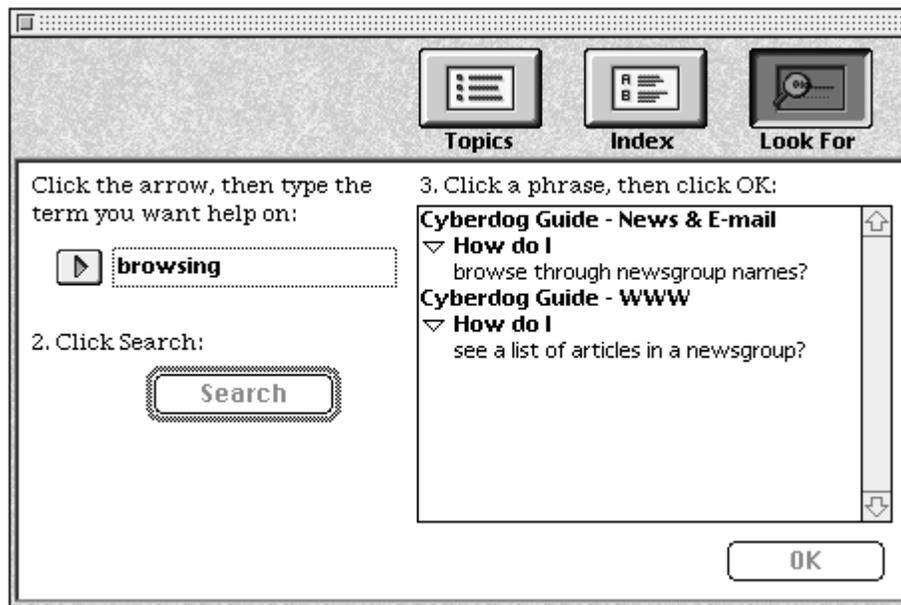


Figure 5. Merged Access window with “Look For” search

If a guide file has an 'mlti' or a 'prts' resource, Apple Guide 2.1 ignores the guide file type specified by the <Help Menu> command. However, for backward compatibility, you probably should declare your multiprocess guides as type Other using the <Help

Menu> command so that Apple Guide versions before 2.1 will list them individually in the Help menu.

As I mentioned before, if there are both multiprocess guide files and a Help guide file, the Help guide gets treated as if it were another multiprocess guide and gets merged with them. If there's only a Help guide file and no multiprocess guide files for an application, the name of the Help guide file appears in its appointed slot in the Help menu, just as in previous versions of Apple Guide. Note that if there are *any* multiprocess guides, the Help guide, if supplied, *must* be a Full Access window guide to be listed in the Merged Access window with the multiprocess guide files.

Multiprocess guide files aren't mixins, as explained in "Mixin vs. Multiprocess Guide Files." For one thing, multiprocess guide files are treated independently by Apple Guide and thus won't have resource conflicts with other multiprocess guide files (unlike mixins). Furthermore, all multiprocess guides must have topic areas and index terms (that is, they must be Full Access window guides); if they don't, as you might expect, they won't be accessible in the Merged Access window.

The 'prts' resource introduced in Apple Guide 2.1 is expressly for OpenDoc. If the current process is an OpenDoc process (any process that supports OpenDoc part embedding), guide files with a 'prts' resource (like those with an 'mlti' resource) are grouped together in a Merged Access window when the user chooses the *Document Name* Guide item from the Help menu.

But before Apple Guide adds multiprocess guide files for an OpenDoc process, it performs one more exclusion check — and unlike all the other exclusion criteria, this one is applied *each* time Apple Guide is launched and *not* when the Help menu is built. Apple Guide compares the list of part editor names in the 'prts' resource with the list of part editors currently in the active process and adds the guide file *only* if it finds a match. Thus, even though the Global Guide Files folder will likely contain multiprocess guide files for every OpenDoc part editor on the user's machine, the user will see help only for editors *currently* in the active process — sort of a "dynamic" Merged Access window. If the 'prts' resource is empty — that is, if it lists no part editor names — the guide will *always* be added to the Merged Access window if the current process is an OpenDoc process.

MIXIN VS. MULTIPROCESS GUIDE FILES

You might be asking, "What about mixins? Aren't they kinda like multiprocess guide files? When would I use mixins instead?"

Mixin guide files are used to add, delete, or replace content in existing guide files. Multiprocess guide files can't do this. Mixins work best for small, incremental changes, but they require good resource management. They also require a main guide file to modify.

Multiprocess guide files in OpenDoc never know which other guide files, if any, will be there when OpenDoc is loaded, so you can't use a mixin in place of a multiprocess guide file — there may not be a main guide file to modify. Also, you don't know which other mixins might be there, so resource conflicts could easily occur.

But you *can* use a mixin to modify your *own* multiprocess guide if you do the following:

- Put the <Mixin> command as the first line in your mixin source and be sure to reference your multiprocess guide file's .sym file to avoid resource conflicts.
- Add all the same exclusions as in your multiprocess guide file. Add additional exclusions if your mixin should activate only in special situations.
- Add an 'mlti' or a 'prts' resource to your mixin if your main guide file has one.
- Make sure your Mixin and your multiprocess guide files both use the <Mixin Match> command so that *your* Mixin guide file mixes only into *your* multiprocess guide file.

USING ONE GUIDE FOR SEVERAL APPLICATIONS

Recall that if you specify a creator code in a guide with the <App Creator> command, the guide file will be removed from the candidate list unless the application's creator code matches. But what if you have a guide for a suite of applications and you want it to appear in the Help menu for each of those applications?

Until now, the only way to do this was to have all the applications in the same folder as the guide file or to have a copy of the guide file (or an alias to it) in each application's folder. With Apple Guide 2.1, you can get the desired result much more cleanly and easily by adding to your guide file an 'apsg' resource listing the application signatures that your guide supports. Then, with your guide file in the Global Guide Files folder, the Help menu will be appropriately populated for every listed application.

If you specify an 'apsg' resource *and* use an <App Creator> command, Apple Guide 2.1 uses only the resource. If there's no resource, the <App Creator> specification is used, if it exists. If you specify neither *and* you put your guide file in the Global Guide Files folder, it will appear in the Help menu for *every* application (which might not be what you want and could greatly annoy your users).

That brings us to the present in the evolution of Apple Guide. Now we'll look at the details of getting your guide file to work with OpenDoc. You'll see how to make help for an OpenDoc part editor accessible to users, and how to add coachmarks, context checks, and events once your guide is up and running.

GETTING A PART EDITOR'S GUIDE INTO THE HELP MENU

Now that you know the history of Apple Guide and the Help menu, you've probably got a pretty good idea of how to get your part editor's guide to appear where you want it. Let's outline the preferred method to accomplish this:

1. Add a 'prts' resource to your guide file. Use the 'prts' resource to specify all the OpenDoc part editors your guide should go with — that is, the editors that when active should have your guide appear in the Merged Access window. Normally, you'll specify only a single part editor, but if you're writing a guide for a collection of related editors, you may list more. If you want your guide to show up no matter which part editors are in the current process, use an empty 'prts' resource (actually two bytes of zeros).
2. Make your guide an Other guide file and specify a creator code of 'odtm', the signature of the OpenDoc shell. This ensures that your guide won't appear in conventional applications if it ever ends up on a machine with an older version of Apple Guide.
3. Install your guide file in the Global Guide Files folder so that it's available to all OpenDoc processes, regardless of how OpenDoc was launched. As mentioned earlier, since OpenDoc is document-centered rather than application-centered, guide files become candidates if they're in the Global Guide Files folder or the same folder as the *document* the user double-clicks to launch OpenDoc, so guide files in the OpenDoc shell application's folder won't normally be considered.

Unless for some reason you want your guide to appear as a multiprocess guide in conventional applications, you don't need and shouldn't add an 'mlti' resource.

Guide Maker doesn't support the new resources yet, so they must be created by hand where required. For the 'prts' resource needed by OpenDoc, I recommend creating a

file named `MyGuideOpenDocResource` that you can then reference from your Guide Script source file with the line

```
<Resource> "MyGuideOpenDocResource", ALL
```

A file on this issue's CD includes samples of these resources and ResEdit templates. For the details, in Rez format, see “New Apple Guide Resources.”

You can provide access to your guide in ways other than Apple Guide's automatic population of the Help menu if you don't mind a little code modification — all of the Apple Guide API calls still work just fine in OpenDoc, so you can add a Guide button or a Help menu item to your part editor and then call `AGOpen` when the user clicks or chooses that item. (See “Giving Users Help With Apple Guide” in *develop* Issue 18 for a detailed description of the Apple Guide API.)

Don't modify the Help menu from within your part editor with the Toolbox calls `HMGetHelpMenuHandle` and `AppendMenu` (although it's perfectly acceptable to do this from within an *application* — even an application that supports OpenDoc embedding). The system, OpenDoc, and Apple Guide don't support this kind of use and many problems will occur. •

If all you need is a simple “book” guide for your part editor, with no coachmarks, context checks, or Apple events, you don't need to do anything else — carrying out the three steps listed above is enough to make Apple Guide help for your part editor accessible within OpenDoc. If you want to provide more elaborate help, read on.

GIVING MORE ELABORATE HELP IN OPENDOC

Once your guide is up and running and the user has selected a topic, Apple Guide 2.1 looks and acts just like previous versions of Apple Guide. The guide window appears on top of the application, and users can click through the guide's panels as they work

NEW APPLE GUIDE RESOURCES

Apple Guide 2.1 introduces the 'mlti', 'prts', and 'apsg' resources to support its new features. These resources must be created by hand.

The 'mlti' resource, by its mere existence, means the file is a multiprocess guide file. This resource is four bytes of zeros.

```
type 'mlti' {
    longint = 0;
};
```

The 'prts' resource is just like an 'STR#' resource — a short specifying the number of part editor names, followed by the names.

```
type 'prts' as 'STR#';
```

The 'apsg' resource is a long specifying the number of application signatures, followed by those signatures.

```
type 'apsg' {
    longint = $$Countof(SigArray);
    array SigArray {
        literal longint;
    };
};
```

Here are some examples of using these Rez templates:

```
// Multiprocess guide file -- conventional app
resource 'mlti' (1000) {};
// Multiprocess guide for OpenDoc to be merged
// when 'Test Clock' is in the active process
resource 'prts' (1000) {{
    "Test Clock"
}};
// Guide only these two applications
resource 'apsg' (1000) {{
    'ttxt', 'MSWD'
}};
```

in the application. But when the guide tries to communicate with the outside world, some things become more complicated. Specifically, you may have to take additional steps when you try to do any of the following:

- Use a coachmark on a part.
- Get context information (perform context checks) on a part editor.
- Send Apple events to a part editor.

The good news is that sending events to other applications such as Apple Guide or the Finder, or getting system context information (such as how many monitors the computer has) or anything else not specified above, works just the same, so I won't talk about those things. Before reviewing the specific changes required to use coachmarks, perform context checks on part editors, and send your part editor Apple events, I need to discuss the biggest difference in approach required to use Apple Guide with OpenDoc: I call it the “target application” problem.

Apple Guide was written to be very System 7 friendly, so almost everything Apple Guide does relies on Apple events. Most of the Apple Guide API calls (such as `AGInstallContextHandler`) secretly use Apple events to get their work done. Unfortunately, when Apple events were designed, OpenDoc wasn't around. Apple events rely on targeting specific processes (usually identified by application signature). Apple Guide assumes that every application has a unique static signature and that only one instance of the application will be running at a time. (If you launch a second document for an application, the second document is opened in the same process as the first one.) Neither of these assumptions holds for OpenDoc.

The process signature for an OpenDoc process is the application signature for the root application. For documents launched with the OpenDoc shell, that signature is 'odtm'. For documents opened into other applications that support embedded parts (as ClarisWorks will soon), the signature is that of the host application. So if, for example, you target a coachmark at the 'odtm' process and your current OpenDoc session is running in ClarisWorks, the coachmark won't fire.

And there can be more than one process with the same signature running. If you already have an OpenDoc document open and go to the Finder to launch a second document, it launches as a separate process. So if both documents are launched using the OpenDoc shell, there will be two processes with the 'odtm' signature running concurrently. Then, for example, if you target a context check at the 'odtm' process, you have no idea which of the two processes will handle the request.

Even if you somehow could manage to target the correct OpenDoc process, a single OpenDoc process can have a number of part editors running inside it — possibly multiple instances of the same part editor. How do you target a specific editor inside a particular process?

Don't abandon hope — all is not lost! But do keep this issue in mind as I describe the steps you need to take to use some of the cooler Apple Guide features.

PROVIDING COACHMARKS

There are two things you need to do to use coachmarks in OpenDoc: always use the Guide Script constant `FRONT` (I'll give examples in a moment), and then use context checks to ensure that your panel is displayed only when the user is in the right process (that is, you need to ensure that the process you want to coachmark is the front process). For the most part, you'll find that coachmarks, except object coaches (because of the target limitation), work just fine in OpenDoc.

MENU COACHES

Menu coaches are used to highlight a particular menu and menu item. You can refer to a menu name and item by number or name.

Menu coaches work fine in OpenDoc. The only recommendation I have (whether or not you're in OpenDoc) is always to specify menu titles and items by name and not by number. This is especially important in OpenDoc, because individual OpenDoc part editors can add menus and menu items at will (check out Cyberdog for a great example of this). For example, to coach the Drafts item in the Document menu, use something like this:

```
<Define Menu Coach> "DocumentDrafts", FRONT, REDUNDERLINE, "Document",  
    "Drafts...", RED, UNDERLINE
```

WINDOW COACHES

Window coaches mark static items in windows. They work in OpenDoc as in applications. For example, you could coach the user to close the window called Log with the following:

```
<Define Window Coach> "CloseBox", FRONT, REDCIRCLE, "Log", CLOSEBOX
```

But to have a window coach highlight a particular element of a part in a window is usually impractical in OpenDoc, because the location of a part in a window isn't predetermined. A part could be all by itself in its own window or anywhere inside a container window. An exception to this is when a part is viewed *only* in a situation where the offset from the window edges is known. A good example of this is the Cyberdog Web browser part, shown in Figure 6. The URL (Uniform Resource Locator) field is always in the same place because the Web browser is always in its own window, so in this case you could coach the URL field with the following window coach:

```
<Define Window Coach> "WebURLField", FRONT, REDARROW(1,4), FRONT,  
    Rect(0,0,125,100)
```

ITEM COACHES

Item coaches are used in Apple Guide to coachmark items specified by dialog ID or balloon ID.



Figure 6. The Cyberdog Web browser window

Dialog IDs work if your OpenDoc part editor brings up a standard dialog with standard dialog items. Or you can use dialog IDs to coachmark items in the OpenDoc shell dialogs. For example, to coach the Save Draft button in the Drafts dialog, you could use the following item coach:

```
<Define Item Coach> "SaveDraftButton", FRONT, REDCIRCLE, DialogID(1)
```

You'll find that because standard Balloon Help doesn't work in OpenDoc except under special circumstances, balloon IDs are probably too tricky to use. The reason for this problem is conflicting assumptions in OpenDoc and Apple Guide about the accessibility of resources. Apple Guide expects all the balloon resource information to be available in the current resource chain. Logically, one would store balloon resources for a part editor in its resource fork, but, due to the intricacies of OpenDoc, a part editor's resource fork *isn't* available when Apple Guide needs it. Since Apple Guide can't get at the Balloon Help resources, it can't look up a balloon ID's rectangle, and thus you can't easily use balloon IDs to coachmark items in OpenDoc.

OBJECT COACHES

Object coaches rely on guide code inside a process to return the coaching rectangle. The name of the desired object to be coached is passed to the specified target application, which responds with the coaching rectangle.

Unfortunately, Apple Guide allows only one object coach handler per process. If two part editors in one OpenDoc process both try to install object coach handlers, the second one will override the first one (that is, any object coaches will be handled by the second editor and never by the first). This means you can't use object coaches reliably with OpenDoc processes.

If you decide to use object coaches in your part editor because you know that yours will be the only one installing an object coach handler, be sure to use the OpenDoc API to do the installation. •

This obstacle to using object coaches in OpenDoc has been noted as a serious concern by many people, including Apple Guide authors and those on the Apple Guide and OpenDoc teams. Some possible solutions have been proposed. We can hope that updated versions of Apple Guide and OpenDoc will support one of them in the near future (although nothing has been promised yet).

APPLESCRIPT COACHES

AppleScript coaches don't require a target application at all, so they don't suffer directly from the target application problem. To determine the target rectangle, though, the script itself usually has to communicate with one or more OpenDoc part editors. You can make part editors scriptable, but remember to use OpenDoc's scripting API, not the regular AppleScript calls.

PERFORMING CONTEXT CHECKS

There are three sources of context checks for guides written for OpenDoc:

- the standard suite of context checks (part of the Standard Includes package on most Apple Guide CDs)
- a new suite of standard OpenDoc context checks (designed expressly for OpenDoc)
- any custom context checks you write for your OpenDoc part editor only

THE STANDARD SUITE OF CONTEXT CHECKS

The standard suite of context checks includes ways of testing basic elements of the traditional Macintosh interface. Here are some examples of these context checks:

- Is a window with a given name the frontmost window?
- Does this Macintosh have more than one monitor?
- Is the Open item in the File menu enabled?

The context check definitions and the resources you must include in your guide to use them are on any Apple Guide authoring CD (such as the CD that comes with *Apple Guide Complete*, the *Custom Solutions* CD, or the Mac OS SDK CD).

These context checks work pretty well in OpenDoc processes. All of the *system* information context checks work (bit depth, printer info, file sharing info, and so on), because they all target the Finder for their information. The *application* information context checks work (again, you'll have to target these with FRONT), except for menu item checks, since OpenDoc controls how the menus are stored and displayed. At present, if you're using the standard context checks in an OpenDoc process, you can't determine whether a menu item is enabled, is checked, or exists at all (though you *can* write a custom context check to do this).

The *process* context checks do work, but they aren't very helpful because they're based on target application signatures. For example, asking if the current active process is 'odtm' will tell you if the active process is the regular OpenDoc shell but won't help you determine whether it's some other OpenDoc process (because the user could have some *other* OpenDoc shell application). Nor will this confirm that the active process is the desired process (since there could be several OpenDoc processes running).

THE STANDARD OPENDOC CONTEXT CHECKS

To provide guide authors with tools to answer questions about OpenDoc processes, a suite of OpenDoc context checks has been written. For these context checks to work, the user must have an OpenDoc shell plug-in called AppleGuidePlugIn correctly installed. This plug-in is installed automatically when Apple Guide 2.1 is installed. If the plug-in isn't installed, all OpenDoc standard context checks will always return FALSE.

Before we look at the available OpenDoc standard context checks, you should note that for every one of these context checks that takes the name of a part editor as one of its parameters, there are two variations: if the second SHORT parameter is 0, the editor names must match *exactly*; if the second SHORT parameter is 1, the actual editor name need only *contain* the text specified in the context check. Both variations exist for all context checks that take a part editor name. All of these functions return a Boolean result (TRUE or FALSE).

Is the Apple Guide plug-in available?

```
<Define Context Check> "IsPlugInAvailable", 'odag', FRONT, SHORT:1
```

This context check is a way to make sure that the plug-in has been installed correctly and is available to run when the next OpenDoc process runs, thus ensuring that any standard OpenDoc context check will return a correct result. The only catch is that the plug-in isn't actually installed until the first OpenDoc process has been launched, so this check will return FALSE if no OpenDoc process has yet been run, even if the plug-in is available. With this limitation in mind, you can define this context check, which ties an Apple Guide context check name to one of the resources you included above. Then you might use it this way:

```
<Define Sequence> "How do I do something?"
<If> IsPlugInAvailable()
    # instruction panels
<Else>
    # panel saying that this guide requires Apple Guide 2.1
<End If>
<End Sequence>
```

You probably won't need to use this context check and will just assume that the Apple Guide plug-in was installed correctly.

Is OpenDoc active and frontmost?

```
<DCC> "IsOpenDocActiveAndFrontmost", 'odag', FRONT, SHORT:2
```

Use this context check to see whether the active process is an OpenDoc process. It isn't application signature based and thus will return TRUE for any OpenDoc process, no matter what the host application is.

Is the part editor named "MyPart 1.0" installed?

```
<DCC> "PartEditorInstalled", 'odag', FRONT, SHORT:4, SHORT:0, LPSTRING
<DCC> "PartEditorInstContains", 'odag', FRONT, SHORT:4, SHORT:1, LPSTRING
```

To determine whether a particular part editor is installed in the OpenDoc Editors folder and is available, use this context check. It has nothing to do with whether an instance of the part editor is currently in the active process. You might use a call like `PartEditorInstalled("MyPart 1.0")` to make sure that a particular part editor has been installed on the machine before you tell users to do something that depends on the part editor's being available. This is one of the functions that takes a part editor name as its argument; you could use the less specific version of the function by calling `PartEditorInstContains("MyPart")`, but take care — you might end up matching someone else's part editor if you're *too* general!

Is the part editor named "MyPart 1.0" in the active process?

```
<DCC> "PartInActiveProcess", 'odag', FRONT, SHORT:6, SHORT:0, LPSTRING
```

This is a way to check whether an instance of the specified part editor is in the active (frontmost) *process*. The corresponding part may or may not be in the active (frontmost) *window*.

Is a "MyPart 1.0" part in the active window or in a nonactive window?

```
<DCC> "PartInActiveWindow", 'odag', FRONT, SHORT:7, SHORT:0, LPSTRING
<DCC> "PartInNonActiveWindow", 'odag', FRONT, SHORT:8, SHORT:0, LPSTRING
```

These two context checks enable you to see whether there's an instance of the part either in or not in the active (frontmost) window.

Is a "MyPart 1.0" part in the active document or in a nonactive document?

```
<DCC> "PartInActiveDoc", 'odag', FRONT, SHORT:14, SHORT:0, LPSTRING
<DCC> "PartInNonActiveDoc", 'odag', FRONT, SHORT:15, SHORT:0, LPSTRING
```

An OpenDoc process can contain multiple *documents*. A particular document in a process might have more than one *window*. These two context checks enable you to see if an instance of the specified part is in *any* of the active document's windows or any windows of a nonactive document. It's unlikely you'll need these checks — most of the time you'll want to check whether a part is in the active window or the active process.

Is a “MyPart 1.0” part the active part (the active frame)?

```
<DCC> "PartIsActiveFrame", 'odag', FRONT, SHORT:10, SHORT:0, LPSTRING
```

This is a way to check whether an instance of the specified part is the currently active part. This is useful to know because a part editor’s menus are usually available only when the part is active.

Is the active part the root part?

```
<DCC> "ActivePartIsRoot", 'odag', FRONT, SHORT:9
```

If you need to determine whether the active part is the root part for the active document, use this check.

Does the active part allow embedding?

```
<DCC> "ActivePartAllowsEmbedding", 'odag', FRONT, SHORT:5
```

With this check you can determine whether the active part is a container part and allows other parts to be embedded inside it. You might use this if, as part of a task, you need to get the user to drag a new instance of a part into the active container part.

Is the active document bundled?

```
<DCC> "ActiveDocumentIsBundled", 'odag', FRONT, SHORT:3
```

This is a way to check whether the active document is bundled. Bundling a document prevents any subparts in the document from being activated; clicking on a subpart in a bundled document will select the subpart but won’t activate it. In essence, this makes all subparts in the document read-only.

Is the active document dirty?

```
<DCC> "ActiveDocumentIsDirty", 'odag', FRONT, SHORT:11
```

This is a way to check whether the active document is dirty (needs to be saved). If this context check returns TRUE, the Save menu item is enabled. You might use this to tell the user to save changes if necessary.

CUSTOM CONTEXT CHECKS

If you still need more specific part information that isn’t available through the standard suite of context checks or the OpenDoc standard context checks, just as with standard applications you’ll need to write custom context checks.

To define and use a custom context check, you must work around two difficulties. The first is the target application issue we’re now so familiar with. As before, it’s easy to work around: when writing the <Define Context Check> command for your custom context check, you’ll need to use the FRONT constant for the target application. The second problem concerns the fact that you could have multiple instances of a part editor running at the same time, in either the same or a different OpenDoc process. This is a problem for both the guide author *and* the custom context check writer.

The primary concern is for the guide author: if there are multiple instances of the same part editor in one or more currently running OpenDoc processes, it’s impossible for your guide to identify which part editor you’re providing help for. Let’s look at an example. Before step 1 of your task, you use the standard OpenDoc context checks to make sure an instance of your part is active. You then tell the user to do step 1. Step 2 requires step 1 to have been completed, so you want to do a custom context check to see if step 1 has been done. However, if there are two instances of your part around, which instance the custom context check queries is unknown. Users may have

successfully completed step 1, but the context check may come back saying they haven't. In this case, they'll be stuck at this point and won't be able to continue.

The context check writer has similar concerns. The way a part editor would install and remove a context check handler would probably be to call `AGInstallContextHandler` in its constructor and `AGRemoveContextHandler` in its destructor. If it's done this way, anytime a new instance of the part is created it overrides (and removes) the previous context handler, so the last instance of the part to be created is the one that will supply context information, no matter which process it's in. In addition, when a part editor calls `AGRemoveContextHandler`, it will remove whichever handler is currently installed; if one of two instances of the part is destroyed, the context handler will be removed, leaving no context handler for the remaining instance.

Unfortunately, there's no simple answer to these concerns at this time. There are partial solutions for particular cases, though. For example, if you know that your part editor will definitely have exactly one instance, you might just take your chances. If you always want to have the context check respond about the currently active instance of the part (if the active frame is the desired part), you can write an 'extm' context check that you install in your guide that asks `OpenDoc` for the currently active frame, and if the part behind that frame is your part, do some context checking on it. As more people try to tackle custom context checking, better solutions will evolve, perhaps using the `ODExtension` mechanism of `OpenDoc`.

APPLE EVENTS AND APPLESRIPT

Sometimes you also want to send Apple events or launch AppleScript scripts from your guide when a user clicks a particular button or goes to a particular panel. As I've said before, this is still possible in `OpenDoc` part editor guides. The only thing that's more challenging is when you want to send an event to your part editor. `OpenDoc` supports Apple event handling for part editors by overriding a number of the standard Apple event Toolbox routines and by providing a way to target a particular part editor. Explaining how to do this is beyond the scope of this article; for more information, read the *OpenDoc Programmer's Guide*, especially Chapter 9, "Semantic Events and Scripting."

REACH OUT AND GUIDE SOMEONE

As you can see, `Apple Guide 2.1` provides a number of new features, both for standard guides and guides written for `OpenDoc` part editors. And despite a few limitations, writing guides for part editors is as easy as writing them for standalone applications. So try it out! Users and reviewers seem to agree: `Apple Guide` — and thus any application or part editor that has guides — is in a class by itself.

RELATED READING

- *Apple Guide Complete: Designing and Developing Onscreen Assistance* by Apple Computer, Inc. (Addison-Wesley, 1995).
- "Giving Users Help With Apple Guide" by John Powers, *develop* Issue 18.
- *OpenDoc Programmer's Guide for the Mac OS* by Apple Computer, Inc. (Addison-Wesley, 1995). Available on the *OpenDoc Developer Release CD* and other places.
- The `guideWorks` World Wide Web page, located at <http://www.guideworks.com>.

Thanks to our technical reviewers Sharon Everson, Troy Gaul, Devon Hubbard, James Miyake, John Powers, and Melissa Sleeter. •



VINCENT LO

THE OPENDOC ROAD

Facilitating Part Editor Unloading

In the traditional application model, the code for an application typically remains loaded until the process quits. In OpenDoc, starting with version 1.0.1, a part editor is loaded when it's needed during a session and unloaded when it's not. As a result, valuable memory space can be reclaimed and reused by other part editors.

Even though part editor unloading is mostly transparent to part editors, there are a few things a part editor should do to ensure the success of this scheme. I'll describe these things after giving you a closer look at how part editor unloading works. Pay careful attention, because the crash you prevent may be your own.

HOW PART EDITOR UNLOADING WORKS

The part editor unloading mechanism is enabled by facilities provided by SOMObjects™ for Mac OS (the Apple implementation for the Macintosh of the IBM SOM™ technology). The basis of part editor unloading is a reference-counting system that enables OpenDoc to keep track of which part objects are in use. I'll explain reference counting and then give the gory details of how part libraries are unloaded, which differs for static and dynamic classes.

Every persistent object (part, frame, link, and so on) in OpenDoc is given a reference count by the draft that creates it. When the object is first created or acquired, its reference count is initialized to 1. Whenever the object is acquired after that (through either the draft's or the object's Acquire method), its reference count is incremented by 1. Whenever the object is released (by a call to the object's Release method), its reference count is decremented by 1.

When all clients have released their references to an object, the reference count of the object is 0, and at this

point the draft can delete the object to regain the memory it occupies. However, deletion may not be immediate when the object's reference count drops to 0. In actuality, object deletion is deferred until the purge mechanism of the draft is triggered. Typically, a purge is initiated by the storage system (for example, during a save operation) or by the document shell (such as when the document shell realizes that memory is running low).

In response to a purge request, the draft deletes all the persistent objects and storage units that aren't in use — that is, objects whose reference count is 0. When all the part objects belonging to a certain part editor are deleted, SOM calls the Code Fragment Manager (CFM) to unload the part editor library. The CFM calls the CFMTerminate routine of the part editor library, and both the code and data sections of the library are destroyed. Some details of how the library is unloaded depend on the kind of SOM class it contains — either static or dynamic.

Objects created using **new *className*** and SOM kernel services (`somNewObject`, `somNewClassReference`, and `SOMObject::somGetClass`) are static class objects. Most (if not all) objects created by a part editor fall into this category. A static class is unloaded when the code that created the static class object is unloaded. Therefore, when a part editor is unloaded, all static classes in the same library are unloaded as well. Other interdependent libraries may also be unloaded; there will be more on this later.

Objects created using the runtime name or ID class-lookup services of SOM (for example, `SOMClassMgr::somFindClass`, `somNewObjectByName`, or `somGetDynamicClassReference`) are dynamic class objects. OpenDoc parts are dynamic class objects, since they're created by name. Extension objects are also dynamic because `ODExtension` is implemented as a dynamic class; subclasses of `ODExtension` inherit its dynamic property as long as they call the parent's `InitExtension` method. Other OpenDoc classes will be converted to dynamic classes as the need arises; check the notes accompanying future OpenDoc releases for a listing of these.

A dynamic class guarantees that its code and the code for its inherited classes won't be unloaded until the last object of the class is deleted. When the last instance of a part class is deleted, SOM unloads the CFM library containing the part class.

VINCENT LO is Apple's technical lead for OpenDoc. In his leisure time, he loves to travel and sample exotic food. Living in Hong

Kong for many years convinced him that no food can scare him, but he'll continue to trot the globe to seek out gustatory challenges. •

REFERENCE-COUNTING GOTCHAS

Every persistent object must have a correct reference count for the part editor unloading mechanism to work. If a part object has a reference count that's higher than the correct count, the object will remain valid throughout the session even though it's no longer being used. This object will keep its associated part editor library from being unloaded until the process quits. Conversely, if an object has a reference count that errs on the low side, the object may be deleted, causing its library to be unloaded. Referencing an invalid object pointer usually results in a crash.

The best way to avoid reference-count errors is to familiarize yourself with OpenDoc persistent objects and follow the recipes outlined in the *OpenDoc Programmer's Guide for the Mac OS*. Be sure to pay special attention to the following potential trouble spots.

Avoid self-referencing. If a part object keeps a reference to itself, its reference count will be at least 1 and its part editor library won't be unloaded until the session ends. Since a reference to the part is passed in as an argument in every ODPart method, a part shouldn't need to store a reference to itself.

Break circular references between parts and frames. Each display frame has a reference to its part (after the part has been internalized). Even though a part isn't required to keep a reference to its display frames, most parts do so for convenience. This creates a circular reference between a part and its display frames, so the reference count alone won't indicate when deleting a part is appropriate. The situation becomes more complicated when the part has embedded frames and also keeps references to them.

To break these circular references, OpenDoc offers the Close and Remove protocols.

- The Close protocol is triggered when a containing part decides to get rid of its runtime-embedded frame objects. The containing part calls the embedded frame's Close method. The frame first calls its part's DisplayFrameClosed method, which should release its reference to the display frame and close its embedded frames (if any) before releasing its reference to the part.
- The Remove protocol works similarly to the Close protocol. However, the protocol is triggered when a containing part decides to remove its embedded frames from both runtime storage and persistent storage. The Remove protocol is propagated through the ODPart::DisplayFrameRemoved and ODFrame::Remove methods.

Unregister frames and parts when idle time is no longer needed. When a part registers its frame with the OpenDoc dispatcher for idle time, the dispatcher retains a reference to the frame. Until the part editor unregisters the frame object from the dispatcher, the object will remain resident in memory with a reference count greater than 0. This will prevent the object from being deleted and its library from being unloaded. The part editor should unregister the frame when the Remove or Close protocol is triggered.

On rare occasions, a part may have a display frame that's not in the frame hierarchy originated from the root frame. For example, a part may have some frames stored for the View in Window command. Don't internalize those frames and register them for idle time until the frames are actually used.

A part may also register itself with the OpenDoc dispatcher for idle time. As in the case of registered frames, the dispatcher retains a reference to the part object. To ensure that the object is deleted and that its library is unloaded at the earliest possible time, the part editor must unregister itself from the dispatcher as soon as idle time is no longer needed. This usually occurs when all the part's display frames have been closed or removed.

Watch extensions for reference-counting problems. OpenDoc's extension mechanism enables separate parts in a document to communicate with each other directly. By creating an associated extension object, a part editor can extend its part interface to satisfy special needs. To enable efficient communication, the extension object maintains a reference to the part object that created it; the part typically keeps a reference to the extension so that it can give out the same extension object again if it's requested.

In general, an extension object is released before its creator, thus preventing any reference-counting problem for the part. But if the reference count of the extension object isn't maintained correctly, or if the client of the extension object refuses to release it, the part object can detach the extension object from itself by calling ODExtension::BaseRemoved. A well-behaved extension object should then report errors to clients when it's being accessed. An even better idea is to avoid using the base removal mechanism and instead to define the scope and lifespan of an extension.

A LIBRARY-UNLOADING GOTCHA

When a part editor is unloaded, SOM unloads the CFM library associated with the part editor. Moreover, if there's another library in the same library closure as the part editor, the CFM will unload it if it doesn't

Listing 1. Creating a dynamic object with ODNewObject

```
#include "ODNewObj.h"

ODObjectNamespace* objNameSpace = (ODObjectNamespace*) nameSpMgr->CreateNameSpace(ev,
                                         kOSAScriptingTool, kODNULL, 1, kODNSDataObjectTypeODObject);
Sample_ScriptRunnerAgent* agent = (Sample_ScriptRunnerAgent*)
                                         ODNewObject("Sample::ScriptRunnerAgent");
objNameSpace->Register(ev, kOSAScriptingTool, agent);
```

belong to another library closure. (A *library closure* is a group of shared libraries whose interdependencies cause them to be loaded together by the CFM.) This means that code other than the part editor residing in the same CFM library closure is unloaded as well. If the developer hasn't foreseen this possibility, it can lead to unfortunate consequences.

Let's consider a typical example. A part editor creates a SOM object from a static class that resides in the same library closure as the part editor. The part editor then installs the object in a name space so that others can access it. If this object doesn't hold a reference to the part, the part may be deleted (and its library closure unloaded) when the object reference is still in the name space. The next time this object is used, a crash will likely occur because the code associated with the object has been unloaded with the part editor.

To prevent this from happening, you should ensure that no class whose code resides in the same library closure as the part editor outlives the part itself. If an object must outlive the part that creates it, the object should

be created dynamically. OpenDoc provides a utility function, ODNewObject, to create objects by name. The code in Listing 1 illustrates how a dynamic object is created with ODNewObject.

As mentioned earlier, parts and extensions are dynamic objects and thus don't require ODNewObject. ODNewObject is used mainly on SOM classes that a part developer has created.

Once a class has been accessed dynamically, the class remains dynamic until the process exits or the use count of the class maintained by SOM goes to 0. Until then, all object references created for the class are considered dynamic by SOM.

COUNTING ON YOU

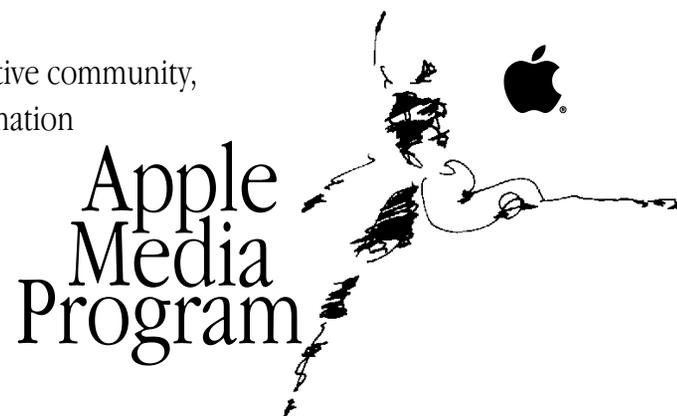
Part editor unloading in OpenDoc is a great scheme for managing memory efficiently. But its success depends on the cooperation of each and every part editor. If you keep in mind the gotchas detailed in this column, your part editor will avoid the pitfalls and reap the benefits of wise resource use.

Thanks to Dave Bice, Erik Eidt, and Troy Gaul for reviewing this column. •

www.amp.apple.com

Designed for content developers and the creative community, the Apple Media Program is your online information source for technology, tools, and resources.

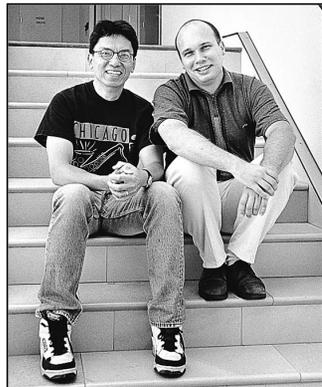
Get up-to-date industry trends on video, film, music, broadcasting, Internet publishing, print media, and game development markets.



For an application form, visit the "Get with the Program" area at <http://www.amp.apple.com> or call 408.974.4897.

Mac OS 8 Assistants in System 7 Applications

Assistants are a key part of the Mac OS 8 help system. An assistant makes an application's features easier to use and more readily accessible. In anticipation of Mac OS 8, this article will show you how to build Mac OS 8-style assistants into your System 7 applications, from design to implementation. We illustrate this with a sample assistant developed for the Internet Configuration System.



**JOSÉ ARCELLANA AND
ARNO GOURDOL**

Developers are constantly pursuing two goals that seem to be at cross-purposes: making applications more powerful and making them easier to use. All too often, power brings complexity, when in fact power can be used to simplify things for the user. Assistants can make the powerful features you're building into your applications easier to access and handle.

An assistant offers the user an alternate interface. It focuses on a specific activity that the average user is likely to want to do and frames the application's functionality to support that activity. The defining aspect of an assistant is the interview, in which the user is asked to supply information about preferences and typical activities accomplished with the application.

Although assistants can be implemented under System 7, they'll be able to do more using Mac OS 8 technologies. In this article, we talk about what Mac OS 8 assistants are and give some general guidelines for designing an interview. We also provide an example of how to implement an interview in System 7. Our sample assistant helps users with the Internet Configuration System (Internet Config), a utility for setting preferences for Internet applications. Internet Config is described in detail in the article "Implementing Shared Internet Preferences With Internet Config" in *develop* Issue 23. The source code for the Internet Setup Assistant is included on this issue's CD.

The final name for assistants, which have also been called experts, has not been decided at the time of this writing. Whichever the final name, the interview-based interaction that forms the basis of assistants will be an integral part of the Mac OS 8 help system. •

JOSÉ ARCELLANA (arcellan@apple.com) is a human interface designer working on Mac OS 8 assistants, Apple Guide, and related technologies. He lives a rich analog life with his wife, their four-year-old child, and a yellow Labrador retriever in an 86-year-old Craftsman bungalow in Oakland. The house has lots of books, four guitars, and no television. •

ARNO GOURDOL (arno@apple.com) has been spotted on top of various San Francisco Bay Area chthonic protrusions with a merry group of Moof hikers in a futile attempt to cure his acrophobia. He has recently been engrossed by the *Epic of Gilgamesh* and would love to find someone with a good copy of the twelve tablets. In his spare time, Arno is the technical lead of the Mac OS 8 assistance and related technologies team. •

INTRODUCING ASSISTANTS

An assistant is a small single-purpose application that can do any or all of the following:

- frame computer-based tasks in terms of real-world activity
- hide details from the user by filtering out options that aren't applicable
- pull together elements from different parts of the user interface to support a single activity
- manage tasks so that they're executed on demand or when a specific condition becomes true
- help the user provide the information needed by conducting interviews
- make reasonable assumptions, based on user context and user activity, that work for the vast majority of users

For example, a résumé-formatting assistant would ask the user how formal or casual the résumé should be. It would then make assumptions, hiding such details as typeface selection and paragraph formatting. An assistant for maintaining a computer would use task scheduling to check for viruses when it makes sense, optimize the hard disk when it needs to, and so on.

At first glance, assistants might seem to resemble Microsoft's wizards. Currently, however, there are differences between them. Wizards don't make reasonable assumptions based on user context and user activity. Instead of filtering out options that aren't applicable, they present all options. While assistants provide an alternate interface to an application or set of applications, wizards might be the only user interface to a task, and they aren't capable of pulling together elements from other places in the interface. Wizards also can't schedule tasks for later execution.

WHEN TO USE ASSISTANTS

Assistants are meant to augment and not replace the more direct ways to control your application. They shouldn't be used to cover up a flawed user interface design, such as dialog boxes or commands that are too difficult to figure out. Users should always be able to do directly in the program's primary interface whatever an assistant does for them indirectly, though it might take more steps to accomplish the task in the primary interface.

How do you determine when a specific area of the user interface needs a redesign and when it could use an assistant? In general, an assistant is most effective when it supports an activity that many users want to perform with your application but that can only be accomplished if the user has a better-than-average familiarity with your application's feature set and user interface. For example, we developed an assistant for Internet Config because it takes several steps in different places of the interface to set up your Internet preferences for the first time.

THE INTERVIEW

An assistant conducts a brief interview, consisting of a series of simple questions, to get the information it needs from the user. The interview should be:

- Short. Ask as few questions as possible. To minimize the number of questions, the assistant should make as many reasonable assumptions as possible.
- Simple. Make the questions easy to answer. If a difficult question needs to be asked, the interview should provide information that helps the user answer the question.

-
- Concise. No words should be wasted, even when providing information to help the user answer a question.
 - Neutral. The tone should be conversational and friendly, not too familiar or too cold.

Note that an interview isn't a dialog box, nor is it a dialog box taken apart and presented in a series of smaller dialog boxes. Adding "Assistant" to the name of a command that calls up a dialog box doesn't make the dialog box an interview or the command an assistant. Finally, the interview isn't a way to give the computer a personality.

THE INTERVIEW WINDOW

The assistant interview takes place in a fixed-size, movable window. Whether it's a regular (document) window, a modal window, or a floating window depends on the context from which you anticipate it being invoked. Assistants should be accessible from appropriate places in your user interface, such as through a menu item or a button in a dialog box. They should be accessible by name (so that the name of the menu item or button is the name of the assistant) or, if the context is clear, by the words "Assist Me."

The interview window contains header, content, and navigation areas, as you can see in Figure 1.

- If the interview window is a document window, the assistant's name appears in the title bar and the header area displays the interview phase that the user is in. If it's not a document window, the header area displays the assistant's name followed by a colon and the interview phase.
- The content area contains text (usually a question and a brief explanation) and editable text fields and other controls that are used to answer questions and enter information.
- The navigation area contains buttons that help the user move through the interview, such as left and right arrow buttons, which lead to the previous or next panel, and a Go Back button that takes the user back to the context from which the assistant was opened. A number between the left and right arrow buttons indicates how many panels the user has been through (see Figure 1).

DESIGNING AN ASSISTANT FOR INTERNET CONFIG

To demonstrate how to add an assistant to your application, the rest of this article describes how we designed and implemented an assistant for Internet Config, a popular shareware utility program.

With Internet Config, the user can create a single file that stores preferences and settings for all Internet services. As long as an e-mail program, Web browser, or other Internet application uses the Internet Config preferences file, the user doesn't have to reenter Internet preferences for each program.

Internet Config's central location for user interaction is the window shown in Figure 2. Clicking each button in the main window brings up a window in which the user enters information and sets preferences.

Internet Config enables users to store a broad range of preferences, from e-mail addresses to file conversion formats. Most users never need to set many of these preferences, but the dedicated Internet habitué probably finds them all useful. Internet Config risks being overwhelming for the sake of completeness — a perfect opportunity for an assistant. Our assistant makes Internet Config easier to use by helping the user create a preferences file that stores only the preferences that are most commonly set.

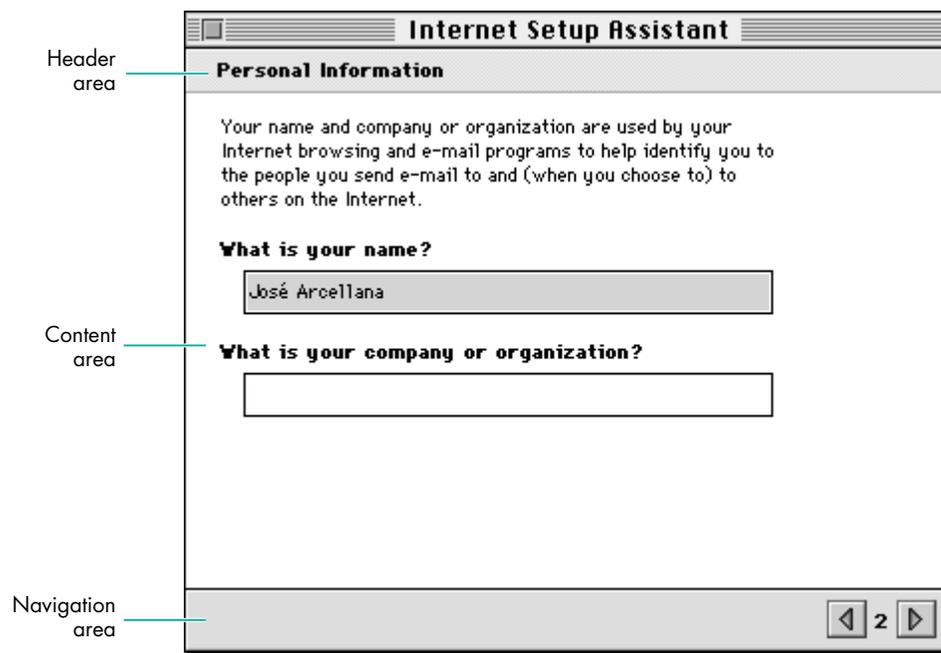


Figure 1. A typical assistant interview window



Figure 2. Internet Config's main window

DECIDING WHAT THE ASSISTANT WILL DO

In designing an assistant, you need to determine what most of your users will want to do. In the case of Internet Config, recent market studies show that most people use the Internet for e-mail, browsing, and downloading files. Given that information, the Internet Config assistant's functionality should be limited to setting preferences in those areas. The power-surfer minority that wants, for example, to change the default text editor can still do so directly through Internet Config's interface.

Assistants should be made available from wherever they make sense. For the Internet Setup Assistant, it would be helpful to add an Assist Me button to the main Internet Config window and an Internet Setup Assistant command to the Help menu. Application programs that use Internet Config could have a similar button in logical places in their user interfaces. In addition, the Internet Setup Assistant could be automatically displayed the first time the user launches Internet Config.

THE INTERNET CONFIG INTERVIEW

Our interview will elicit the information we need in order to store a simplified set of preferences for the user. As shown in Figure 3, the interview begins with an introduction that tells the user in plain language the type of questions that will be asked and describes what the assistant will do (or not do) based on the user's answers.

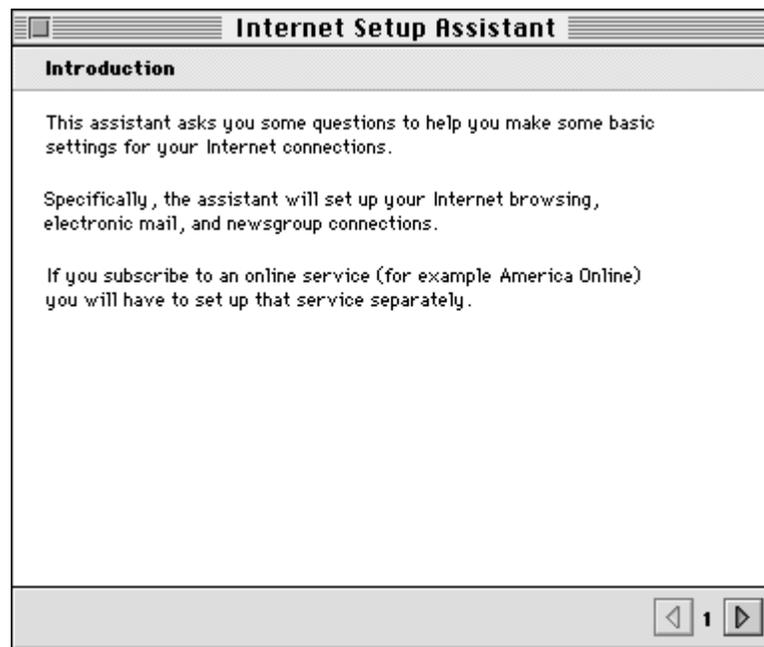
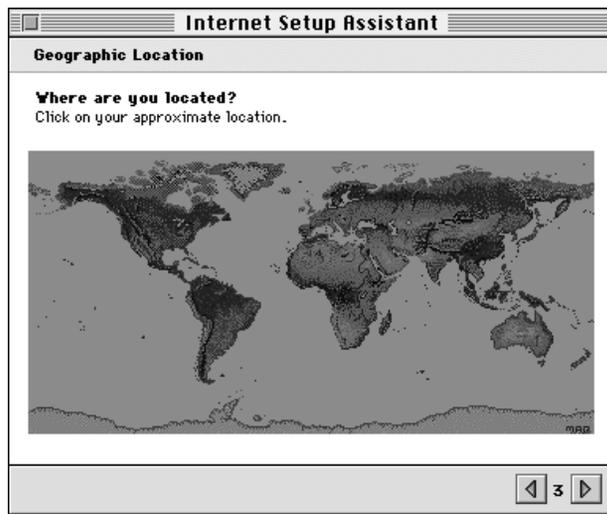


Figure 3. The Internet Setup Assistant introduction

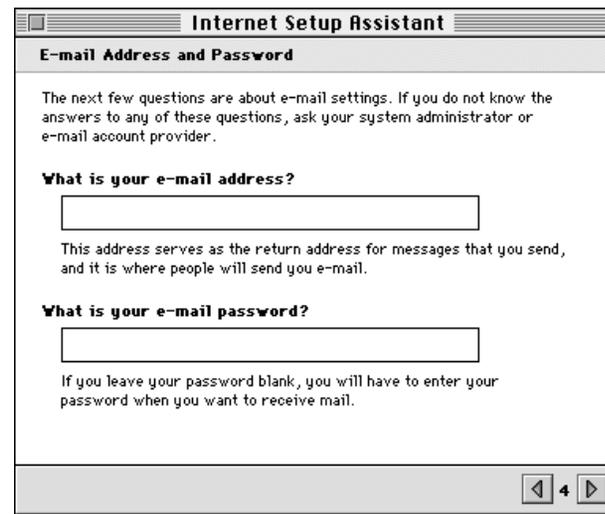
The next panel, titled “Personal Information,” was shown earlier in Figure 1. This panel poses two questions that are easy to answer: it asks for the user’s name (filling in a default supplied by the file sharing setup) and company or organization. This starts the interview off smoothly while still obtaining necessary information.

The remaining interview panels that ask questions are listed below (in the order they appear) and are shown in Figure 4. Note that for a few of the questions, the assistant provides some information because the questions might be too difficult for some users to answer. The goal is to keep the interview self-contained, so that the user doesn’t need to go to Apple Guide or a manual to figure out what to do.

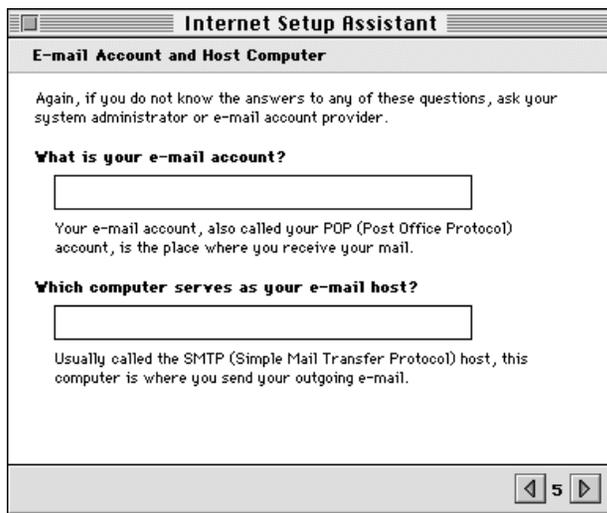
- Geographic Location — This panel asks a question that doesn’t directly relate to what the assistant does for the user. Many users might be confused if asked to select an FTP server; instead, the interview asks for the user’s location and then the assistant uses the answer to make a reasonable guess as to what the user’s default FTP servers are.
- E-mail Address and Password — The password appears as bullets when typed and is encrypted.
- E-mail Account and Host Computer
- Signature — This panel asks for a “signature” to be appended to the user’s e-mail messages. A line of hyphens is supplied as the default.
- World Wide Web Home Page — In Internet Config, setting the Web home page preference takes place in the Other Services window, listed among less commonly used preferences such as the WAIS gateway and the Whois host. Many users might be stumped by these options and not find what they’re looking for. Since we’ve determined that most users are interested in setting their Web site, the interview covers setting this preference. The default Web site entered is taken from the Web browser program.
- Newsgroup Host Computer



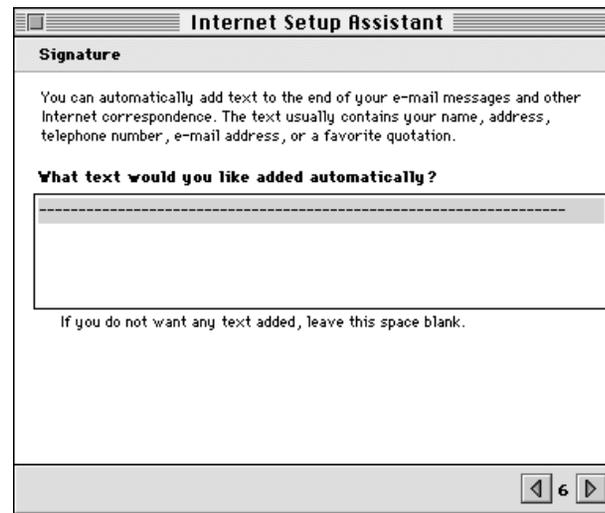
Geographic Location



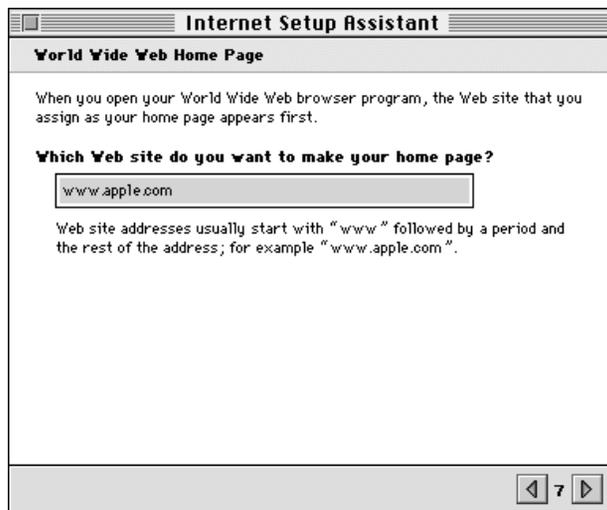
E-mail Address and Password



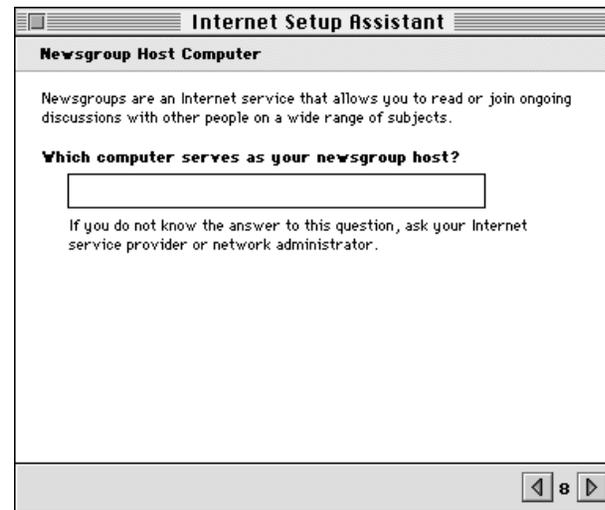
E-mail Account and Host Computer



Signature



World Wide Web Home Page



Newsgroup Host Computer

Figure 4. Internet Setup Assistant interview questions

When a user types an answer that's clearly wrong (such as an e-mail address that doesn't include the @ character), we recommend that you integrate the error trapping into the flow of the interview questions, rather than presenting an alert box. For example, when the user clicks the right arrow button and there are invalid values in the current panel, the next panel should point out the error and restate the question. The goal is to preserve the question-and-answer, conversational characteristics of the interview.

Finally, when the assistant has asked all the questions, it presents the conclusion panel (Figure 5). The user can see more details by clicking the Show Details button (which then changes to Hide Details); the assistant shows the information it will use in creating the Internet preferences file (name, organization, e-mail address, and so on), summarizing the user's interview responses.

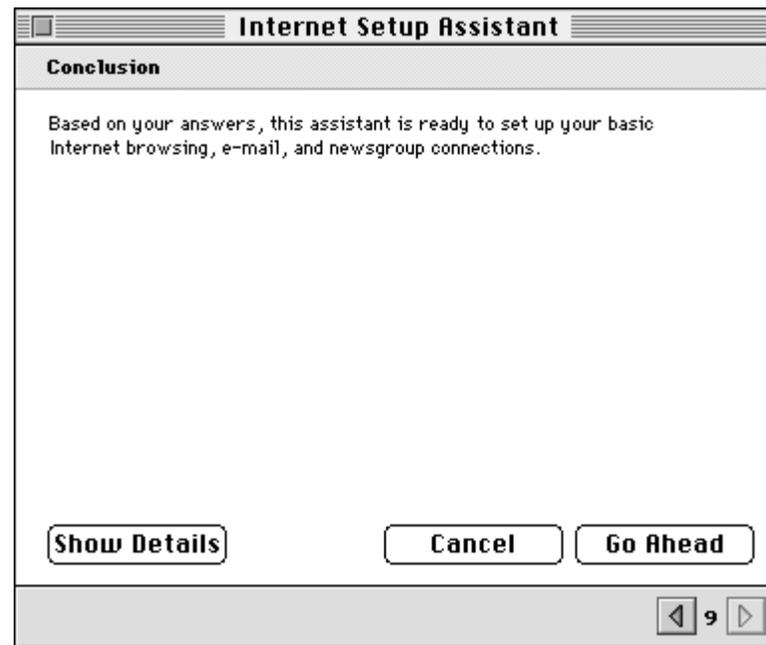


Figure 5. The Internet Setup Assistant conclusion

That's it: ten questions in seven panels, plus an introduction and a conclusion.

IMPLEMENTING THE INTERVIEW

You can use your favorite application framework to develop an assistant, or write it from scratch as a small application. On this issue's CD, we provide sample code for developing an assistant. We don't recommend using Apple Guide to conduct the interviews. Assistants and Apple Guide are different components of the help system; they should look related but still different from each other. Also, Apple Guide is a less-than-efficient tool for implementing assistant interviews.

The programming techniques used in this example are specific to System 7 but will continue to work under Mac OS 8. As long as you avoid using undocumented features or directly accessing data structures that have an accessor routine, your code should work fine under Mac OS 8. In addition, if you insulate your code from specifics of the Macintosh Toolbox, it will be easier to add Mac OS 8 features later on. For some details about Mac OS 8 compatibility, see the article "Planning for Mac OS 8 Compatibility" in *develop* Issue 26.

Our assistant is based on a simple framework that provides a lightweight object-oriented coating on top of the Macintosh Toolbox. For example, we have classes that provide an object-oriented layer on top of Point (CPoint), Rect (CRect), WindowPtr (CWindow), DialogPtr (CDialog), and so on. We also have a simple application shell, TApplication. Those files are grouped in the framework folder.

The classes aren't dependent on each other, so you can use them easily in your existing application. You can create an assistant dialog by using an object of a TAssistant subclass and sending it the appropriate events. Your usual framework can still be used for handling your event loop and your application's windows. To implement the appearance of assistants using your own framework, check out the TAssistant class to see how we've done it.

The interview is presented in a dialog by a TAssistant object. The class TAssistant is a subclass of CMultiDialog, which allows you to have subdialogs that can be switched in and out as needed. You could use a similar technique to switch among multiple panels in a preferences dialog. Our implementation uses the ShortenDITL and AppendDITL routines, as shown in Listing 1. For another way to change panels, see the article "Multipane Dialogs" in *develop* Issue 23.

Listing 1. Changing panels in the assistant's interview

```
void CMultiDialog::SetSubDialog(SInt16 subDialogID)
{
    if (GetWindowRef() == NULL)
        CreateWindow();

    if (GetSubDialog() != subDialogID) {
        // Save parameters in current panel.
        for (int i = 0; i < kDialogParametersCount; i++) {
            if (fParameters[i].dialogItem != 0)
                GetItemText(fParameters[i].dialogItem, fParameters[i].value);
        }

        // Remove items from current dialog.
        if (fSubDialogID > 0)
            ShortenDITL(GetDialogRef(),
                CountDITL(GetDialogRef()) - fCommonItems);

        fSubDialogID = subDialogID;

        {
            // Add new dialog items to the dialog.
            Handle items = GetResource('DITL', GetSubDialog());
            assert(items != NULL);
            AppendDITL(GetDialogRef(), items, overlayDITL);
            ReleaseResource(items);
        }

        // Prepare the items in the dialog.
        DoPrepareDialog();
    }
}
```

When dialogs are displayed in assistants, you often need to capture the information the user enters so that you can put it back if the user goes back to that panel. You need to be able to do this until the user clicks the Go Ahead button in the last panel. To simplify this task, we substitute a key string, which is stored in the DITL of the dialog, for a string that can change dynamically. The substitution is done each time the dialog is displayed. Listing 2 shows how to substitute parameters in the dialog by replacing any instance of a key that appears in a static or editable text dialog item with a corresponding value. In addition, if an editable text item contains only a key, the value entered in the dialog will be associated with the key. This is an extended version of the ^0 parameter substitution done by the Dialog Manager.

Listing 2. Substituting parameter values in dialogs

```
void CDialog::SubstituteParameters(void)
{
    LazyHandle substitutionText, itemText;

    // Reset the association between parameters and dialog items.
    for (int i = 0; i < kDialogParametersCount; i++)
        fParameters[i].dialogItem = 0;

    // Loop through all dialog items.
    for (int item = 1, itemCount = CountDITL(GetDialogRef()); item <= itemCount; item++) {
        SInt16 itemType = GetItemType(item);

        // If it's a static or editable text item...
        if (itemType == kEditTextDialogItem || itemType == kStaticTextDialogItem) {
            Boolean itemTextChanged = false;

            // Copy the text to a handle.
            Str255 itemTextString;
            GetItemText(item, itemTextString);
            itemText.Set(&itemTextString[1], itemTextString[0]);

            // Loop through the parameters.
            for (int j = 0; j < kDialogParametersCount; j++) {
                // If the parameter is used as a nonempty key...
                if (fParameters[j].key[0] != 0) {
                    if (itemType == kEditTextDialogItem &&
                        ::IdenticalString(itemTextString, fParameters[j].key, NULL) == 0) {
                        // If the edit field contains only the key, associate the item index with the
                        // parameter. The parameter value is updated when the text changes.
                        fParameters[j].dialogItem = item;
                    }
                    {
                        // Replace the key with the parameter value, using the Script Manager.
                        substitutionText.Set(&fParameters[j].value[1], fParameters[j].value[0]);
                        if (::ReplaceText(itemText, substitutionText, fParameters[j].key) > 0)
                            itemTextChanged = true;
                    }
                }
            }
        }
    }
}
```

(continued on next page)

Listing 2. Substituting parameter values in dialogs (*continued*)

```
    if (itemTextChanged) {
        // The item text has changed. Put the modified text back in the dialog item.
        Str255 s;
        s[0] = itemText.GetSize();
        BlockMoveData(*itemText, &s[1], s[0]);
        SetItemText(item, s);
    }
}
```

When the assistant starts up, we try to capture as much information as possible about the user's environment. For example, we use Internet Config information that the user has previously entered.

When the user has entered all the information we ask for and clicked the Go Ahead button, we again use Internet Config, this time to set the user's preferences. The CInternetConfig class provides a C++ wrapper to the Internet Config API. See the details in the source code.

THE POWER OF ASSISTANCE

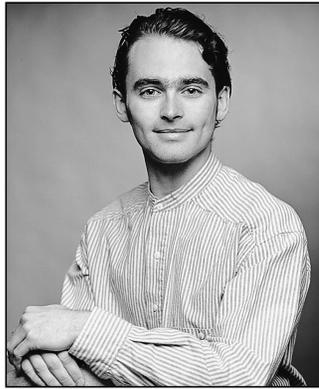
As you can see, the Internet Setup Assistant doesn't try to do everything for all users. Instead, it helps complete tasks in a way that a majority of users will find useful and valuable.

This sample assistant, though a relatively unambitious demonstration, should start you thinking about how to design and develop assistants for your own applications. Nothing is quite as powerful as simplicity.

RELATED READING

- "Implementing Shared Internet Preferences With Internet Config" by Quinn "The Eskimo!", *develop* Issue 23.
- "Multipane Dialogs" by Norman Franke, *develop* Issue 23.
- "Planning for Mac OS 8 Compatibility" by Steve Falkenburg, *develop* Issue 26.
- *Mac OS 8 Revealed* by Tony Francis (Addison-Wesley, 1996), Chapter 13, "Assistance Services."

Thanks to our technical reviewers Deeje Cooley, Winston Hendrickson, Rick Mann, Jim Palmer, and Jim Rodden. •



DAVE EVANS

BALANCE OF POWER

Stalking the Wild Defect

Once again I found myself bleary eyed and fighting sleep, yet I continued to search for understanding. Having already struck down two possible causes for my enigma, I was now searching for new clues. I stubbornly refused to rest until I had flushed out the software defect.

My journey had begun modestly enough as I chanced upon a capricious crash in my software. I wondered which assumption or logic was at fault. Armed with only my low-level debugger, I began a hunt that would consume me into the dead of night. On this adventure through the dark Mac OS interior, I crossed rivers of mode switches, hopped islands of cross-TOC glue, and set snares in a jungle of native PowerPC code.

In this column I'll walk you through one facet of that relentless pursuit, pointing out the key landmarks I used to navigate and demonstrating the tools I used to survive. This should help guide you through your own future explorations of the innards of PowerPC code.

ON THE HUNT

Programming for a Power Macintosh may appear similar to your efforts on a 680x0-based Macintosh, but on close inspection you'll find PowerPC code far more interesting to debug. The relatively simple landscape of a 680x0 world gives way to confusing and insidious terrain on a Power Macintosh. Routine descriptors, dual assembly languages, and native glue are obstacles that impede your progress.

My subject was a crash that occurred when PowerPC applications called `MaxApplZone`. I was certain the problem was in my recent system software changes, but I needed to see what happened right before the crash to

understand it. I started by setting a breakpoint when an application called `MaxApplZone`. (Later I'll describe a good technique for setting these breakpoints.) Then I traced through the system routine and looked for anything startling.

One application executed the following code just before calling `MaxApplZone`:

```
0093B260  mflr    r0
0093B264  stw     r31,-0x0004(SP)
0093B268  stw     r30,-0x0008(SP)
0093B26C  stw     r29,-0x000C(SP)
0093B270  stw     r0,0x0008(SP)
0093B274  stwu   SP,-0x0050(SP)
0093B278  lwz    r30,-0x3940(RTOC)
0093B27C  bl     MaxApplZone
```

The preamble to `MaxApplZone` saves registers R29 to R31 on the stack, creates a stack frame, and loads a local variable into R30 from the application's TOC globals before calling the routine. If we trace through this and then step into the `bl` (branch and link) instruction to `MaxApplZone`, we find the following:

```
0094CBFC  lwz    r12,-0x7E60(RTOC)
0094CC00  stw    RTOC,0x0014(SP)
0094CC04  lwz    r0,0x0000(r12)
0094CC08  lwz    RTOC,0x0004(r12)
0094CC0C  mtctr  r0
0094CC10  bctr
```

This code is standard cross-TOC glue. The caller of a routine has the responsibility to set the TOC register (RTOC) correctly for it. Routines imported from other code fragments will have a different TOC value than the application. The PowerPC Code Fragment Manager supplies the correct TOC value and the address of the imported routine in a pair of long words called a *transition vector*, or TVector. In this case, the TVector is stored as global data at the application's TOC value minus `$7E60` bytes. This glue code loads the TVector's address in R12 and then uses that to load the address of the routine in R0 and the new TOC value. It uses the counter register and the `bctr` (branch to counter register) instruction to jump to the correct address, so the return address in the link register will not be changed.

After tracing through this glue code, we find ourselves in a different kind of glue. The `MaxApplZone` TVector points to a routine in the `InterfaceLib` code fragment,

DAVE EVANS and fellow Apple engineer Rus Maxham took another adventure by motorcycle this summer. This time they journeyed to Utah and skirted the Great Salt Lake. Turning north, they discovered the beautiful and unspoiled vistas of Idaho.

Cottonwood flower petals rained on them as they crossed into Washington. Hectares of wheat farms and the blustery Columbia River guided them to Oregon. One cracked tailpipe and two quarts of oil later, they finally arrived home in California. •

as listed below. On this computer, you can guess that the code fragment is in ROM because the address of the routine is very high, \$40A0E30C in this case. Since the routine is in ROM, you can't effectively set a breakpoint at its beginning.

```
MaxApplZone
+00000 40A0E30C  mflr   r0
+00004 40A0E310  stwu   SP,-0x0040(SP)
+00008 40A0E314  stw    r0,0x0048(SP)
+0000C 40A0E318  lis    r0,0x0001
+00010 40A0E31C  subic  r5,r0,0x5F9D
+00014 40A0E320  lwz    r3,MaxApplZone(r0)
+00018 40A0E324  li     r4,0x3802
+0001C 40A0E328  bl     CallOSTrapUniversalProc
+00020 40A0E32C  lwz    RTOC,0x0014(SP)
+00024 40A0E330  lwz    r12,0x0048(SP)
+00028 40A0E334  addic  SP,SP,0x0040
+0002C 40A0E338  mtlr   r12
+00030 40A0E33C  blr
```

You might expect the real MaxApplZone routine to do much more than what appears in this routine. In fact, this routine is simply glue for the 680x0 A-trap table: it gets the address of MaxApplZone from that trap table (don't try this yourself without GetOSTrapAddress, kids) and then uses the CallOSTrapUniversalProc routine to call the address.

Most of the routines in InterfaceLib are actually just like this glue routine for the trap table. Because the routines go through the trap table, PowerPC applications will be affected by patches to the trap table; if they were to bind directly with the system code fragments, patches would be bypassed.

To continue with our tracing, we must step up to and then into CallOSTrapUniversalProc. This takes us to more cross-TOC glue:

```
40A06D10  lwz    r12,0x0008(RTOC)
40A06D14  stw    RTOC,0x0014(SP)
40A06D18  lwz    r0,0x0000(r12)
40A06D1C  lwz    RTOC,0x0004(r12)
40A06D20  mtctr  r0
40A06D24  bctr
```

Since CallOSTrapUniversalProc is part of the Mixed Mode Manager, it's implemented in the MixedMode code fragment. This cross-TOC glue finds the TVector for that routine and calls through to it. When we step through this and over the last **bctr** instruction, we're magically transferred not to the Mixed Mode Manager but instead to 680x0 code. Wow! MacsBug knew we were calling a universal procedure pointer, so it spared us the trace through the mode switch and took us

directly to the location of the universal procedure pointer, in this case the following 680x0 code:

```
0031B160  MOVE.L  ApplLimit,D0
0031B164  MOVE.L  HeapEnd,D1
0031B168  SUB.L   D1,D0
0031B16A  MOVEQ   #$14,D1
0031B16C  CMP.L   D0,D1
0031B16E  BLE.S  *+$000A
0031B170  MOVEQ   #$00,D0
0031B172  MOVE.W  D0,MemErr
0031B176  RTS
0031B178  JMP     $00167FCC
```

From my experience tracing through the system, I'd guess that this 680x0 code is a patch on top of the real MaxApplZone, because it compares two numbers and in one of only two cases jumps to an absolute address. The absolute address was probably set when this code was installed as a patch, and it points to either the real MaxApplZone routine or another patch.

The patch appears to check whether the value of the ApplLimit low-memory global is within 20 bytes of the value of HeapEnd. If so, it simply returns noErr in the low-memory global MemErr without calling through to the real MaxApplZone. This patch is probably part of the system software, designed to fix a bug in the ROM without having to replace the entire real MaxApplZone routine.

Now if we trace through this patch and visit the absolute address \$167FCC from the patch, we find the following:

```
No procedure name
00167FCC  *_MixedModeMagic
00167FCE  BTST   D3,D0
00167FD0  ORI.B  #$00,D0
00167FD4  ORI.B  #$00,D0
00167FD8  ORI.B  #$3002,D0
00167FDC  ORI.B  #$04,D1
00167FE0  ORI.B  #$8274,D7
00167FE4  ORI.B  #$00,D0
00167FE8  ORI.B  #$00,D0
00167FEC  ORI.B  #$A036,D0
```

Aha! This ugly disassembly is actually a routine descriptor in disguise. The **_MixedModeMagic** trap invokes the Mixed Mode Manager from 680x0 code, and it always appears at the beginning of a routine descriptor. Since this trap is at the beginning of each routine descriptor, you can simply construct a routine descriptor and then jump to it in 680x0 code. The **drd** dcmd in MacsBug will let you see this routine descriptor in a meaningful way. When I typed **drd pc** in this case, I saw the contents of Listing 1.

Listing 1. Displaying a routine descriptor

```
drd: 00167fcc
MixedModeMagic: 0xAAFE, version: #7, flags: 0x00 (NotIndexable)
LoadLoc: 0x00000000, reserved2: 0x00000000, SelectorInfo: 0x00 (No Selector)
Routine Count (zero-based): 0x0000 (#0)
---- Routine Record 0x0000 (#0) at 0x00167fd8 ----
ProcInfo: 0x00003002, Reserved1: 0x00000000, ISA: #1 (PowerPC)
Record Flags: 0x0004 (Absolute, IsPrepared, NativeISA, PassSelector, IsNotDefault)
ProcPtr: 0x00078274, offset: 0x00000000, selector: 0x00000000
```

Notice the number of fields displayed in Listing 1. For simple routine descriptors like this one, you'll only need to look at the ProcPtr entry on the last line of the display. More complicated routine descriptors have an array of routines, and you'll need to look for a passed selector to determine which one is actually used. •

The last line of the listing shows a ProcPtr value of \$00078274. This is the address of the TVector for MaxApplZone in the MemoryMgr code fragment. Since the TVector structure has the routine address as its first element, dereferencing that address once will produce the address of the actual routine. Typing **ilp @78274** to dereference the TVector and disassemble PowerPC code showed me this:

```
__MaxApplZone
+00000 000CA1A8 mflr    r0
+00004 000CA1AC stwu   SP,-0x0040(SP)
+00008 000CA1B0 stw    r0,0x0048(SP)
+0000C 000CA1B4 bl     __HSetStateQ+0073C
+00010 000CA1B8 crmove cr7_SO,cr7_SO
+00014 000CA1BC extsh  r4,r3
+00018 000CA1C0 li    r3,0x0000
+0001C 000CA1C4 bl     SetEmulatorRegister
+00020 000CA1C8 lwz   RTOC,0x0014(SP)
+00024 000CA1CC lwz   r12,0x0048(SP)
+00028 000CA1D0 addic  SP,SP,0x0040
+0002C 000CA1D4 mtlr  r12
+00030 000CA1D8 blr
```

This is the MaxApplZone routine in the Memory Manager. It appears to call more substantial subroutines when it branches to __HSetStateQ+0073C, but this is the actual routine.

WALKING BACK OUT

We've braved routine descriptors, glue, and patches to make it this far. I won't dive further into the Memory Manager for this illustration, but let's try an instructive walk back out from the MaxApplZone routine.

After tracing through this routine, we step over the **blr** instruction to branch back to the link register address. To our surprise we not only switch back to 680x0 emulation mode but we appear to be lost in darkness. The following 680x0 F-line instruction will be executed next:

```
No procedure name
0162D0A0 DC.W    $FE02
```

We switched back to 680x0 emulation mode because we're returning from the 680x0 patch call to a routine descriptor. Typing **ip** to disassemble at the current location shows what appears to be garbage, however:

```
No procedure name
0162D08C NEGX.L  D7
0162D08E EOR.W   D3,(A0)+
0162D090 BCHG   D0,-(A2)
0162D092 DC.W    $D0F0
0162D094 DC.W    $FFFF
0162D096 ORI.B   #$00,D4
0162D09A DC.W    $FFFF
0162D09C ORI.B   #$A063,D0
0162D0A0 *DC.W   $FE02
0162D08C NEGX.L  D7
0162D08E EOR.W   D3,(A0)+
0162D090 BCHG   D0,-(A2)
0162D092 DC.W    $D0F0
0162D0A2 ORI.B   #$9C,D0
0162D0A6 BCLR   D0,D3
0162D0A8 BCHG   D0,-(A2)
0162D0AA ADD.B   D0,(A0)
```

Here's the secret: The \$FE02 F-line instruction is very much like the _MixedModeMagic trap in that it can signal the transition from emulated 680x0 code to PowerPC code. Just as with the routine descriptor that we saw earlier, executing the \$FE02 instruction will in this case cause us to switch back to PowerPC native mode and will bring us to a completely different address.

Listing 2. The stack upon return to PowerPC code

```
Displaying memory from sp
0162D080 DDDD DDDD DDDD DDDD 7FFF 7FFF 4087 B758 -t······· ··@á
0162D090 0162 D0F0 FFFF 0004 0000 FFFF 0000 A063 ΣX·b-··········
0162D0A0 FE02 0000 009C 0183 0162 D110 0000 3802 †c·····ú·É·b-···
```

Truly perceptive readers might have noticed that the program counter at the \$FE02 instruction is actually on the stack. Listing 2 shows a memory dump of the first 48 bytes of the stack at this time. Notice that the word at the beginning of the third line (at \$162D0A0) is the \$FE02 instruction we're about to execute.

As we trace over that \$FE02 instruction, we find ourselves back inside the InterfaceLib glue routine for MaxApplZone. Tracing through those last instructions finally takes us back to the application code where we started, as shown here:

```
MaxApplZone
+00020 40A0E32C lwz      RTOC,0x0014(SP)
+00024 40A0E330 lwz      r12,0x0048(SP)
+00028 40A0E334 addic   SP,SP,0x0040
+0002C 40A0E338 mtlr   r12
+00030 40A0E33C blr
No procedure name
0093B280 lwz      RTOC,0x0014(SP)
0093B284 li      r31,0x0001
```

Notice that when we returned to a previous code fragment, we immediately restored the TOC register to a value saved on the stack. Not only is the caller responsible for setting the TOC register before calling a routine, it's also responsible for restoring this register when the call returns.

This concludes our romp through the wilderness of the modern PowerPC environment. We traced from an application's code fragment, through the InterfaceLib fragment and then a patch in the trap table, to a routine descriptor for the real MaxApplZone routine, and ultimately back again.

CATCHING POWERPC CALLS

Earlier I glossed over how to set a breakpoint and catch an application as it calls MaxApplZone. Now I'll describe a good trick for doing this.

The MacsBug debugger doesn't implement 680x0 A-trap break commands for PowerPC code yet. But you can easily mimic the A-trap break feature in PowerPC code, using the FindSym, PlayMem, and PPCJump

MacsBug macros. You can use those macros if you install the file "PowerPC dcmds" (which you'll find on this issue's CD) into your MacsBug Preferences folder.

Say, as an example, that you'd like to catch all PowerPC code that calls the Toolbox routine ReleaseResource. PowerPC code fragments access this routine by importing its entry point from the InterfaceLib code fragment. Typing **FindSym ReleaseResource** on my Power Macintosh 8100 produces the following:

```
findsym: "ReleaseResource"
"ReleaseResource" #1796 TVec 0001acc0
(40a15978,0001ea14) in "InterfaceLib"
```

FindSym is case sensitive. When looking for an entry in InterfaceLib, for example, you must spell the routine name exactly and capitalize letters perfectly; typing "releaseresource" rather than "ReleaseResource" will not work. •

This tells us a few things. ReleaseResource's TVector is located at the address \$0001ACC0. That vector contains the address for the routine at \$40A15978 and the InterfaceLib's TOC value, which is \$0001EA14. FindSym will return TVector addresses for each application or fragment bound to the routine. In System 7 these will usually all be the same TVector.

If I need to catch callers to ReleaseResource, I could then simply type **brp 40a15978** to set a PowerPC breakpoint at the beginning of the InterfaceLib code. On the Power Macintosh 8100, however, this address is in ROM. Setting breakpoints in ROM is more difficult for MacsBug, which returns this message:

```
Warning: This requires stepping through each instruction
```

Your Macintosh might become unusable if MacsBug is forced to single-step through all the code. Because MacsBug can set a breakpoint in RAM with less difficulty, we'll now use the PlayMem and PPCJump macros to set an equivalent breakpoint in RAM.

PlayMem is a MacsBug variable that points to 512 bytes of scratch memory in RAM. The PPCJump macro

expands to a set of PowerPC instructions for jumping to an absolute address. So the command

```
sl PlayMem PPCJump 40a15978
```

writes the following instructions to MacsBug's scratch memory:

```
lis    r0,40a1      | 3C0040A1
ori    r0,r0,5978   | 60005978
mtctr  r0           | 7C0903A6
bctr                   | 4E800420
```

Now I'll replace the value of the TVector with our new code in scratch space, by typing **sl 0001acc0 PlayMem**. PowerPC code bound with InterfaceLib will now call my new code instead of ReleaseResource, but my code will then correctly pass control to ReleaseResource.

Finally, typing **brp PlayMem** will set the PowerPC breakpoint we want.

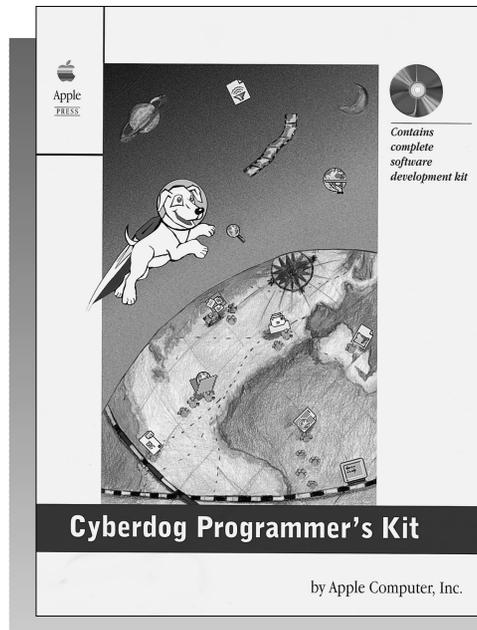
When PowerPC code tries to call the ReleaseResource trap via InterfaceLib, execution will stop at my breakpoint in PlayMem. At that point, typing **ipp lr** will list PowerPC instructions around the address in the link register, quickly showing me which code was calling the trap.

AFTER THE HUNT

Although I seriously doubt I would find enjoyment in hunting live animals, I've found the hunt for software defects truly rewarding. Some problems are a definite challenge, and I often learn something new about the Mac OS with each riddle solved. I hope that knowing the details of my pursuit will help you in your own future quests.

Thanks to Nitin Ganatra, Pete Gontier, Jim Luther, and Alex Rangel for reviewing this column. •

Explore OpenDoc



with Apple Press and Addison-Wesley

Cyberdog Programmer's Kit

Cyberdog is the compelling new Internet-access technology that gives users flexible and fully extensible access to the World Wide Web, news groups, e-mail, and many other network services. The Cyberdog Programmer's Kit offers everything you need to create Cyberdog-aware components—tutorials and code examples, a complete reference to the Cyberdog C++ classes and methods, and the Cyberdog software development kit on the accompanying CD-ROM.

OpenDoc Programmer's Guide, Apple Computer's official reference, gives an overview of OpenDoc development and describes OpenDoc programming in detail. The accompanying CD-ROM contains the OpenDoc Class Reference, the complete reference to the OpenDoc programming interface.

• 688 pages w/CD-ROM • \$44.95 • 0-201-47954-0

OpenDoc Cookbook, a companion volume to OpenDoc Programmer's Guide, provides tutorials and code samples that show you how to create OpenDoc software components for the Mac OS platform.

• 206 pages • \$24.95 • 0-201-47956-7

▲▲ Addison-Wesley Publishing Co.

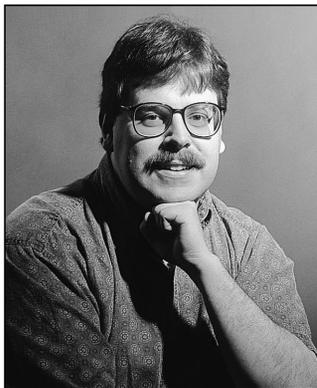
<http://www.aw.com/devpress/>

Apple Press, a new publishing effort from Apple Computer, brings you the latest information on Apple technology. Apple Press publishes books and book/CD packages that appeal to a wide market—from developers and programmers, to educational professionals, and to the home and business markets.

Available at fine technical bookstores in your area, or call 1-800-822-6339 to order.
For international orders fax 617-942-2829.

Game Controls for QuickDraw 3D

Whether the user is navigating a starship or examining a model of the DNA helix, your first-person 3D application must allow user control of the camera movements in a scene. You must keep changing the camera's position and orientation in response to what the user wants to see. Here you'll learn how to create those camera movements and handle the user's directions. As part of the bargain, you'll even get a refresher course in the associated geometry.



PHILIP MCBRIDE

Letting the user control the movement of the camera (and thus the view) is critical to first-person interactive 3D games and extremely useful in 3D modeling systems. Through QuickDraw 3D's camera functions and supporting mathematical functions, you can create game controls that direct the position and orientation of a camera. In general, game controls take user input from any input device and control the camera in ways that emulate movements of players, such as people or aircraft. Game controls are useful for any type of 3D viewer application, including 3D Internet browsers.

You'll start your career as a camera operator by learning about the basic moves you can make with the camera. Then you'll create the various camera movements, keep the camera movements smooth, and translate user inputs to move the camera. The sample code (which is provided on this issue's CD) is a 3D viewer application with camera movements activated by the keyboard or the mouse. In all of the code, the geometry has been kept as simple as possible, but if you need to brush up, you'll find a refresher course on calculating points and vectors in 3D space.

For an overview of QuickDraw 3D, turn to "QuickDraw 3D: A New Dimension for Macintosh Graphics" in *develop* Issue 22. That article discusses topics like reading models, using a viewer, creating a camera, and managing documents that have 3D information. To learn more about those and related topics, see the list of recommended reading at the end of this article.

MOVING THE CAMERA

We'll be controlling camera movements based on first-person viewing, so the camera will be our eyes. But before we move through a scene, let's take a look at the kind of camera moves we plan to use.

PHILIP MCBRIDE (mcbride@apple.com) is currently adding QuickDraw 3D and QuickTime VR to HyperCard 3.0. He used to spend time contemplating the meaning of the universe until he figured it out. Now he can be seen wandering the halls at Apple and mumbling something about

needing more content. Lately, Philip has been looking into investing in anteaters after learning that a full 20% of the earth's biomass is made up of ants and termites. Just think about that overcrowding the next time someone says we don't need to invest in space travel. •

The camera movements you would create in a 3D game for a person who is driving a vehicle or walking on level ground are examples of ground movements. These camera moves include moving forward, backward, sideways to the left, and sideways to the right, plus turning to the left (pan or yaw left) and turning to the right (pan or yaw right). Figure 1 illustrates these basic ground movements.

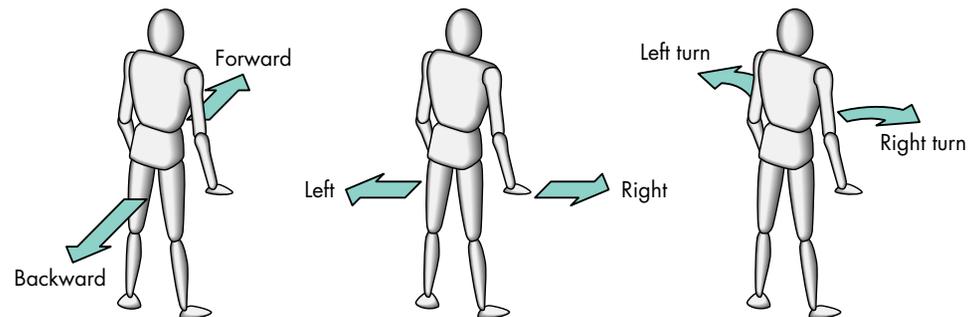


Figure 1. Ground movements

You can also go airborne with a variety of camera movements. These fancier camera moves are changes that might be typical of an aircraft. They include ascending and descending (moving upward and downward), pitching (tilting) up and down, and rolling (tilting) left and right. Figure 2 illustrates these moves.

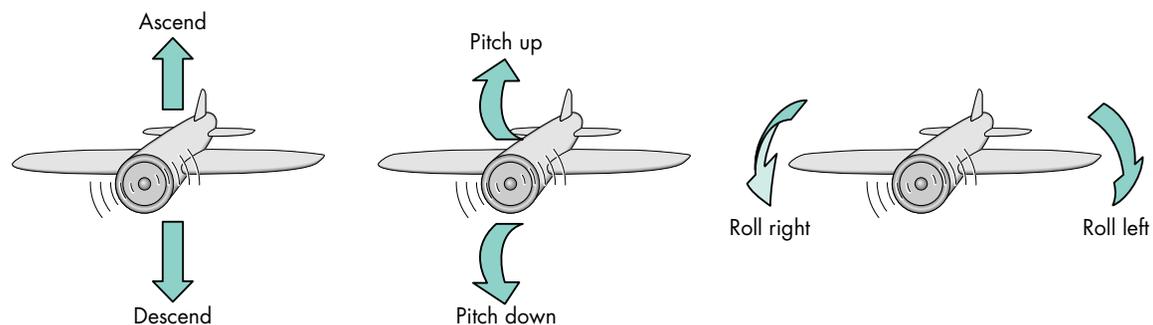


Figure 2. Air movements

Now to the fun part — let's get that camera moving! What you must do to achieve the previously described camera movements, both ground and air, involves some geometry. If you're like most of us and have forgotten your 3D geometry, see "3D Geometry 101" for a refresher course. The 3D geometry for our camera moves is quite simple; it will stick to the kinds of calculations illustrated in "3D Geometry 101."

First, let's take a look at our world. In Figure 3, we have an object in the world coordinate system and a camera looking at the object. The camera has its own coordinate system defined by its location (in world coordinates), up vector, and point of interest.

We'll be dealing with the vectors making up the camera's coordinate system for many of our movement functions, so let's keep these in our application's document structure. We'll keep the camera placement data there as well.

The document structure looks like this:

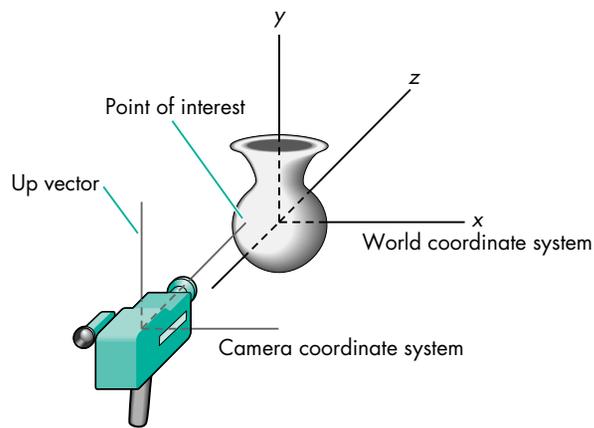


Figure 3. Our world

```
typedef struct _DocumentRecord {
    ...
    TQ3Point3D    cameraLocation;
    TQ3Point3D    pointOfInterest;
    TQ3Vector3D   xVector;
    TQ3Vector3D   yVector;    // up vector
    TQ3Vector3D   zVector;
    ...
} DocumentRecord, *DocumentPtr;
```

The first time we set up our camera, we'll set the values in our document to correspond to the initial camera position. Then with each subsequent movement of the camera, we'll update these fields. The initial camera data is constructed by the code in Listing 1. In the function `MyGetCameraData`, we do some of our geometric calculations to get the x and z vectors. We subtract the two endpoints (the initial and final points) of the z vector to get that vector. And we get the x vector by cross-multiplying the y and z vectors.

Listing 1. Initializing the camera data

```
void MyGetCameraData(DocumentPtr theDocument, TQ3CameraObject theCamera)
{
    TQ3CameraPlacement    cameraPlacement;

    // Get the camera data.
    Q3Camera_GetPlacement(theCamera, &cameraPlacement);

    // Set the document's camera data.
    theDocument->cameraLocation = cameraPlacement.cameraLocation;
    theDocument->pointOfInterest = cameraPlacement.pointOfInterest;
    theDocument->yVector = cameraPlacement.upVector;

    // Calculate the x and z vectors and assign them to the document.
    Q3Point3D_Subtract(&theDocument->pointOfInterest, &theDocument->cameraLocation,
        &theDocument->zVector);
    Q3Vector3D_Cross(&theDocument->zVector, &theDocument->yVector, &theDocument->xVector);
}
```

3D GEOMETRY 101

If you're new to 3D programming (and perhaps a little rusty on your math), here's a brief introduction to some of the 3D concepts you'll find in this article's code.

A *point* is represented in 3D space by x , y , and z values in a coordinate system. A *vector* is a magnitude (length) and direction; it's represented by an initial point (usually the origin of the coordinate system) and a final point $\{x, y, z\}$. Figure 4 illustrates a point and a vector in 3D space.

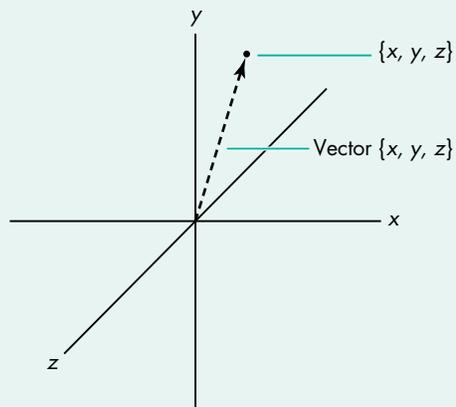


Figure 4. A point and a vector in 3D space

To add a vector and a point, you place the vector's initial point on that point (keeping the vector's direction and magnitude). The new final point of the moved vector is the point resulting from the addition. (See Figure 5.)

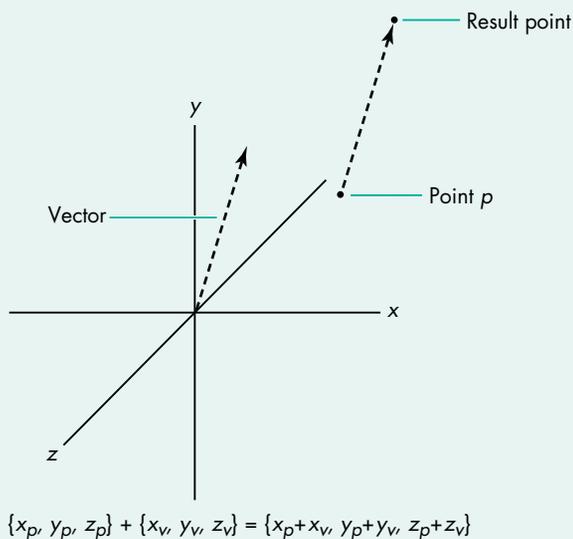


Figure 5. Adding a vector and a point

To subtract a vector from a point, you place the vector's final point on that point (keeping the vector's direction and magnitude). The new initial point of the moved vector is the result (Figure 6).

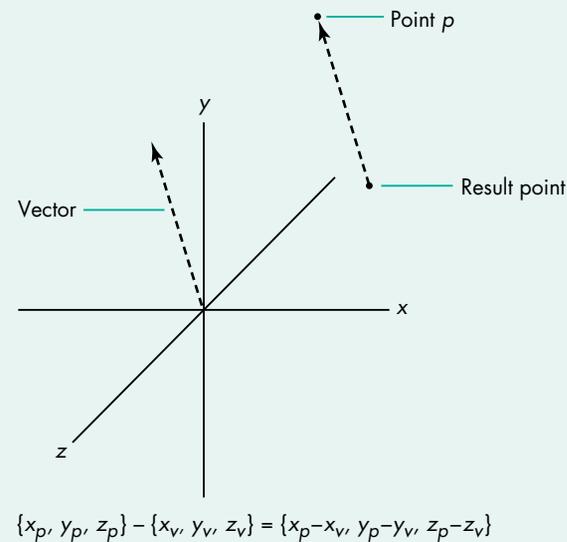


Figure 6. Subtracting a vector from a point

To create a vector between two points, you subtract the vectors defined by the points (called *position vectors*). To do this, you first reverse (turn around) the second vector and place its initial point on the final point of the first vector. Then you make a new vector from the first vector's initial point to the second vector's new final point. This new vector has the direction and magnitude of the vector between the two points (Figure 7).

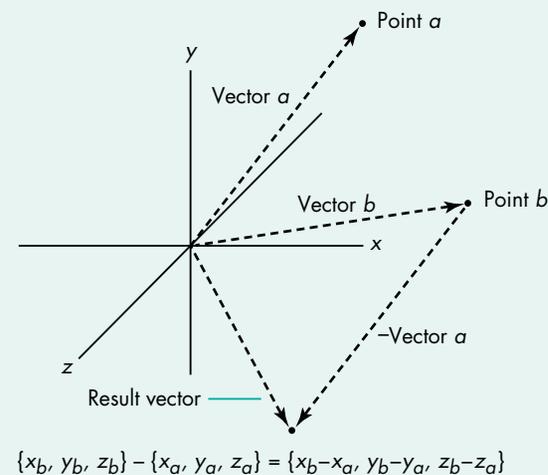


Figure 7. Creating a vector between two points

A translation of a point or a vector by T_x , T_y , and T_z values moves the point or the vector by adding the T values to its own values (Figure 8).

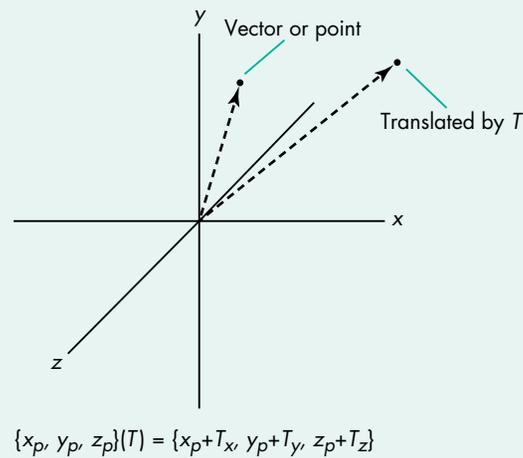


Figure 8. Translating a point or a vector by T

In Figure 8, the translation value T is really from the translation part of a transformation matrix. A *transformation matrix* is used to transform a point or a vector by translation, rotation, and scaling. The transformation matrix you use is 4×4 — with the upper-left 3×3 portion acting as the rotation matrix, the bottom-left 1×3 portion acting as the translation matrix, and the top-left to bottom-right diagonal of the rotation matrix acting as the scaling matrix. The following transformation matrix has elements labeled for translation (T), rotation (R), and scaling (S). The fourth column is ignored for simplicity.

$$\begin{Bmatrix} S_x * R_{0,0} & R_{0,1} & R_{0,2} & 0 \\ R_{1,0} & S_y * R_{1,1} & R_{1,2} & 0 \\ R_{2,0} & R_{2,1} & S_z * R_{2,2} & 0 \\ T_x & T_y & T_z & 1 \end{Bmatrix}$$

When you apply a transformation to a point or a vector, you multiply by the matrix, as in the following formula for our point $\{x, y, z\}$ and a transformation matrix:

$$\begin{Bmatrix} \{S_x * R_{0,0} * x + R_{1,0} * y + R_{2,0} * z + T_x\}, \\ \{R_{0,1} * x + S_y * R_{1,1} * y + R_{2,1} * z + T_y\}, \\ \{R_{0,2} * x + R_{1,2} * y + S_z * R_{2,2} * z + T_z\} \end{Bmatrix}$$

As you can see from this formula, if you only want the matrix to apply a translation (the T 's), the 3×3 rotation

matrix will be all 0's except for the scaling diagonal, which will be all 1's.

A rotation of a vector through an arbitrary angle about different axes will use various R elements (the 3×3 rotation matrix of the transformation matrix), depending on which axis you're rotating about. For rotations of θ about the x axis, you get the matrix

$$\begin{Bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{Bmatrix}$$

For rotations about the z axis, you get

$$\begin{Bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{Bmatrix}$$

And for rotations about the y axis, you get the following matrix:

$$\begin{Bmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{Bmatrix}$$

So to apply a rotation about an axis, you simply multiply the appropriate rotation matrix by the vector. In Figure 9, the vector on the right is rotated 90° about the z axis in the $\{x, y\}$ plane.

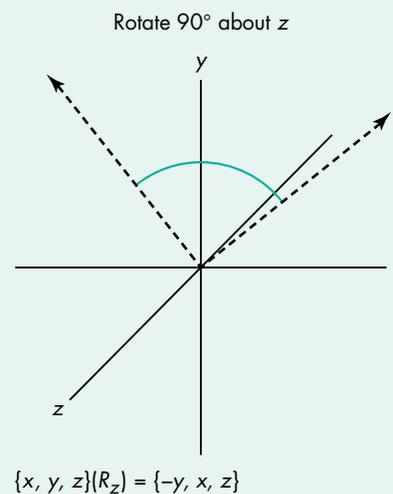


Figure 9. Rotating a vector about an axis

After the fields in our document have been updated by some camera movement function, we'll want to reset the camera to that new data with the function `MySetCameraData` (Listing 2).

Listing 2. Setting the camera data after a move

```
void MySetCameraData(DocumentPtr theDocument, TQ3CameraObject theCamera)
{
    TQ3CameraPlacement cameraPlacement;

    // Set the camera placement data.
    cameraPlacement.cameraLocation = theDocument->cameraLocation;
    cameraPlacement.pointOfInterest = theDocument->pointOfInterest;
    cameraPlacement.upVector = theDocument->yVector;

    // Set the camera data to the camera.
    Q3Camera_SetPlacement(theCamera, &cameraPlacement);
}
```

With that camera infrastructure, we're ready to move the camera around a bit. You can find the code for all the moves on this issue's CD. Here you'll find only the code for those movements that are unique. Code for those moves not shown (but previously mentioned) is almost identical to one of the functions shown in the listings.

To move the camera along the z axis either forward or backward, we call the function `MyMoveCameraZ` (Listing 3). This function translates the camera location and point of interest by the given delta. Note that the associated z vector isn't changed.

To move the camera along the x axis (right or left) or along the y axis (ascending or descending), you use code similar to Listing 3. The only difference is that you base the translation on the change in x or y instead of the change in z . In both cases, the associated vectors don't change.

Next, to rotate the camera right or left about the y axis, we call the function `MyRotateCameraY` (Listing 4). This function first creates a transformation matrix whose rotation matrix represents rotating about the y axis. It then transforms both the z and x vectors by that rotation (thus rotating those two vectors about the y axis). From the rotated z vector, we obtain the point of interest by adding the camera location to the vector.

Rotating the camera about the x axis (pitching up or down) or about the z axis (rolling left or right) is similar to rotating it about the y axis. The main difference is in how the rotation matrix is constructed (from the axis in question) and which axes are rotated (the other two). The only other difference is that when rotating the camera about the z axis, you don't have to update the point of interest because it doesn't change.

SMOOTH SAILING

To see what we've done to our world, we need a rendering loop, which you'll find in the code on the CD. Since we don't do anything special in our rendering loop, we'll skip the details. For an explanation of rendering loops, see the article "QuickDraw 3D: A New Dimension for Macintosh Graphics" in *develop* Issue 22.

Listing 3. Moving the camera along the z axis

```
void MyMoveCameraZ(DocumentPtr theDocument, float dZ)
{
    TQ3ViewObject    theView;
    TQ3CameraObject  theCamera;
    TQ3Vector3D      scaledVector;
    TQ3Point3D       newPoint;

    // Get the view and the camera objects.
    theView = theDocument->theView;
    Q3View_GetCamera(theView, &theCamera);

    // Scale the z vector to make it dZ longer.
    Q3Vector3D_Scale(&theDocument->zVector,
                    dZ/Q3Vector3D_Length(&theDocument->zVector), &scaledVector);
    // Move the camera position and direction by the new vector.
    Q3Point3D_Vector3D_Add(&theDocument->cameraLocation, &scaledVector,
                          &newPoint);
    theDocument->cameraLocation = newPoint;
    Q3Point3D_Vector3D_Add(&theDocument->pointOfInterest, &scaledVector,
                          &newPoint);
    theDocument->pointOfInterest = newPoint;

    // Set the updated camera data to the camera.
    MySetCameraData(theDocument, theCamera);

    // Update the view with the changed camera and dispose of the camera.
    Q3View_SetCamera(theView, theCamera);
    Q3Object_Dispose(theCamera);
}
```

Listing 4. Rotating the camera about the y axis

```
void MyRotateCameraY(DocumentPtr theDocument, float dY)
{
    TQ3ViewObject    theView;
    TQ3CameraObject  theCamera;
    TQ3Vector3D      rotatedVector;
    TQ3Matrix4x4     rotationMatrix;

    // Get the view and the camera objects.
    theView = theDocument->theView;
    Q3View_GetCamera(theView, &theCamera);

    // Create the rotation matrix for rotating about the y axis.
    Q3Matrix4x4_SetRotateAboutAxis(&rotationMatrix,
                                   &theDocument->cameraLocation, &theDocument->yVector, dY);

    // Rotate the z vector about the y axis.
    Q3Vector3D_Transform(&theDocument->zVector, &rotationMatrix,
                        &rotatedVector);
}
```

(continued on next page)

Listing 4. Rotating the camera about the y axis (*continued*)

```
theDocument->zVector = rotatedVector;

// Rotate the x vector about the y axis.
Q3Vector3D_Transform(&theDocument->xVector, &rotationMatrix,
    &rotatedVector);
theDocument->xVector = rotatedVector;

// Update the point of interest from the new z vector.
Q3Point3D_Vector3D_Add(&theDocument->cameraLocation,
    &theDocument->zVector, &theDocument->pointOfInterest);

// Set the updated camera data to the camera.
MySetCameraData(theDocument, theCamera);

// Update the view with the changed camera and dispose of the camera.
Q3View_SetCamera(theView, theCamera);
Q3Object_Dispose(theCamera);
}
```

The real issue for us in viewing our camera movements is how smooth and fast those moves appear. The factors that determine how smoothly and quickly the moves work are the sizes (scales) of the deltas (the arguments to the movement functions) and the speed of the machine (and therefore the subsequent speed of the rendering loop). Adjusting for the speed of the machine is beyond the scope of this article.

The sizes of the deltas determine the size of the jumps taken by each camera movement. If the deltas are very small, the camera will move very slightly. And if these movements are repeated, the camera will appear to move slowly over time. If the deltas are large, the camera will appear to move fast.

If you move the camera too slowly, the movement will appear jumpy because the user will see the delays in rendering time. If you move the camera too fast, the movement will appear jumpy because, well, you're making the camera take big jumps. To find just the right speed, you need to experiment with the sizes of the deltas. The main thing to notice is that you should correlate the deltas to the size of the model.

Listing 5 shows how you might set up the delta multipliers (called *factors* here) that are used to help control movement. From the model's bounding box, the `MyInitDeltaFactors` function determines the size of the largest dimension. This model size is then used to generate the various factors for different movement functions. Since accelerating the movements (say, by a control key) is quite useful, this function sets that up too.

Your mileage may vary, so it's a good idea to take your camera out for a spin and see what factors work for your application.

CONTROLLING THE CONTROLS

Now that you have the means of moving the camera this way and that, you need to have something controlling those movements. Our application will use the keyboard and the mouse.

Listing 5. Creating delta factors based on the model's dimensions

```
void MyInitDeltaFactors(DocumentPtr theDocument)
{
    TQ3BoundingBox      viewBBox;
    TQ3Vector3D         diagonalVector;
    float               maxDimension;

    // Get the bounding box and find the scene dimension.
    MyGetBoundingBox(theDocument, &viewBBox);
    Q3Point3D_Subtract(&viewBBox.max, &viewBBox.min, &diagonalVector);
    maxDimension = Q3Vector3D_Length(&diagonalVector);

    // Now set the delta factors.
    theDocument->xRotFactor = kXRotFactorBase * maxDimension;
    theDocument->yRotFactor = kYRotFactorBase * maxDimension;
    theDocument->zRotFactor = kZRotFactorBase * maxDimension;
    theDocument->xMoveFactor = kXMoveFactorBase * maxDimension;
    theDocument->yMoveFactor = kYMoveFactorBase * maxDimension;
    theDocument->zMoveFactor = kZMoveFactorBase * maxDimension;

    // Set up the control factor.
    theDocument->controlFactor = kControlFactorBase * maxDimension;
}
```

To take input from the keyboard or the mouse, or both, we don't do anything unusual. For the keyboard, we take the key-down events as they happen and determine whether any other keys were held down at the time of the event (for multiple key inputs). For the mouse, we just continually track it.

In both cases, the user can indicate movement along more than one dimension. For example, if moving the mouse forward means "forward" and moving the mouse left means a combination of "turn left" and "roll left," a mouse movement that's both forward and to the left is a combination of three camera movements.

Based on whether the user input is simple or complex, our code makes calls to the appropriate camera movement functions. In the case of the mouse, the speed of the mouse (the difference between the last position and the current position) is also used to adjust the deltas for the camera movement. Listing 6 shows the code used for mouse tracking, but without the error handling and some details of GWorlds and local coordinates (see this issue's CD for the full source code). Here we've hard coded the meanings of the different mouse movements and control keys for simplicity. Ideally, you would have this stored in preference data that the user can set.

The code for handling keyboard input is even simpler. See the CD for that part of the code.

Many other input devices are also applicable, especially 3D input devices. The proper way to handle such input devices is through the QuickDraw 3D Pointing Device Manager with its controllers and trackers. To use this approach, we would need to define a tracker for our camera and assign it to the available controllers. We would also change the camera movement functions so that they took deltas of both position and orientation. See the book *3D Graphics Programming With QuickDraw 3D* and the Graphical Truffles column "Making the Most of QuickDraw 3D" in *develop* Issue 24

Listing 6. Tracking the mouse

```
void MyDoMouseMove(WindowPtr theWindow, EventRecord *theEvent)
{
    DocumentPtr    theDocument;
    Point          newMouse;
    long           dx, dy, oldX, oldY;
    float          xRot, yRot;
    short          usingControl = false;

    // Get the document from the window.
    theDocument = MyGetDocumentFromWindow(theWindow);

    // Get the current mouse position.
    GetMouse(&newMouse);
    oldX = newMouse.h;
    oldY = newMouse.v;

    // If the control key is down, we're in depth mode.
    if (theEvent->modifiers & controlKey)
        usingControl = true;

    // Loop, moving the camera while the mouse is down.
    while (StillDown()) {
        // Get the next mouse position.
        GetMouse(&newMouse);

        // Calculate the difference from the last mouse position.
        dx = newMouse.h - oldX;
        dy = oldY - newMouse.v;

        // If there's some difference, move the camera.
        if ((dx != 0) || (dy != 0)) {
            // Calculate the rotation about the y axis (pan) and rotate.
            yRot = ((float) dx * (kQPi / 180.0)) / theDocument->width;
            MyRotateCameraY(theDocument, -yRot * theDocument->yRotFactor);

            // If the control key is down, move along the z axis; otherwise, rotate about the x axis.
            if (usingControl) {
                // Move the camera along the z axis (change in mouse's y).
                MyMoveCameraZ(theDocument, dy * theDocument->zMoveFactor);
            } else {
                // Calculate the rotation about the x axis (pitch) and rotate.
                xRot = ((float) dy * (kQPi / 180.0)) / theDocument->height;
                MyRotateCameraX(theDocument, xRot * theDocument->xRotFactor);
            }
            // Update the screen for each move.
            MyUpdateScreen(theDocument);
        }
        // Set the current mouse position as the old mouse position for the next update.
        oldX = newMouse.h;
        oldY = newMouse.v;
    }
}
```

for more on controllers and trackers. (As of now, QuickDraw 3D doesn't have built-in controllers for the mouse and the keyboard, so this code handles them directly.)

AIMING FOR EFFICIENCY

To make the geometry and the code for this article clearer, some efficiency issues were ignored. But for most applications, the time spent in moving the camera will be minimal when compared to the time spent rendering and displaying each frame.

However, if the time used for the rendering-rastering phase is minimal and the camera movements use a more significant percentage of the total time, there are a number of solutions. The ultimate efficiency solution is to avoid making any multiplications or divisions in the camera movements by using finite differencing techniques when calculating the moves. This strategy involves keeping more information about each intermediate change and making only the incremental calculations necessary for the next move. This approach is similar to operator reductions in compilers.

MAKING YOUR NEXT MOVE

A number of applications can use game controls like those discussed here, not just first-person 3D games. Another application that's a good candidate for the kinds of game controls presented here would be a 3D Internet browser. You would want similar 3D controls, but you would also want some controls for selecting Web hot spots that would take you to another 3D Web site.

So now the next move is up to you.

RECOMMENDED READING

- "QuickDraw 3D: A New Dimension for Macintosh Graphics" by Pablo Fernicola and Nick Thompson, *develop* Issue 22.
- "Graphical Truffles: Making the Most of QuickDraw 3D" by Nick Thompson and Pablo Fernicola, *develop* Issue 24.
- *3D Graphics Programming With QuickDraw 3D* by Apple Computer, Inc. (Addison-Wesley, 1995).
- *Mathematical Elements for Computer Graphics*, 2nd Edition, by David F. Rogers and J. Alan Adams (McGraw-Hill, 1990).
- *Tricks of the Mac Game Programming Gurus* by Jamie McCornack and others (Hayden Books, 1995).

Thanks to our technical reviewers Rick Evans, Richard Lawler, John Louch, Tim Monroe, Nick Thompson, and Dan Venolia. •



DANIEL I. LIPTON

GRAPHICAL TRUFFLES

A Library for Traversing Paths

The QuickDraw GX graphics system is based on shape objects that are used by reference. An application creates a shape object such as a path or a polygon by passing in data that represents the geometric points of the shape to be drawn or otherwise manipulated. The QuickDraw GX graphics system then stores this information in its internal database and returns to the application a reference to the shape object. This reference is then passed to the various QuickDraw GX routines that perform operations on the shape.

Since the QuickDraw GX graphics system is maintaining the original data in its database, the application often won't keep this data around. Also, an application may not even have created the geometric points in the first place, as in the case of converting text into a path.

It's often desirable, for a variety of purposes, for an application to retrieve the geometric information from a shape object. Given the richness of geometric information that these objects can contain, it can be a nontrivial task to read back the information. This column describes a C library that any application can incorporate for the purpose of traversing the geometric information in QuickDraw GX paths.

WHAT'S IN A QUICKDRAW GX PATH OBJECT

The QuickDraw GX graphics system provides several types of graphics primitives with which to create visual content: lines, curves, polygons, paths, typography, and bitmaps. In this system, curves are quadratic Béziers that can be defined by three control points, the middle point being off the curve and the other two being on. Figure 1 depicts a single quadratic curve segment.

A QuickDraw GX path object is just a conglomeration of curve and line segments, resulting from an array of

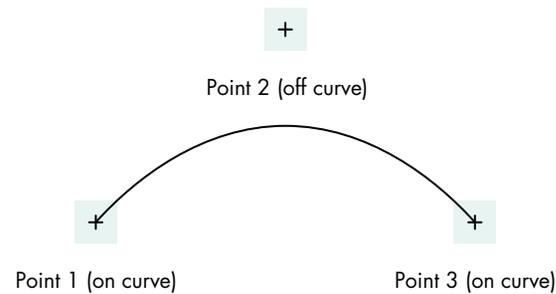


Figure 1. A QuickDraw GX quadratic curve segment

points. The object can contain multiple contours, each contour being a group of connected segments.

The question arises, if we look at a specific point in a path structure, whether the point is part of a line segment or part of a curve segment. The answer is that in addition to the points themselves, a path contains an array of flags, one for each point, indicating whether the point is on or off the path. To represent a single contour for a path object, QuickDraw GX uses the `gxPath` data structure:

```
struct gxPath {
    long          vectors;
    long          controlBits[1];
    struct gxPoint vector[1];
};
typedef struct gxPath gxPath;
```

The `vectors` field is an integer that specifies the number of points in the contour, the `controlBits` field is a bit array representing the on-curve/off-curve flags, and the `vector` field is an array of points for the contour.

To represent a path object, QuickDraw GX uses the `gxPaths` data structure:

```
struct gxPaths {
    long          contours;
    struct gxPath contour[1];
};
typedef struct gxPaths gxPaths;
```

The `contours` field is an integer specifying the number of contours, and the `contour` field is an array of `gxPath` structures, one for each contour.

Hence we have enough information to figure out what the points mean. If we see two on-path points in a row, we know that represents a line. If we see an on-path

DANIEL I. LIPTON (daniel_lipton@powertalk.apple.com) has worked at Apple for seven years and is one of the original

QuickDraw GX team members. In his spare time, Dan runs a small business repairing perpetual motion machines. •

point followed by an off-path point followed by an on-path point, we know that's a quadratic curve segment.

So to read the QuickDraw GX path object to determine the actual shape, all we have to do is get a point and then get the next point. According to the previous description, it would be safe to assume that the very first point in a contour is an on-path point. Then, if the next one were also on the path, we'd have a line; if it were off, we'd know that we'd have to read a third one (which by definition would have to be on) and we'd have a curve.

The only trouble is that those assumptions aren't necessarily true. The design of QuickDraw GX could have restricted applications to using only those patterns of on-path/off-path points, disallowing two consecutive off-path points and requiring the first point and the last point in a contour to be on the path; however, it didn't.

In the interest of saving memory, QuickDraw GX allows two consecutive off-path points to imply a middle on-path point — known as an *implicit point* — exactly halfway between the off-path points. An example of this is shown in Figure 2.

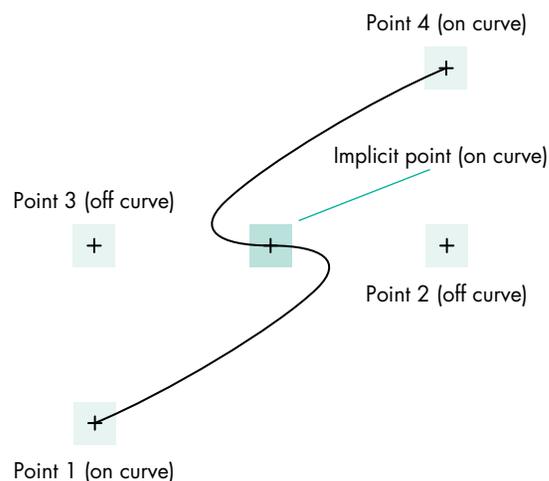


Figure 2. A QuickDraw GX quadratic path

For each implicit point there's a memory savings of 8 bytes in QuickDraw GX. This allows us to define geometries in less space than would be required in other popular graphics models that use cubic Bézier curves without implicit points, but it does complicate traversing the path.

QuickDraw GX also allows the first or the last point of a contour to be off the curve, in the case where the contour is closed. This further complicates path traversal.

THE SHAPEWALKER LIBRARY

You can use the ShapeWalker library (which is included on this issue's CD) to avoid having to write a huge blob of code to deal with all those points and flags discussed above. It allows an application to pass in a QuickDraw GX shape object and be sent back (via callbacks) each line and curve segment in the shape. All implicit points are resolved by the library, so the client sees only complete line or curve segments.

The header file to be used with the library defines types for four callbacks and a prototype for the ShapeWalker function:

```
// Function is called to move to a new point
// (start new contour).
typedef Boolean (*TpwMovetoProc)(gxPoint *p,
                                void* refcon);

// Function is called to draw a line from
// current point to p.
typedef Boolean (*TpwLinetoProc)(gxPoint *p,
                                void* refcon);

// Function is called to draw a curve from
// current point (which will be p[0]) through
// p[1] to p[2].
typedef Boolean (*TpwCurvetoProc)(gxPoint p[3],
                                   void* refcon);

// Function is called to close a contour.
typedef Boolean (*TpwClosepathProc)
    (void* refcon);

// Return result will be true if path walking
// was terminated by one of the callbacks.
Boolean ShapeWalker(gxShape theShape,
                   TpwMovetoProc DoMoveto,
                   TpwLinetoProc DoLineto,
                   TpwCurvetoProc DoCurveto,
                   TpwClosepathProc DoClosepath,
                   void* refcon);
```

When using the library, you provide four callbacks and a refcon. Each callback will get passed the refcon and possibly point information. It's suggested that the client maintain whatever state information is necessary for the purpose at hand. The refcon can be a pointer to a structure containing the state information. One typical component of such information that most clients would need is the notion of the current point. The current point is the piece of the path we've looked at most recently, representative of the state of processing the shape. This current point should be updated as segments come through the callbacks. (We'll see this in a moment in our sample application.)

Each callback must also return a result of type Boolean, giving the client a mechanism for causing the library to terminate traversal of the shape before completion. Return false and the shape walker will continue on to the next segment; return true and it will terminate early. This can be used to catch errors in processing the points, or to terminate processing if you've finished with the shape before the last point is reached.

The four callbacks are as follows:

- **DoMoveto** — This procedure is called at the start of each new contour in the shape. It gets passed a single point and the refcon. The point identifies the location of the beginning of the contour. If the client is maintaining a current point via the refcon, it should be updated to the point passed in.
- **DoLineto** — This procedure is called for each line segment in the contour. It gets passed a single point and the refcon. The point represents the end point of the line segment. The start point of the line segment is whatever point we last saw; that will be

the current point if one is being maintained. If the client is maintaining a current point via the refcon, it should be updated to the point passed in.

- **DoCurveto** — This procedure is called for each quadratic curve segment in the contour. It gets passed an array of three points and the refcon. The three points correspond to the control points of the curve. The first point in the array will be the current point if one is being maintained. The current point should then be updated to reflect the third point in the array.
- **DoClosepath** — This procedure is called at the end of every contour if the QuickDraw GX shape is closed (has the `gxClosedFrameFill` shape fill attribute). Closing a contour implies connecting the last point in the contour (whatever the current point is when this function is called) with the first point in the contour (the point passed into the `DoMoveto` function).

The code shown in Listing 1 is a sample application (`SamplePathWalker.c`) that converts a piece of text to a path and then uses the `ShapeWalker` library to read the

Listing 1. Sample application using the `ShapeWalker` library

```
// The following structure is used to maintain a state while walking a shape.
typedef struct {
    gxPoint    currentPoint;    // current point
    gxPoint    firstPoint;     // first point in contour
} TestWalkRec;

#define fix2float(x) ((double)x / 65536.0)

Boolean TestMoveto(gxPoint *p, TestWalkRec* pWalk);
Boolean TestMoveto(gxPoint *p, TestWalkRec* pWalk)
{
    printf("Begin new contour: %f, %f\r\n", fix2float(p->x), fix2float(p->y));
    pWalk->currentPoint.x = p->x;
    pWalk->currentPoint.y = p->y;
    pWalk->firstPoint.x = p->x;
    pWalk->firstPoint.y = p->y;
    return (false);
}

Boolean TestLineto(gxPoint *p, TestWalkRec* pWalk);
Boolean TestLineto(gxPoint *p, TestWalkRec* pWalk)
{
    printf("Line from %f, %f to %f, %f\r\n", fix2float(pWalk->currentPoint.x),
          fix2float(pWalk->currentPoint.y), fix2float(p->x), fix2float(p->y));
    pWalk->currentPoint.x = p->x;
    pWalk->currentPoint.y = p->y;
    return (false);
}
```

(continued on next page)

Listing 1. Sample application using the ShapeWalker library (*continued*)

```
Boolean TestCurveto(gxPoint p[3], TestWalkRec* pWalk);
Boolean TestCurveto(gxPoint p[3], TestWalkRec* pWalk)
{
    printf("Curve from %f, %f through %f, %f, to %f, %f\r\n", fix2float(p[0].x), fix2float(p[0].y),
           fix2float(p[1].x), fix2float(p[1].y), fix2float(p[2].x), fix2float(p[2].y));
    pWalk->currentPoint.x = p[2].x;
    pWalk->currentPoint.y = p[2].y;
    return (false);
}

Boolean TestClosepath(TestWalkRec* pWalk);
Boolean TestClosepath(TestWalkRec* pWalk)
{
    printf("Closing the contour\r\n\r\n");
    pWalk->currentPoint.x = pWalk->firstPoint.x;
    pWalk->currentPoint.y = pWalk->firstPoint.y;
    return (false);
}

main()
{
    gxShape      theShape;
    gxPoint      location = {ff(100), ff(100)};
    TestWalkRec  walker;
    Boolean      result;

    theShape = GXNewText(5, "Hello", &location);
    GXSetShapeTextSize(theShape, ff(50));
    GXSetShapeType(theShape, gxPathType);
    GXSetShapeFill(theShape, gxClosedFrameFill);

    result = ShapeWalker(theShape, TestMoveto, TestLineto, TestCurveto, TestClosepath, &walker);
    GXDisposeShape(theShape);
}
```

points from the result. In this example the callback procedures are used only to print out the points in the segments, but of course they can be used to do a lot of other things as well.

WALKING THE PATH

The files PathWalking.h and PathWalking.c are all that are required to use the ShapeWalker library in your application (for the sake of brevity, PathWalking.c isn't shown in this column). This library should make it easy for your application to process QuickDraw GX path objects. For completeness, the library will also process

curve objects, line objects, rectangle objects, and polygon objects in a similar manner. All other shape types will result in the posting of the "illegal_type_for_shape" graphics error. (Graphics errors can be polled with the GXGetGraphicsError function.)

The ShapeWalker library is actually based on the same code used by QuickDraw GX in its built-in GX-to-PostScript translator for printing. The library's versatility means that its uses in your application are limited only by your imagination, so get creative and try it out!

Thanks to Dave Hersey, Ingrid Kelly, and Dave Polaschek for reviewing this column. •

Macintosh Q & A

Q *What books and articles would you recommend that provide strategies for debugging?*

A Here's a list of resources that can help you with debugging on the Macintosh:

- *How to Write Macintosh Software* by Scott Knaster and Keith Rollin (Addison-Wesley, 1992). This book describes how to find all the bugs you wrote when you used memory manipulation in C.
- *Debugging Macintosh Software With MacsBug*, by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991), and *MacsBug Reference and Debugging Guide* by Apple Computer, Inc. (Addison-Wesley, 1990). These books don't describe the latest version of MacsBug; check the MacsBug 6.5.2 release notes for additional details.
- "Macintosh Debugging: A Weird Journey Into the Belly of the Beast" by Bo3b Johnson and Fred Huxham, *develop* Issue 8, and "Macintosh Debugging: The Belly of the Beast Revisited" by Fred Huxham and Greg Marriott, *develop* Issue 13.
- "Debugging on PowerPC" by Dave Falkenburg and Brian Topping, *develop* Issue 17.
- "Balance of Power: MacsBug for PowerPC" by Dave Evans and Jim Murphy, *develop* Issue 22.
- "KON & BAL's Puzzle Page," in every issue of *develop* since Issue 9.

Q *I have a customer who's encountering a problem using my product. Can you suggest a way to use MacsBug to diagnose problems at a customer site?*

A Yes. Here, in a few easy steps, is a technique for using MacsBug to diagnose problems in the field:

1. Install a clean copy of the latest MacsBug.
2. Create a file using ResEdit (or Resorcerer, or whatever) containing an 'mxbm' resource (which contains MacsBug macro definitions) and install it into the MacsBug Preferences folder.
3. In this 'mxbm' resource, define the macro **everytime** to call the **stdloginto** macro as follows:

```
stdloginto 'Send to the programmer'
```

This way, if MacsBug is ever invoked due to a program error, a log of what occurred will be automatically generated. The log, named "Send to the programmer," will appear on the desktop.

4. Have your customer send you the log file created by the above steps.

See page 219 of the *MacsBug Reference and Debugging Guide* by Apple Computer, Inc. (Addison-Wesley, 1990) for details of the **everytime** macro. For details of what the **stdloginto** macro does, look at the 'mxbm' resource named "log stuff" in MacsBug's resource fork.

Q *We're developing an application that uses Apple Guide. It's working well on 680x0 Macintosh computers but is presenting a problem on the Power Macintosh, because of AppleGuideGlue. If we import this library as "weak," the program runs but crashes when we call any Apple Guide routines. If we import "strong," the program simply refuses to run. What can we do?*

A Linking with the .xcoff file produces a reference to a shared library named AppleGuideGlue. Unfortunately, the Apple Guide extension provides a library named AppleGuideGlueLib instead, so the reference isn't resolved and the application fails to launch.

The AppleGuideGlue.xcoff file has been changed to AppleGuideGlueLib.xcoff on the latest Mac OS SDK CD. You can use that one, or just rename the one you have before including it in your project.

In MPW, you can rename the library in the link process. If you're using Symantec C or C++ or CodeWarrior, however, the name of the file has to be correct for the matching library to be found at run time. Note that CodeWarrior ignores the ".xcoff" suffix if it's present in the filename, while Symantec must have the ".xcoff" suffix to properly include the file in the project.

Q *My QuickDraw GX printer driver has a 'ptyp' of "A4 portrait" as the default paper type (via the isDefaultPaperType flag). But when a user chooses my driver from the Page Setup dialog, A4 is selected as the default paper type in the desktop printer, though my driver has no 'ptyp' named A4. How can I set my own paper type (A4 portrait) as the default?*

A The paper-matching code is working incorrectly. QuickDraw GX internally adds the standard paper types (such as A4 and US Letter) to the options for your driver. The bug is that QuickDraw GX thinks it's finding a better fit for the current page dimensions than the assigned A4 portrait paper type. It then defaults to the internal A4 paper type.

The only workaround at this time is to remove the paper type that you're incorrectly defaulting to. If you're defaulting to a nonstandard paper type, such as Letterhead, Stationery, or Three-hole Punch, the best workaround is to remove that type from the Extensions folder. If you're defaulting to another paper type, the easiest thing you can do is to open your driver with a resource editor and remove or edit the 'ptyp' resource for the paper type that's incorrectly matching. (Open the resource and you'll see the paper type name embedded in the data.)

Q *I'm creating a QuickDraw GX page that contains a line of single-layer text shapes, with each word a different color. The page displays correctly when it's opened in SimpleText but shows a bug when it's printed to a PostScript printer: each line prints with one color instead of each word being a different color. Any ideas?*

A This is a bug that occurs only with single-layer text shapes that have a nil style in their face layer. There's a workaround that should be used anytime you do a one-layer text face, except for italics — this workaround slows down italic drawing but speeds up all other cases.

Create a "generic" style object (with GXNewStyle) to replace the nil style. Set the text size to 1.0 (important) and the pen to 0 in the style. The other fields are irrelevant to this fix. Set your text face's style to this "generic" style and the problem will disappear.

Q *I'm having a problem, apparent at very small font sizes (6 points and below), with the output quality of some fonts that emerge from a QuickDraw GX vector driver. My application uses gxLayouts for text display and editing. If I create my output using*

GXDrawShape to render the layout shapes, the small characters begin to look very crude: character height varies by about 30% between some letters, and curved letter forms degenerate to rough polygons. What can I do to improve the quality?

A Layouts (like all typographic shapes) have hints turned on by default. If the font you're using isn't hinted at small point sizes, using hints messes up the appearance of the text rather than helping it. Try using the layout shape and setting the `gxNoMetricsGridText` and `gxNoContourGridText` bits in the text attributes. The results at small sizes should be better.

Q *I'm writing an Open Transport client program, and I'm confused about how to perform an orderly release when I receive the T_ORDREL message. When I get the T_ORDREL message I'm supposed to call `OTRcvOrderlyDisconnect`. The documentation for `OTRcvOrderlyDisconnect` says that I can then continue to send data but that I can't read data without getting an "out of state" error (`kOTOutStateErr`). Is this correct?*

A Yes, it is. Your confusion is due more to the dynamics and subtleties of X/Open Transport Interface (XTI) programming than to Open Transport itself.

Let's examine an orderly disconnect situation. Assume that two nodes have an established TCP connection. Endpoint A has finished sending data and indicates closure by invoking an `OTSndOrderlyDisconnect` call (this translates into sending an end-of-file signal — FIN — over the wire). Endpoint B receives a T_ORDREL message. If, however, B hasn't finished receiving the data, B must continue until it gets back `kOTNoDataErr`. At this point, B initiates an `OTRcvOrderlyDisconnect` (which acknowledges A's FIN). This is known as a "half-close"; B can still send data to A (which will still receive T_DATA events), but if A attempts to send to B, A will receive an "out of state" error.

A, of course, should also continue accepting data until receiving `kOTNoDataErr`. A should then call `OTRcvOrderlyDisconnect`, thereby completing the other side of the link teardown. Both sides can then unbind.

If, however, either endpoint's network code is written such that T_ORDREL and T_DATA events are handled at different priorities (for instance, the T_ORDREL is handled at the notifier, but the T_DATA is deferred to SystemTask time), a race condition can occur. Your program must ensure that all data has been read before calling `OTRcvOrderlyDisconnect`.

There's also a subtlety of XTI programming that you should be aware of. It's possible that `OTSndOrderlyDisconnect` or `OTRcvOrderlyDisconnect` will return with a TLOOK error. This means that there's another event pending; your program must call `OTLook` to gather that event.

According to the XTI specification, the `OTSndOrderlyDisconnect` and `OTRcvOrderlyDisconnect` calls can fail because of a pending T_DISCONNECT event. XTI is trying to tell you that the connection to that endpoint broke. This can happen easily in our modern, wacky, asynchronous world of networks, and your program will have to call `OTRcvDisconnect` to acknowledge that your endpoint dropped.

Q *I've implemented a server endpoint that hands off the connection to a hand-off endpoint. After the server processes a connect request with the `OTAccept` call, the asynchronous handler for the hand-off endpoint is passed a T_DATA event. When the handler makes*

the OTRcv call, however, it returns error -3168 (kOTStateChangeErr). Can you explain this?

A This problem occurs only when there's a hand-off (secondary) endpoint involved. The way Open Transport is implemented, it's possible for an asynchronous hand-off endpoint to receive a T_DATA event before the connect mechanism is completed. After accepting a connection, an asynchronous listener endpoint can expect to receive a T_ACCEPTCOMPLETE call. The "accepting" or hand-off endpoint can expect to receive the T_PASSCON event.

It's possible for the hand-off endpoint to receive the T_DATA event before receiving the T_PASSCON event, and this apparently is what's happening to you. When this happens, set a flag to defer receiving the data until later. After the T_PASSCON event is received, check the flag and issue the OTRcv call if the flag is set. (Note that after deferring the handling of the T_DATA event, your handler won't receive this event again until you process all of the data presently available.)

Q *What's the relationship between the classic AppleTalk "self-send" variable and the one in Open Transport AppleTalk?*

A In version 1.1, Open Transport AppleTalk shares the self-send variable with classic AppleTalk, so if you set the variable with the classic PSetSelfSend call, the effects are seen by both AppleTalk and Open Transport clients. If you're using Open Transport, you can change the variable with an OTIoctl call, as shown here:

```
enum {
    kATalkFullSelfSend    = MIOC_CMD(MIOC_ATALK, 47)
};

static OSStatus OTSetSelfSend(EndpointRef ep, Boolean enable_self_send)
{
    OSStatus result;

    result = OTIoctl(ep, kATalkFullSelfSend, (void *) enable_self_send);
    if (result > 0)
        result = 0;
    return result;
}
```

Note that like the PSetSelfSend call, the OTIoctl call returns the previous value of the self-send variable as either 0 (it was previously disabled) or 1 (it was previously enabled). As in classic AppleTalk, it's rarely appropriate to restore the value of self-send when you're done, so the code above maps both results to 0 (noErr).

Here's why the value shouldn't be restored. The self-send value is a Boolean, not a counter. For example, imagine the following sequence:

1. Self-send starts out false.
2. Client A sets self-send to true and is returned false as the previous value.
3. Client B sets self-send to true and is returned true as the previous value.
4. Client A quits, "restoring" self-send to false.

In the end, client B is left with self-send set to false, which is incorrect.

For this reason, the standard practice is to set self-send if you need it and not attempt to restore it when finished. Because many clients follow this convention, it's important that your program work even if self-send is true.

Future versions of Open Transport will most likely have self-send always on for Open Transport native clients, and loop-back packets will be filtered out only for classic clients if PSetSelfSend wasn't called.

Q *When I make a synchronous OTConnect call from a TCP client to a TCP server that's passively awaiting an incoming connection, I find that even before the server responds with the OTListen and OAccept calls, the OTConnect call completes with no error. At this point, if I examine the client endpoint state, I find that it's in the T_DATAXFER state. Can you explain this?*

A As mentioned in the XTI specification (available with the Open Transport release), "TCP does not allow the possibility of refusing a connection indication. Each connect indication causes the TCP transport provider to establish the connection. Therefore t_listen() and t_accept() have a semantic which is slightly different than that for ISO providers." Consequently, the server will accept the TCP connection request if the current number of connections allows it. The XTI specification states that "when the transport detects a T_LISTEN, TCP has already established the connection." The client, whether in synchronous or asynchronous mode, will receive notice that the connection was established. For synchronous endpoints, TCP completes the three-way connection handshake. For asynchronous endpoints, the OTRcvConnect call must be made to complete the handshake.

Q *In my Open Transport TCP-based server application, I use a specific socket for receiving incoming connection requests. If I relaunch the server immediately after quitting, the initialization calls complete without error, but the server never receives any incoming connection requests. If I wait several minutes before relaunching the server, this problem doesn't occur. It appears that there's some internal timeout for disconnected connections. Is there a solution to this problem so that the server can be relaunched without waiting for the timeout?*

A TCP has a two-minute timeout on a binding after a connection has closed before the same port can be bound to again. This prevents stale data from corrupting a new connection. For this reason, you see a delay before you can successfully bind to the port again.

There's a way around this, using the IP_REUSEADDR option and the OTOptionManagement call. Set this option on all of your listening endpoints before you bind, and the problem should disappear.

Note that even after you use the IP_REUSEADDR option, at most one endpoint that's in a state less than connected (listening; unbound doesn't count) may be bound to a given port. Any number of connected or closing endpoints may be so bound to other unique ports, however.

The following sample shows how to set this option. The function takes two input parameters, the EndpointRef that you want to set the option for, and the state of the option that you want, typically true. The function returns a result of

OSStatus: if negative, it's the error returned from the OTOptionManagement call; if positive, it's the status field returned by OTOptionManagement (this means the call completed successfully but the status field had a value other than T_SUCCESS). If 0 (kOTNoError), then of course there was no error.

```
#include <OpenTransport.h>           // Open Transport files
#include <OpenTptInternet.h>
/* input:  reuseState (true: no delay, false: normal delay state)
   output: if result less than kOTNoError, it's the error returned by
   OTOptionManagement. Otherwise, the status value is returned as defined
   in OpenTransport.h:
       T_SUCCESS      = 0x020,    return kOTNoError if success
       T_FAILURE      = 0x040,
       T_PARTSUCCESS  = 0x100,
       T_READONLY     = 0x200,
       T_NOTSUPPORT   = 0x400
*/

OSStatus DoNegotiateIPReuseAddrOption(EndpointRef ep, Boolean reuseState)
{
    UInt8    buf[kOTFourByteOptionSize]; // Buffer for fourByte option
    TOption* opt;                        // Option ptr to make items
                                           // easier to access

    TOptMgmt req;
    OSStatus err;
    Boolean  isAsync = false;

    opt = (TOption*)buf;                // Set option ptr to buffer.
    req.opt.buf = buf;
    req.opt.len = sizeof(buf);
    req.opt.maxlen = sizeof(buf);        // We're using req for the
                                           // return result also.

    req.flags = T_NEGOTIATE;            // Negotiate for option dealing
    opt->level = INET_IP;                // with an IP-level function.
    opt->name = IP_REUSEADDR;
    opt->len = kOTFourByteOptionSize;
    *(UInt32*)opt->value = reuseState;   // Set the desired option
                                           // level, true or false.

    if (OTIsSynchronous(ep) == false) { // Check if ep is synchronous.
        isAsync = true;                  // Set flag if async.
        OTSetSynchronous(ep);           // Set endpoint to sync.
    }
    err = OTOptionManagement(ep, &req, &req);
    if (isAsync == true)                 // Restore ep state if necessary.
        OTSetAsynchronous(ep);

    // If no error, check the option status value.
    if (err == kOTNoError) {
        if (opt->status != T_SUCCESS)    // If not T_SUCCESS, return
            err = opt->status;           // the status.
    }
    return err;
}
```

 *I'm implementing a passive TCP connection. Can I hand off the connection to a different port address?*

A No, the hand-off connection endpoint must be bound to the same address as the endpoint that passed off the connection. This is an XTI requirement, as discussed in Appendix B of the XTI specification, Section B.3.

Q *I'd like my network client software to be able to abort an asynchronous OTConnect in progress — to allow a user, for example, to recover from an attempted connection to a nonexistent IP address. I've been calling OTSndDisconnect to abort it, but when I check the return code, I get a kOTOOutStateErr error. What gives?*

A Using an OTSndDisconnect is the proper way to abort an OTConnect in progress. After a successful call to OTConnect, the endpoint state will transition from T_IDLE to T_OUTCON. Calling OTSndDisconnect returns the endpoint state to T_IDLE.

You may be getting kOTOOutStateErr for one of the following reasons:

- The original OTConnect failed. Determine this by checking the OTConnect result.
- The connection broke and was asynchronously handled by your notifier. In this case, your endpoint would no longer be in the T_OUTCON state before you do the disconnect.

A good rule of thumb is always to confirm the endpoint state before doing the OTSndDisconnect to ensure that the endpoint isn't already disconnected.

Q *I have a question regarding T_DATA event handling for multiple active endpoints. Let's say I have two endpoints open, endpoint 1 and endpoint 2. Data arrives for endpoint 1, which then receives a T_DATA event. If data arrives for endpoint 2 before the data for endpoint 1 is read, it's my understanding that endpoint 2 won't get a T_DATA event until the data for endpoint 1 is read. Is that correct? In other words, does Open Transport queue multiple T_DATA events corresponding to multiple endpoints?*

A XTI or Open Transport endpoints are handled independently of each other. Whatever events are pending on one endpoint have (for the most part) no effect on any other endpoints.

Assume that endpoint 1 gets notified of a T_DATA event. Following this, a separate T_DATA event is queued up for endpoint 2. As soon as the notifier for endpoint 1 completes and returns to Open Transport, the notifier for endpoint 2 will be invoked. This behavior isn't contingent upon whether endpoint 1 processed the event, although of course endpoint 1 won't receive any more T_DATA events until its current T_DATA event is cleared. Keep in mind that waiting too long to process endpoint 1's T_DATA event will result in the exhaustion of buffers in the lower protocol layers.

Q *Given an AppleTalk network and the node address of a Macintosh, how can I remotely retrieve the Network Name specified in the Sharing Setup control panel?*

A The only universal way to determine a Macintosh's "flagship" name is to target an NBP lookup of type "workstation" to that particular node. At first glance, it would seem that we could get the desired result by calling PConfirmName (since it allows us to direct the NBP LkUp to the specific node by using the confirmAddr field, whereas PLookupName would broadcast it to an entire

zone). The PConfirmName call doesn't return the NBP Tuple information to the application, however: under classic AppleTalk, PConfirmName's sole purpose is to confirm or deny the existence of a registered NBP name. This leaves you with several alternatives.

Under classic AppleTalk, you have two options. The first option is to use the PLookupName call. This is a little complicated because PLookupName requires that you specify the "zone name" of the target node. You have to call GetZoneList and parse through the replies (illustrated in *Inside Macintosh: Networking*, page 4-7) to extract a list of zone names that correspond to your target's network number. (Note that if you're on an extended network, it's possible for an AppleTalk zone to have a range of network numbers.) Once you have a list of suspected network zones that the target is on, you can then direct a PLookupName to those zones and parse through the responses to find the one that matches your target's node address.

The second option under classic AppleTalk is to form the NBP LkUp packet yourself and send it via DDP. You can open and register your own DDP listener by using the POpenSkt call. You can then form your own NBP LkUp packet and transmit it to the target node's NBP listener socket (socket 2) with the PWriteDDP call. The target will respond to you with an NBP LkUp-Reply, which will cause your DDP socket listener to be called. You can parse the reply there.

Writing a DDP socket listener is tricky, but it's illustrated in the Network Watch (DMZ) sample provided on this issue's CD. Examine the doEcho function in the files dMZAT.c and SktListener.a. Writing a socket listener for the Power Macintosh can be challenging because of classic AppleTalk's 680x0 roots. If you're stuck with a classic AppleTalk system, however, this is the recommended approach.

If your code is written to run under Open Transport, you're in luck. You can specify the target address in the TLookupRequest data structure used by the OTLookupName function. Check out the DoSendLkUpReq function in DDPSample.cp, found on any Open Transport SDK CD. Since the programming model is so much simpler, you may want to investigate the Open Transport approach.

Q *I need to get the full pathname to a document in a callback where the only relevant piece of information I have is the WindowPtr for the window that contains the document. I can get the filename from the window title, but I don't know the directory ID or volume reference number. Is there any way to obtain the dirID and vRefNum from the WindowRecord?*

A No, there's no way to extract the file system information you need from a WindowRecord. A WindowRecord includes only structural human interface information (which might include the filename as the window's title) and has no intrinsic tie to any file on the disk. As you've implied, you must have the directory ID and volume reference number to extract a full pathname. Once you know the vRefNum and the parent dirID, you'll be able to use one of the full path routines in the sample code MoreFiles on this issue's CD to construct a full pathname.

If the file is open and you have the refNum for its access path, you can call PBGetFCBInfo to get the vRefNum and dirID (and then use them to get the

pathname). For more information on PBGetFCBInfo, see *Inside Macintosh: Files*, pages 2-237 through 2-238.

Q *We sell a game that incorporates our own proprietary 3D technology. We'd like to be able to take advantage of 3D acceleration hardware, if it's available on the user's system. What do you recommend we use?*

A Use the Rendering Acceleration Virtual Engine (RAVE) application development interface, which defines an abstract standardized hardware interface for applications to communicate with and control 3D hardware. If you adapt your game to draw to RAVE, it'll be compatible with any hardware 3D accelerators with RAVE-compatible drivers. RAVE is also a cross-platform specification, so code you write for the Macintosh will be easier to port to Windows 95 or Windows NT (if at some point in the future RAVE is implemented for Windows).

Q *We created a QuickDraw 3D application similar to the TextureEyes demo distributed by Apple: it maps a moving image texture onto a spinning cube. The display quality of TextureEyes, however, is much better than ours. We're using large high-quality textures (480 x 320), but the image mapped onto the cube is quite chunky even with a 3D accelerator card, and the animation seems to be slower and jerkier. What's TextureEyes' secret?*

A No secrets: TextureEyes is a straightforward implementation of the texturing of QuickDraw 3D geometries. The problem is that the quality of your textures is actually too high!

QuickDraw 3D uses a trilinear MIP map algorithm to obtain the best possible quality texture mapping. To create an MIP map from an image requires creating subimages sized for every inverse power of 4 — that is, 1/4, 1/16, 1/64, and so on. The process of creating an MIP map for every texture takes time, and larger textures take longer. TextureEyes uses a 128 x 128 source for its movie and video textures. For more information on MIP maps, see *Computer Graphics: Principles and Practice*, by Foley, VanDam, Feiner, and Hughes (Addison-Wesley, 1996).

Q *My quartz watch is eerily accurate. Why?*

A A quartz watch uses the vibrations of a quartz crystal as its time reference. The frequency of vibration in the crystal depends on three factors: voltage, pressure, and temperature. To keep the watch accurate, all three of these must be kept constant.

The electronics in the watch provide a nice constant voltage, and the atmosphere provides a nice constant pressure. But what really ensures that the watch is accurate is that it's worn on your wrist: the constant temperature of your body in contact with the watch ensures that the crystal operates at a constant temperature.

These answers are supplied by the technical gurus in Apple's Developer Support Center. For more answers, see the Macintosh Technical Q&As on this issue's CD or on the World Wide Web at

<http://dev.info.apple.com/techqa/Main.html>. (Older Q&As can be found in the Macintosh Q&A Technical Notes on the CD.)[•]



DAVE JOHNSON

THE VETERAN NEOPHYTE

Your Friend the Drill Sergeant

There are a ton of different ways to learn to shoot pool. You can just bash the balls around, trying to pocket them, and eventually you'll get better at it. You can play games with other people, which increases the motivation somewhat, and probably learn a little faster. (Some people claim you should always play for money, because it makes it matter so much more.) But one of the most powerful ways to practice pool is plain old drill: setting up the same situation over and over, trying to make the shot a little better, a little more accurate, every time. Concentrated, repetitive drill is incredibly helpful in the early stages of learning the game, but it doesn't stop there. Drill remains a useful practice method virtually forever. Many experts who have been playing for 30 years still do regular drills, and still benefit from them.

But this stands in sharp contrast to programming, another skill I like to exercise (and analyze). Drill can be useful for programming neophytes, for learning such things as typing and the syntax of the language. But no experienced programmers I know engage in regular drill. The thought is actually ludicrous. What would you do? Write the same loop over and over, trying to do it a little faster or more accurately each time? Create a Hello World program from scratch 100 times in a row so that it becomes automatic? I don't think so.

So what's the difference between learning programming and learning pool? Why does drill have lasting value for one but not the other?

A worthy question, I thought to myself. It's deep enough that the answer should take a while to find, and interesting enough that the journey will keep my attention. So I girded myself for a long and arduous quest, set off smartly to find the answer, and stumbled over it immediately: drill is useful for learning *mechanics* — like high-precision muscular tasks — but it isn't very

useful for learning high-level problem-solving skills. Since experienced programmers spend most of their time on problem solving and very little on mechanics, drill just isn't an effective tool for getting better at programming once you're past the early stages.

Well, jeez, that was too easy. Isn't there more to it than that? Surely there must be deep and profound differences between learning to shoot pool and learning to program, since the tasks themselves are so completely different. Programming is like — well, you know what it's like, or you wouldn't be reading *develop*. It's mostly abstract and logical, and most of the real action takes place deep in your head or deep in the machine, far from the real world. Shooting pool is something else altogether. It's unabashedly physical, it often defies logic, and the action takes place where everyone can see it, on a huge table made of wood and slate and rubber and cloth.

I started playing pool fairly seriously several months ago, and I'm still embarrassingly terrible at it. In my typical overenthusiastic, obsessive-compulsive fashion, I dove in with both feet: I researched pool at the library, bought and read pool books, studied pool videotapes, cruised the Net for pool stuff, and jabbered about pool to anyone who came within earshot. The result was perhaps predictable. In no time, my knowledge of pool theory completely outstripped my ability to put it into practice. So although I could often see what to do in a certain situation, I couldn't actually do it. All bark, no bite.

To rectify this situation I started doing the only thing that would help: practicing doggedly. I took a lesson from a good instructor, and started hanging out at the pool hall as much as possible, putting in the practice time. Of course I was hoping that I'd suddenly make remarkable improvements in my game. But remarkable improvement is tough to come by in pool.

At first glance pool seems like it should be very straightforward. You have nearly perfect spheres undergoing nearly perfectly elastic collisions, so the paths of the balls should be nearly perfectly predictable, right? Wrong. Like most things that take place in the real world (as opposed to inside a computer), there's a whole seething world of subtleties and nonlinearities and complexities just beneath the surface. The actual grungy details — the drag of the cloth, the spin of the balls, the fleeting grip they have on each other when they collide — all affect the paths of the balls profoundly, and are so complex and intertwined that people argue endlessly about what's really going on. Superstitions, theories, rough approximations, and empirical formulas

DAVE JOHNSON (dkj@apple.com) recently determined that there are 6451 books currently in print whose titles begin with

"How to..." He's wondering how he'll ever find the time to read them all. •

abound. And all this complexity is set in motion in one tiny instant, by the impact of a chalked leather cue tip on a smooth plastic ball for a few milliseconds. If you ever needed an example of something with a sensitive dependence on initial conditions, this is a doozy.

Because of its complexity and sensitivity, progress in pool is slow no matter how you approach it. Playing pool is one of those things you can do all your life and keep getting better at, like playing a musical instrument. And like learning an instrument, just playing is lots of fun, and can be fine practice. But concentrated drill on the basics, especially for a beginner like me, helps in a way no other kind of practice can. When I started regularly doing drills, the effect on my game was immediate and tangible (if not as remarkable as I might have liked).

Good drill in pool involves intentional, conscious moving of your muscles the same way over and over, paying attention to the details of arm position, follow-through, rhythm, aim, and so on. You're trying to consciously train your muscles to learn the motions, so that those motions can be performed *unconsciously* later. To use a handy computing metaphor (always a good idea when talking to programmers), drill is like programming an EPROM: it pushes something that initially requires conscious control (the software) down into the unconscious realm (the hardware). Drill helps you deliberately “wear grooves” in your brain.

But that kind of “hardware” programming happens with any learning experience; it's not unique to drill. In fact, that's what learning *is*. Drill is just one kind of focused, repetitive practice that helps you learn certain things faster. All learning involves pushing stuff “down into the hardware.” (And I mean that literally: scientists are starting to identify the physical changes that happen in brains when animals learn.) To muddle my metaphors a little, learning is like climbing an endless terraced hill, where what you learn becomes the ground you stand on to reach the next level. Details that once required your full attention get tucked down into the unconscious realm and are hidden, in the same way that the details of your code get tucked down into subroutines and are forgotten. The point is this: once you learn anything, you can climb up on top of it, and other things that were unreachable before are brought within your grasp.

In my zeal to uncover the differences between learning pool and learning programming, I failed to notice the most remarkable thing of all: their similarity. The two goals couldn't be more different. Programming is the crafting of precision machinery in a tightly controlled

environment; pool is poking a ball with a stick (albeit in precise and skillful ways). Yet learning the two skills — for that matter, learning anything — is the same process. In fact, the more examples of learning I looked at, trying to categorize and separate them, the more the differences faded and the similarities came into focus.

In every case, learning is the same kind of journey. We climb that tiered structure, that terraced hill, standing on what we've learned before so that we can reach the new stuff. We slowly convert tasks that initially require our full attention into automatic, mechanical ones, and that conversion to mechanics is what allows us to turn our attention to more meaningful, higher-level tasks.

Attention seems to be the limiting factor here — we don't have much of it. Reaching once again for the low-hanging fruit on the computational metaphor tree, attention is like a single-threaded program with a tiny stack: we can only pay attention to a small handful of things at a time. The funny thing is that our brains aren't single threaded at all. Far from it! They are unbelievably prodigious and capacious things, and they actually *are* handling all the details, all the way down. We just aren't aware of it. And believe me, that's a good thing.

Without some way to convert conscious activities into unconscious ones, to push the details down out of sight — to *program ourselves* — we'd never get anywhere. Our meager helping of attention would be used up in no time. If we had to struggle with typing and syntax on every line of code we wrote, we'd never get the program written. If we had to consciously move each and every muscle all the time, we actually *wouldn't* be able to walk and chew gum at the same time — plain old standing around would probably be out of the question, much less hitting the cue ball with a little right English to sink the eleven ball and go two rails down table to get position on the thirteen. Lucky for us, the ability to program ourselves is built in. With a little desire and disciplined practice, we can do truly amazing things.

And oh, I *do* want to sink that thirteen ball. I really, really do...

RECOMMENDED READING

- *Byrne's Standard Book of Pool and Billiards* by Robert Byrne (Harcourt Brace, 1987).

Thanks to Lorraine Anderson, Jeff Barbose, Brian Hamlin, Bo3b Johnson, and Ned van Alstyne for their review comments. •

Dave welcomes feedback on his musings, so please let him know what you think. •

Newton Q & A: Ask the Llama

Q *How can I open an application so that it displays a particular data item?*

A If the target application supports Find, you can do that as long as three things are true:

- You know the application symbol.
- You know the application soup name and data format.
- The application supports the ShowFoundItem message.

If all of these are true, you can send the application the ShowFoundItem message with the appropriate arguments. Check the *Newton Programmer's Guide* for the arguments to ShowFoundItem. Be aware that not every application takes a soup entry as one of the arguments. That's why you need to know the application's data format. You can check whether the application supports the ShowFoundItem message with the following code:

```
local theApp := GetRoot().(kAppSymbol);
if theApp AND theApp.ShowFoundItem exists then
    // application installed and supports the message
```

Q *We have an application for a Newton device that communicates with the desktop. Because of the structure of our data, we'd like to be able to request a particular NewtonScript object. We thought of sending the reference or address of the NewtonScript object to the desktop and using that as the identifier, but we could find no way to do this. Are we missing something?*

A Unfortunately (or fortunately, depending on your point of view), Newton 2.0 OS doesn't provide a way to get the memory address of an object. Actually, since NewtonScript can relocate objects at will, providing an address would not be a good idea. There's an alternate approach: you can maintain an array of the objects you want to export. The array index can be used in much the same way as the address. As an example, in the code below, the memory "address" for **object2** would be 1. In other words, **myObjectArray[1]** would give you **object2**.

```
object1 := "foo";
object2 := {can: 'aid, eee: "an", a: "...yep"};
object3 := [1,2,3];
myObjectArray := [object1, object2, object3];
```

If you need to indicate that an object has already been transferred to the desktop, you can simply replace the object at the relevant array index with NIL.

Q *I'm designing my data structures. I figure I could use either two cursors onto two different soups or two cursors onto the same soup. Which is the more efficient solution?*

A You can measure efficiency in two relevant ways: by time or by memory usage (or both). The time to create the two cursors will be the same regardless of the number of soups, but more heap space will be required for two soups. With two soups, it may take less time to find items that exist in just one soup than when

The llama is the unofficial mascot of the Developer Technical Support group in Apple's Newton Systems Group. Send your Newton-

related questions to dr.llama@newton.apple.com. The first time we use a question from you, we'll send you a T-shirt. •

searching a larger, combined soup. However, with two soups you won't get as much benefit from the operating system's caching of the entries; there's more overhead information to swap in and out of the heap, which increases the time required to get data.

The real answer is to test it with your actual data and see. Overall, two cursors on one soup sounds like the more efficient way to go. Your question implies that you're going to have two completely different sets of data. You can do this in one soup by using indexes, because entries with either no indexed slot or NIL in an indexed slot won't participate in that index. That is, when you create a cursor that uses that index, entries with NIL values will be ignored by that cursor.

Something that might occur to you is using tags to implement the two different sets of data (that is, each set would have a unique tag value), but this doesn't work as well as using an index. With an index, you can navigate to an entry in $O(\log n)$ time, where n is the number of entries that are in that index. In other words, the time taken to navigate to a particular entry will be directly related to the log of the total number of entries. If your query includes a `beginKey/endKey` or `startExclKey/endExclKey` subrange, the system finds that subrange very quickly. It can then quickly step through entries in between.

The operating system gets the set of tags for an entry efficiently, but it has to know which entries to get the tags for first. So with no other way to narrow the search, it will check all the entries, assuming you aren't using an index. Getting the tags is actually very efficient, but indexes work better for subranging.

Q *I'm trying to compile a program that works with both Newton 1.x and Newton 2.0 OS devices; however, it won't compile. Newton Toolkit complains that I have a bad magic pointer, but I know that the value is defined in the MessagePad platform file. The offending code is as follows:*

```
local theCountries :=
  call kGetUserConfigFunc with ('commonCountries);
if ClassOf(ROM_Countries) = 'frame then
  // on a 1.x unit
  labelCommands := foreach item in theCountries collect
    ROM_Countries.(item).name;
else
```

This would give a nice pop-up menu of countries on a 1.x unit. Why doesn't it work?

A This is a subtle problem. In Newton Toolkit 1.5 and later there are certain functions called *constant functions* that will evaluate at compile time when their arguments are constant. The most common ones are `GetLayout`, which will return a reference to another Newton Toolkit layout, and `LocObj`. The `ClassOf` function is another one of these.

At compile time, a magic pointer is considered a constant value. That means that the `ClassOf` call in your conditional is executing at compile time. Of course, there's no Newton device around at compile time, so Newton Toolkit is unable to dereference the magic pointer. Hence the error.

One workaround is to set a local to the value of the magic pointer and use that local in your conditional. This works because the value of the argument to `ClassOf` is no longer a constant, so it will not be called at compile time.

```
local mpCountries := ROM_Countries;
if ClassOf(mpCountries) = 'frame then
```

Q *My communications program has a number of standard packets of information. I'm trying to set up constants for each of these standard packets. However, for the packet*

```
constant kHaltAndCatchFireMessage := "\u102cff1003";
```

Newton Toolkit complains that there's an "odd number of digits between \u's." I count ten, which looks even to me. Does Newton Toolkit need a remedial math course?

A Actually, Newton Toolkit is doing fine in math, but it should say "bytes" instead of "digits." There are ten hex digits in your string, but there are two hex digits per byte, so your string is five bytes long. Since Unicode is a double-byte representation, there are four hex digits per Unicode character. You have ten hex digits, or two and a half Unicode characters, which is an invalid Unicode string.

You can either add two more hex digits to your string or use the `MakeBinary` and `Stuff...` functions. If you're dealing with data that's not strings, the latter method is the best one for compatibility. It's also likely to keep you saner.

Q *I'm trying to dial the following number using a Newton Fax Modem with my MessagePad 130: "18005551234,,,,,,1,,,408-555-1234,,,123-456-789-123,,,". I get an error -16013 in my communications code whenever I do this. I need to use the long string because it contains a calling card number. My modem dials correctly and the modem at the other end picks up. I even hear the chirping whistle of exchanging bits. But suddenly things just stop and the error occurs. Any clues?*

A Yes. First star on the right, then straight on till morning. But that's a different story. In answer to your question, it looks as though you're timing out on the connection attempt. Modems have a set amount of time to establish a connection, and the commas are reducing the time they have. Each of the commas will insert a delay into the dialing. For most modems, the time for each comma is controlled by register S08 and usually defaults to 2 seconds. You have 19 commas, so that's 38 seconds, which leaves very little time for the modems to sync up (the chirping whistle exchange you're hearing).

The solution is to increase the timeout of the modem to a more reasonable value. When you're thinking about the timeout, remember that each digit will take around 95 milliseconds to dial. There will also be a line connection time of about 2 seconds, a ring time of a few seconds, and the final sync-up or negotiation time of 2 to 15 seconds. You should increase your timeout values to at least 60 seconds. If that doesn't work, add 30-second increments. You can do a binary search to narrow it down to an optimal value.

To set the timeout for the modem, use 60 for the `waitForCarrier` (sixth) argument of the `kCMOModemDialing` option. The following bit of code will do this:

```
// make a modem option data structure based on user preferences
local option := MakeModemOption();
// modify the timeout value
option.data.arglist[6] := 60;
// set that option in your endpoint
ep:Option(option);
```

Q How can I tell whether a tap is the first of a double tap?

A Unfortunately, the RUM (Read User Mind) ASIC didn't get completed in time for Newton 2.0 OS, so we were unable to implement the `IsFirstTap` global function. We also looked at a wireless link to one of the 900-number pay-by-the-minute psychic lines but couldn't figure out how to bill the user.

But seriously, you can't tell. The best you can do is to hold off processing the first tap for some amount of time. If you receive another tap in that time, it's a double tap. The drawback is that if it isn't a double tap, you've lost the unit parameter from the first tap, since you can't save this parameter. The other option is to follow a user interface guideline: Make the second tap an extension of the functionality that happens with the first tap. Then there's no need to handle the first tap in a special way.

Q *We have a problem with union soups. We have an application that creates soups and transfers them to a PC. The soups can get sent down to a different MessagePad. If the user inserts a storage card and selects it as the default store, we can't successfully add an item to the soup. Our code does a `GetUnionSoupAlways`, then tries to add an entry using `AddToDefaultStoreXmit`. The Newton throws an exception that tells us there's no `soupDef`. We're sure that the soup doesn't exist on the store, but we thought that `GetUnionSoupAlways` creates the soup if you try to add something. One thing we thought of was to use `RegUnionSoup`, but our transfer application doesn't know what the `soupDef` is. Is there a way to copy a `soupDef` from one store to another or to get the `soupDef` from an existing soup?*

A Well, first you need some good onions, then some stale French bread, Gruyère cheese...oh, sorry, I thought you said "onion soup."

For a union soup to work properly in Newton 2.0 OS, a `soupDef` must exist in at least one of two places: it can be registered with the OS via `RegUnionSoup`, or it can exist within a soup that's on a mounted store. `GetUnionSoupAlways` should fail if there's no `soupDef` present. However, in the current release of the Newton 2.0 OS ROMs it doesn't. This means the problem is deferred until you first try to add an entry, which is when the OS tries to create the soup but can't find the `soupDef`. That's why you get the error on the call to `AddToDefaultStoreXmit`. Of course, this doesn't help you, but there are a few options:

- Make sure that a `soupDef` is registered, via `RegUnionSoup`.
- Make sure that an actual soup exists on some store and that that soup contains an embedded `soupDef`. The soup doesn't actually have to have any entries. You can use the `CreateSoupFromSoupDef` function or the `GetMember` soup message to do this. For example:

```
RegUnionSoup(kMySoupDef):GetMember(GetStores()[0]);
```
- Don't use union soups; instead, have your download application just send the store either the `CreateSoupXmit` or the `GetSoup` message.
- Write some smart code that checks to see if a soup with the same name exists on any store and duplicate that soup on the new default store. If you use `GetIndexes/CreateSoupXmit` and `GetAllInfo/SetAllInfoXmit`, you should be able to make a reasonably similar soup.

Unfortunately, there's no supported way to directly access the `soupDef` of an existing soup.

Q *I have an application that performs some lengthy initializations in the `installScript`. I need a slip to come up and inform the user that this action is occurring. The problem is that the `BuildContext` slip I create at the beginning of the `installScript` doesn't show up when the `installScript` is running. How can I get a slip to come up in my `installScript`?*

A I assume that you do something like create the slip, send the slip an Open message, and then do a tight loop with some initializations. If so, the system has probably opened your view, but your `installScript` is still executing. That means the system cannot refresh the display.

One possible approach is to call `RefreshViews` to force the system to update the display. However, if your progress indication is dynamic, you'll have to call `RefreshViews` each time you change the progress slip. A better approach is to use the `DoProgress` call, which provides a standard "Your Newton device is doing something" interface for the user. You may also want to do your initialization in a deferred action.

Q *I'd like to add another item to the address picker pop-up list in my "To:," "Cc:," and "Bcc:" pickers that would allow the user to create an e-mail address without adding it to the Names soup. The user interface reasoning behind wanting to do this is to avoid cluttering the Names soup with addresses that are used only once. I've successfully added an item to the picker and caught the `pickActionScript` for it. The pick item that I want to use to add this temporary name appears in the `protoAddressPicker` pop-up list. Now I want to bring up an editor and add my temporary item. I tried the call*

```
GetDataDefs(' |nameRef.email| ):New(tapInfo, self);
```

from my `protoAddressPicker`'s `pickActionScript`, but I got an exception: Object {class: nameRef.email, name: "E-Mail addresses", preferredRouting: [string.email], ...} is read-only. Why can't I create a new data object with this?

A Usually answers to these questions are reasonably self-contained; this is an exception. Before you can understand this answer, you really need to read up on list pickers and data, or you'll be tripped up by the subtle differences between `nameRefs` and `dataDefs`.

The transport system uses a structure called a `nameRef` for information on where to send things. It so happens that `nameRefs` use the data definition registry as a repository. However, a `nameRef` is a different beastie from a `dataDef`. To create a new empty `nameRef` structure, you can use the call

```
GetDataDefs(' |nameRef.email| ):MakeNameRef(tapInfo, self);
```

Then you can use some sort of floating editor to enter the values. I suggest using a `protoFloatNGo` that contains a `newtFalseEntryView` and then appropriate slot views for the fields of the `nameRef` that you want to edit.

Thanks to jXopher Bell, Henry Cate, Bob Ebert, David Fedor, Ryan Robertson, Jim Schram, Maurice Sharp, and Bruce Thompson for these answers. •

If you need more answers, check out <http://dev.info.apple.com/newton> on the World Wide Web. •

QuickTime Quandary

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 So, BAL, it seems we need to increase our advertising budget to rope in more guest Puzzle Page authors.
- BAL People don't realize the fame and fortune that comes with being part of the Puzzle Page. I heard that one person had a lot of good luck shortly after making her first Puzzle Page submission.
- KON I heard that another person who thought about submitting a column to the Puzzle Page, but then decided not to, lost some valuable files.
- BAL Enough chain letter tactics. Maybe if we write another column about QuickDraw or QuickTime, someone will be hungry enough for a change of pace to submit his own Puzzle Page.
- KON I figure we should talk about the Internet and then take the Puzzle Page public! \$10 to \$12 a share sounds about right to me. Then we can have some serious writing bounties!
- BAL Maybe we'll change the medium. Instead of printing it, we'll deliver it over TV.
- KON OK, actually I do have a weird problem. It might be QuickTime-related. I'm trying to do some video digitizing, but every time I bring

KONSTANTIN OTHMER AND BRUCE LEAK

dropped off this press release in lieu of the usual biographical information:

PALO ALTO, California, April 1, 1996 — Balkon Heavy Industries today announced PuzzleMill™, a next-generation, low-cost, networked virtual puzzle architecture for the World Wide Web, corporate intranets, infinity, and beyond. "Along with our industry-standard Puzzle Page column, we've set the agenda for digital puzzling into the

next century," said BAL, Balkon's senior vice president for corporate restructuring. Balkon's executive vice consul for corporate misconduct KON will be acting as grist for the PuzzleMill until a sack of flour can be found to replace him. KON let spill that the beta version of PuzzleMill can be downloaded free of charge from <http://www.always.balkon.com> "until we achieve critical mass, at which point we'll charge as much as we want for it, darn it." •

up the Video Settings dialog to choose a compression method, my machine locks up.

BAL Locks up? How? Dead cursor, no MacsBug, the works?

100 KON The cursor is still alive. I can even go into MacsBug and choose Exit to Shell and everything's fine. But I can't capture video since the dialog just hangs.

BAL What does the dialog look like when it hangs? Can you click in other applications?

95 KON The basic structure is there. It draws the outline of the Choose CODEC pop-up menu, but there's no text. You can click all you want, but no context switch occurs. You're stuck.

BAL I've never heard of that before. Why does this stuff always happen to you, KON?

KON Believe me, I'd love to know the answer to that puzzle!

BAL With all your problems, you should write a book on debugging.

KON Anyway, back to my QuickTime nightmare. Any ideas?

BAL Clearly, that case has worked for millions of people for a long time. Tell me more. There's probably something funny about your machine.

90 KON I have a Power Macintosh 8100/80. I had an HPV card but then traded Shannon for his AV card. I tried to buy an AV card, but it's really hard to get one.

BAL You always have weird hardware and other stuff. What version of the system are you running? This isn't some beta card again, is it?

KON I had version 7.1.2 originally, but as soon as I started having problems with QuickTime I figured I'd take the opportunity to "upgrade" to System 7.5. All my hardware is stock Apple stuff. I've written a lot of crazy programs, the results of many of which have appeared in these pages, but the hardware seems to be kosher.

BAL Which version of 7.5?

KON I started with plain 7.5. Then I heard about the fix release, so I upgraded to 7.5.1. I put the project on hold for a while, and heard about a fix for the fix, so I installed 7.5.2. There was a fix-cubed release, so now I'm running 7.5.3 and it still happens. Maybe I should wait for 7.5.4?

BAL Come on, KON. Clearly the problem isn't related to a system release. You did a clean install, right?

KON Last I heard, the magic incantation was to drag the Finder inside the Preferences folder, rename the System Folder, jog around your chair three times, and say a prayer.

BAL Did you sacrifice a frog?

85 KON Seriously, the system install is fine.

BAL What version of QuickTime is it?

80 KON QuickTime 2.1. That was the latest version I could find. Movies play back OK — it's just this capture thing that's giving me fits.

BAL I guess you've tried replacing QuickTime. Does this happen in other applications as well?

KON I've tried three different applications. Every one has the exact same symptoms: it locks up when it brings up the Video Settings dialog.

BAL On one level that makes sense. Applications just call QuickTime to put up that dialog. But what doesn't make sense is that you have a clean system install, standard Apple hardware, and the latest QuickTime version, and it hangs. That's crazy. Any weird extensions or anything?

KON Nope. Totally clean install.

BAL Swap the hard drive.

70 KON Still happens.

BAL Swap the video card.

60 KON 60 and falling fast.

BAL The monitor?

KON Let's leave sense-line bugs for a later date. Now that you've swapped out the whole system except the motherboard, the TV repairman approach is over.

BAL Fine. Give me MacsBug. I'll break into the debugger when the system hangs and try to figure out what's going on.

50 KON It looks like you're in the Font Manager routine RealFont, which is being called from a loop in QuickTime. Here's what it looks like:

Disassembling from 1C5B562

```
'CDEF 0064 0F6E'
+015B2 01C5B562    MOVEQ    #$01,D0                |7001
+015B4 01C5B564    MOVE.W   D0,-(A7)              |3F00
+015B6 01C5B566    _TextFont                      ; 0019D0E4 |A887
+015B8 01C5B568    MOVEQ    #$08,D6                |7C08
+015BA 01C5B56A    BRA.S    'CDEF 0064 10E6'+015CA ; 01C5B57A |600E
+015BC 01C5B56C    SUBQ.L   #$2,A7                |558F
+015BE 01C5B56E    MOVEQ    #$01,D0                |7001
+015C0 01C5B570    MOVE.W   D0,-(A7)              |3F00
+015C2 01C5B572    ADDQ.W   #$1,D6                |5246
+015C4 01C5B574    MOVE.W   D6,-(A7)              |3F06
+015C6 01C5B576    *_RealFont                      ; 408C2B2E |A902
+015C8 01C5B578    MOVE.B   (A7)+,D7              |1E1F
+015CA 01C5B57A    TST.B    D7                    |4A07
+015CC 01C5B57C    BEQ.S    'CDEF 0064 10E6'+015BC ; 01C5B56C |67EE
+015CE 01C5B57E    MOVE.W   D6,-(A7)              |3F06
+015D0 01C5B580    _TextSize                      ; 0019D18C |A88A
```

BAL Aha! It's starting to sound a little like a QuickDraw bug to me! What kind of nastiness did you put in that code, KON?

40 KON Not so quick, pal. RealFont is returning just fine. But the loop calling it doesn't terminate.

BAL Well, RealFont just tells you whether a particular font size exists. QuickTime calls RealFont to make sure that the drawing operation in the Video Settings dialog will look good: if the requested font size doesn't exist, things will scale and look really ugly. In that case, QuickTime increments the font size and keeps looking.

KON OK.

-
- BAL These dialogs should be drawn with the system font. Is there some strange problem with your fonts that persists across system installs? I thought I told you to swap hard drives.
- 30 KON I did a fresh install on a new hard drive and the problem continued.
- BAL Hmm. It sounds like the bug is that QuickTime is searching for a system font size that won't be scaled — that is, that's real. It probably expects it to be there. If it's not, QuickTime spins forever looking.
- KON OK. So why can't it find it?
- BAL What font are you looking for?
- 20 KON The font ID passed into RealFont is 1 (AppFont), which RealFont converts internally to the application font by reading the short at 0x984 (ApFontID).
- BAL What font is it?
- KON How do I figure that out?
- BAL Call GetFontName. We don't even need to write a program to do this. You can hack the stack from MacsBug. First, go to some trap call so that you know the proper registers are saved and all of that. Subtract 6 from the stack. Put the address of where you want the name to end up at the old stack address, and the fontNum, 1, after that. Put the address of the GetFontName trap, 0xA8FF, at location 0, set the PC to 0, and trace.
- KON You should probably turn off EvenBetterBusError when doing this sort of thing.
- BAL Well, yes, that's true. Or, alternatively, we could find a large free block somewhere and put the code there. Of course, we'd have to be sure the call doesn't move memory, or our code might be written over — although for this single trap it doesn't matter. Anyway, you get the idea.
- KON The work you'll go through to keep from writing any real code! Wait. Where do I get the memory for the name?
- BAL You have three choices: you could have subtracted another 255 bytes or so from the stack and just used that. Better yet is to use some of MacsBug's internal buffer. When you use the **dh** command, MacsBug puts the hex data in a buffer and disassembles it. The address in the disassembly is the address of the MacsBug buffer. Finally, you could look for a free block with lots of space and use that.
- 10 KON OK. The font name comes back 0.
- BAL If you trust that, it means there's no font with that fontNum. So it makes sense that QuickTime would never find a RealFont at any size for that fontNum.
- KON So why didn't the system install fix it?
- BAL The system font ID is stored in PRAM and is put in a low-memory global during startup. Apparently the install process doesn't touch PRAM. Zap your PRAM by holding Command-Option-Shift-P-R during startup — user friendly!
- KON OK. Now it works.
- BAL So the installer should clear PRAM when a new system is installed. It should keep your video card configuration and other settings, which really belong on the hard disk as well, but should clear stuff like the

default fonts since they may not exist, or they might be renumbered, in the new install.

KON And QuickTime shouldn't spin in an endless loop expecting something to exist.

BAL PRAM is a holdover from the 128K Macintosh. It was designed as a closed system that might never have a hard drive. At that time there were only floppies, so it made a lot of sense to store system parameters with the machine rather than the media. But since then, no one has ever revisited whether PRAM is needed.

KON The machine still has a ROM, for crying out loud! I guess it's too soon to give up those silly incantations of rebuilding the desktop and zapping PRAM. By the way, I understand the problem was worked around in QuickTime 2.5 by aborting the font search loop at a maximum point size of 36.

BAL Nasty.

KON Yeah.

SCORING

90-100 Yeah, sure. And you just had lunch with D. B. Cooper.

70-85 Congratulations! You've just qualified to write the next Puzzle Page.

40-60 Your spirit guides must be with you today.

10-30 Care to join our poker game? •

Thanks to Peter Hoddie, Josh Horwich, and Bo3b Johnson for reviewing this column. •



How're we doing?

If you have questions, suggestions, or even gripes about *develop*, please don't keep them to yourself. Drop us a line and let us know what you think.

Send editorial suggestions or comments to develop@apple.com or to:

Caroline Rose
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
crose@apple.com
Fax: (408)974-9423

Send technical questions about *develop* to:

Dave Johnson
Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
dkj@apple.com
CompuServe: 75300,715
Fax: (408)974-9423

Please direct all subscription-related queries to Apple Developer Catalog, P.O. Box 319, Buffalo, NY 14207-0319 or to order.adc@applelink.apple.com. You can also call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, or (716)871-6555 elsewhere.



INDEX

For a cumulative index to all issues of *develop*, see this issue's CD. •

A

ActiveDocumentIsDirty (Apple Guide 2.1) 67
ActivePartAllowsEmbedding (Apple Guide 2.1) 67
ActivePartIsRoot (Apple Guide 2.1) 67
AddFancierLanguageModel (Speech Recognition Manager) 16, 18, 20
“Adding Speech Recognition to an Application Framework” (Monroe) 22–33
AddToDefaultStoreXmit (Newton Q & A) 116
AddValue (OpenDoc) 49
AEGetParamPtr, Speech Recognition Manager and 12
AGInstallContextHandler (Apple Guide 2.1) 68
AGOpen (Apple Guide 2.1) 61
AGRemoveContextHandler (Apple Guide 2.1) 68
<App Creator> command (Apple Guide) 55, 56, 60
AppendDITL, Mac OS 8 assistants and 79
AppendMenu, Apple Guide 2.1 and 61
Apple events, sending from Apple Guide guides 68
Apple Guide
 populating the Help menu 54–55
 vs. Mac OS 8 assistants 78
Apple Guide 2.0, populating the Help menu 56
Apple Guide 2.1
 and Apple events 68
 and AppleScript scripts 68
 application signatures 56
 building the Help menu 54–60
 coachmarks 62–64
 context checks 64–68
 document specific help 56
 multiprocess guide files 56–59
 new features 53–54

 new resources 61
 placing a part editor's guide into the Help menu 60–61
 populating the Help menu 56
 using with OpenDoc 59–68
AppleGuideGlue, Macintosh Q & A 102–103
AppleGuideGlueLib, Macintosh Q & A 103
AppleGuideGlueLib.xcoff file, Macintosh Q & A 103
AppleGuideGlue.xcoff file, Macintosh Q & A 103
AppleGuidePlugIn, OpenDoc and 65
AppleScript, launching scripts from Apple Guide guides 68
AppleScript coaches (Apple Guide 2.1) 64
AppleTalk
 Network Name (Macintosh Q & A) 108–109
 self-send variable (Macintosh Q & A) 105–106
application signatures (Apple Guide 2.1) 56
ApplLimit low-memory global, MaxApplZone and 83
'apsg' resource (Apple Guide 2.1) 56, 57, 60, 61
Arcellana, José 72
ASCII standard, OpenDoc and 38
assistants (Mac OS 8)
 error trapping 78
 interviews 73–74, 75–81
 substituting parameter values in dialogs 80–81
 in System 7 applications 72–81
 vs. Apple Guide 78
 vs. Microsoft wizards 73

B

back issues of *develop* 5
“Balance of Power” (Evans), Stalking the Wild Defect 82–86
ento container suite (OpenDoc) 39
BinHex part kind (OpenDoc) 43
breakpoints, setting in PowerPC code 85

BuildLanguageModel (Speech Recognition Manager) 17

C

CallOSTrapUniversalProc (Mixed Mode Manager), MaxApplZone and 83
camera movements (in QuickDraw 3D applications) 87–97
 3D geometry 90–91
 delta factors 94, 95
 initializing camera data 89
 keyboard/mouse controls 94–97
 moving/rotating the camera 92–94
 rendering loops 92
 setting camera data 92
CategoryUserString resource (OpenDoc) 44
CDocDemoApp::CDocDemoApp constructor 23
CDocDemoApp::~CDocDemoApp destructor 23, 32–33
CDocDemoApp.cp source file 23
CDocSpeech class 23, 25, 32
CDocSpeech.cp file 25–26
CDocSpeech.h header file 24–25
Çelik, Tantek 37
CFMTerminate 69
ChangeKind (OpenDoc) 51
CInternetConfig, Mac OS 8 assistants and 81
ClassOf (Newton Q & A) 114–115
CloneInto (OpenDoc) 49–50
Close (OpenDoc) 70
cmd_New constant (PowerPlant), speech recognition and 29, 32
CMultiDialog, Mac OS 8 assistants and 79
coachmarks (Apple Guide 2.1) 62–64
Code Fragment Manager (CFM)
 library closure 71
 unloading OpenDoc part editor libraries 69, 70–71
Commons, Peter 53
Communications library (LaserWriter 8.4) 36
constant functions (Newton Q & A) 114–115

context checks (Apple Guide 2.1)
 64–68
 application information 65
 custom 67–68
 process 65
 standard OpenDoc suite
 65–67
 standard suite 65
 system information 65
 Converter library (LaserWriter
 8.4) 36
 CreateSoupFromSoupDef
 (Newton Q & A) 116
 CreateSoupXmit (Newton Q & A)
 116
 Curbow, Dave 37
 Cyberdog
 Apple Guide 2.1 and 53
 Web browser window 63

D

dataRef (Newton Q & A) 117
 DDP socket listener (AppleTalk),
 Macintosh Q & A 109
 debugging
 onsite with MacsBug
 (Macintosh Q & A) 102
 PowerPC 82–86
 resources on (Macintosh
 Q & A) 102
develop back issues 5
 dirID (Macintosh Q & A)
 109–110
 DocDemo (PowerPlant), adding
 speech recognition to 22
 DoClosepath (QuickDraw GX)
 100
 document file commands (File
 menu), speech recognition and
 27–29
 document-specific help (Apple
 Guide 2.1) 56
 DoCurveto (QuickDraw GX) 100
 DoLineto (QuickDraw GX) 100
 DoMoveto (QuickDraw GX) 100
 DoProgress (Newton Q & A) 117
 DoSendLkUpReq (Open
 Transport), Macintosh Q & A
 109
 double tap (Newton Q & A) 116
 Downloader library (LaserWriter
 8.4) 36
drd dcmd (MacsBug) 83–84
 dynamic class objects (SOM),
 OpenDoc and 69

E

EditorKinds resource (OpenDoc)
 43
 EditorPlatformKind resource
 (OpenDoc) 44
 Editor Setup control panel
 (OpenDoc) 40
 “Editor Substitution Explained”
 (OpenDoc) 40
 Evans, Dave 82–86
everytime macro (MacsBug),
 Macintosh Q & A 102
 experts. *See* assistants
 extendPrDefault, LaserWriter 8.4
 and 34
 extendPrValidate, LaserWriter 8.4
 and 34
 Externalize (OpenDoc) 49
 ExternalizeKinds (OpenDoc) 52
 'extm' context check (Apple Guide
 2.1) 68

F

fDirty flag (OpenDoc) 49
 feedback window (Speech
 Recognition Manager) 7, 12, 14
 fidelity ordering (OpenDoc) 49
 FindCommandStatus, for
 document windows 31
 FindSym (MacsBug) 85
 “flagship name,” determining
 (Macintosh Q & A) 108–109
 FRONT Guide Script constant
 (Apple Guide 2.1) 62–64, 65,
 67
 Full Access window (Apple Guide
 2.1) 56–59

G

game controls (QuickDraw 3D)
 87–97
 3D geometry 90–91
 controlling camera
 movements 87–92
 controlling the controls
 94–97
 See also camera movements
 “Game Controls for QuickDraw
 3D” (McBride) 87–97
 gDocSpeechObj, deleting 23–24
 <Gestalt> checks (Apple Guide)
 55, 56
 GetAllInfo/SetAllInfoXmit
 (Newton Q & A) 116

GetFontName, KON & BAL
 puzzle 121
 GetIndexes/CreateSoupXmit
 (Newton Q & A) 116
 GetLayout (Newton Q & A) 114
 GetMember (Newton Q & A) 116
 GetOSTrapAddress,
 MaxApplZone and 83
 GetSoup (Newton Q & A) 116
 GetUnionSoupAlways (Newton
 Q & A) 116
 GetValue (ODStorageUnit)
 (OpenDoc) 49
 GetZoneList (AppleTalk),
 Macintosh Q & A 109
 Global Guide Files folder (Apple
 Guide) 56, 59, 60
 Gourdol, Arno 72
 “Graphical Truffles” (Lipton), A
 Library for Traversing Paths
 98–101
 gTopLanguageModel (Speech
 Recognition Manager) 11, 14
 Guide Maker (Apple Guide 2.1)
 60
 Guide menu. *See* Help menu
 GXDrawShape, Macintosh Q & A
 104
 GXGetGraphicsError 101
 gxLayouts, Macintosh Q & A
 103–104
 GXNewStyle, Macintosh Q & A
 103
 gxNoContourGridText,
 Macintosh Q & A 104
 gxNoMetricsGridText, Macintosh
 Q & A 104
 gxPath structure 98
 gxPaths structure 98

H

HandleRecognitionDoneAE
 (Apple event handler), Speech
 Recognition Manager and 9,
 12, 13–14, 16
 HandleSpeechBegunAppleEvent
 (Apple event handler), speech
 recognition and 30–31
 Help menu
 building with Apple Guide
 54–60
 with a multiprocess guide
 item 57
 placing a part editor’s guide
 into 60–61

populating with Apple Guide 54–56

<Help Menu> command (Apple Guide) 55, 58–59

HFSFlavor value type (OpenDoc) 48

HMGetHelpMenuHandle, Apple Guide 2.1 and 61

HTML part kind (OpenDoc) 43

I

implicit points (QuickDraw GX) 99

InitPartFromStorage (OpenDoc) 48

installScript (Newton Q & A) 117

InterfaceLib, MaxApplZone and 83, 85

Internet Configuration System (Internet Config)

- creating Mac OS 8–style assistants for 72–81
- main window 75

Internet Setup Assistant 72, 75–78, 81

interview window, for Mac OS 8 assistants 74, 75

IP_REUSEADDR (Open Transport), Macintosh Q & A 106–107

isDefaultPaperType flag (QuickDraw GX), Macintosh Q & A 103

IsOpenDocActiveAndFrontmost (Apple Guide 2.1) 66

IsPlugInAvailable (Apple Guide 2.1) 65–66

item coaches (Apple Guide 2.1) 63–64

J

Johnson, Dave 2, 111

K

kCMOModemDialing (Newton Q & A) 115

kExtendPrintRecOp (LaserWriter 8.4) 34

kGetExtendedPrintRecOp (LaserWriter 8.4) 34

KindCategories resource (OpenDoc) 44, 45

KindUserString resource (OpenDoc) 44

kODCategoryArchive 41

kODCategoryDrawing 42

kODCategoryOutline 42

kODCategoryPageLayout 42

kODCategoryPersonalInfo 42

kODCategorySpace 42

kODPlatformDataType 45

kODPlatformFileType 45

kODPropContents 48

kODPropPreferredEditor 40

kODPropPreferredKind 48

“KON & BAL’s Puzzle Page” (Othmer and Leak), QuickTime Quandary 118–122

kOTNoDataErr (Open Transport), Macintosh Q & A 104

kOTOStateErr (Open Transport), Macintosh Q & A 104, 108

kSetExtendedPrintRecOp (LaserWriter 8.4) 34

kSRFeedbackAndListeningModes property (Speech Recognition Manager) 11, 12

kSRLanguageModelFormat property (Speech Recognition Manager) 12–14

kSROptional property (Speech Recognition Manager) 20

kSRRefCon property (Speech Recognition Manager) 11, 14

kSRSpelling property (Speech Recognition Manager) 9, 14

kSRTEXTFormat property (Speech Recognition Manager) 12

L

language models (Speech Recognition Manager) 7, 8, 17

- active 7, 14–16, 27
- building 11, 13, 16–18, 27–29
- constraining 12
- embedded 16, 27
- emptying and refilling 19
- enabling/disabling (parts of) 18–20, 29–30
- manipulating 18–20
- saving into resources 20

language objects (Speech Recognition Manager)

- enabling/disabling 31
- saving/loading 20
- See also* language models

LaserWriter driver version 8.4 34–36

error codes 36

extended print records 34

one-pass printing 35–36

PPD files and 36

PrGeneral opcodes 34–35

print dialogs 34–35

shared libraries 36

Leak, Bruce 118

library closure (CFM) 71

Lipton, Daniel I. 98

LocObj (Newton Q & A) 114

Lo, Vincent 69

M

McBride, Philip 87

Macintosh Q & A 102–110

“Mac OS 8 Assistants in System 7 Applications” (Arcellana and Gourdol) 72–81

MacsBug, onsite debugging with (Macintosh Q & A) 102

MakeBinary, Newton Q & A 115

MakeLanguageModels, speech recognition and 25, 26, 27–29

MaxApplZone, PowerPC

- debugging and 82–85

Memory Manager, MaxApplZone and 84

menu coaches (Apple Guide 2.1) 63

Merged Access window (Apple Guide 2.1) 57–59

_MixedModeMagic trap (Mixed Mode Manager) 83, 84

Mixin guide files (Apple Guide 2.1) 55

- vs. multiprocess guide files 59

<Mixin Match> command (Apple Guide 2.1) 59

“Mixin vs. Multiprocess Guide Files” (Apple Guide 2.1) 59

'mli' resource (Apple Guide 2.1) 56–59, 60, 61

Monroe, Tim 22

multipane print dialog (LaserWriter 8.4) 35

multiprocess guide files (Apple Guide 2.1) 56–59

- and Help guide files 59
- vs. Mixin guide files 59

'mxbm' resource, MacsBug and (Macintosh Q & A) 102

MyGetCameraData (QuickDraw 3D) 89

MyGuideOpenDocResource (Apple Guide 2.1) 61

MyInitDeltaFactors (QuickDraw 3D) 94, 95
MyMoveCameraZ (QuickDraw 3D) 92, 93
MyRotateCameraY (QuickDraw 3D) 92, 93–94
MySetCameraData (QuickDraw 3D) 92

N

nameRef (Newton Q & A) 117
NBP LkUp (AppleTalk),
Macintosh Q & A 108–109
NBP LkUp-Reply (AppleTalk),
Macintosh Q & A 109
Network Name (AppleTalk),
Macintosh Q & A 108–109
“New Apple Guide Resources”
(Apple Guide 2.1) 61
Newton 2.0, Newton 1.x
compatibility (Newton Q & A)
114–115
Newton Fax Modem, setting
timeout (Newton Q & A) 115
Newton Q & A: Ask the Llama
113–117
NewtonScript objects (Newton
Q & A) 113

O

object coaches (Apple Guide 2.1)
64
ODEExtension 69
ODEExtension::BaseRemoved 70
ODFrame::Remove 70
ODNewObject 71
ODPart 70
ODPart::DisplayFrameRemoved
70
ODSession 45
'odtm' creator code (Apple Guide
2.1) 60, 62, 65
ODTranslation 45
OpenDoc
circular references 70
part editors 37–52
part kinds 37–52
part viewers 37
persistent objects 70
reference counting 69–70
self-referencing 70
standard context checks for
Apple Guide 2.1 65–67
unloading part editors 69–71
using Apple Guide 2.1 with
59–68

See also part kinds

OpenDoc part kinds. *See* part kinds
Open Transport
determining Network Name
(Macintosh Q & A) 109
hand-off (secondary)
endpoints (Macintosh
Q & A) 104–105
orderly disconnect
(Macintosh Q & A) 104
TCP server connections
(Macintosh Q & A)
106–108
Open Transport AppleTalk, self-
send variable (Macintosh
Q & A) 105–106
OTAccept, Macintosh Q & A
104, 106
OTConnect, Macintosh Q & A
106, 108
Othmer, Konstantin 118
OTIoctl, Macintosh Q & A 105
OTListen, Macintosh Q & A 106
OTLook, Macintosh Q & A 104
OTLookupName, Macintosh
Q & A 109
OTOOptionManagement,
Macintosh Q & A 106–107
OTRcv, Macintosh Q & A 105
OTRcvConnect, Macintosh
Q & A 106
OTRcvOrderlyDisconnect,
Macintosh Q & A 104
OTSndDisconnect, Macintosh
Q & A 108
OTSndOrderlyDisconnect,
Macintosh Q & A 104

P

Pallakoff, Matt 6
part categories (OpenDoc) 41–43
predefined 41–43
user strings 44
PartEditorInstalled (Apple Guide
2.1) 66
PartEditorInstContains (Apple
Guide 2.1) 66
part editors (OpenDoc) 37–52
adding Apple Guide guides
to 53–68
embedding support 50
placing a guide into the Help
menu 60–61
Undo support 50–51
unloading 69–71
See also part kinds

PartInActiveDoc (Apple Guide
2.1) 66
PartInActiveProcess (Apple Guide
2.1) 66
PartInActiveWindow (Apple
Guide 2.1) 66
Part Info dialog (OpenDoc)
changing preferred kind 51
part kind pop-up menu 51
specifying part kind 45–46
translating parts 51–52
PartInNonActiveDoc (Apple
Guide 2.1) 66
PartInNonActiveWindow (Apple
Guide 2.1) 66
PartIsActiveFrame (Apple Guide
2.1) 67
part kinds (OpenDoc) 37–52
binding process 40
changing 46–47
creating documents 47–48
editor substitution 40
handling user actions 47–52
human interface principles
45–47
opening documents 48–49
part categories 41–43
preferred kind 39, 45, 48,
51
resources required 43–45
saving documents 50
standard vs. proprietary
38–39
supporting multiple 39
transferring data 49–51
translating or converting
parts 51–52
See also part editors
part viewers (OpenDoc) 37
path (Speech Recognition
Manager) 8
path objects (QuickDraw GX)
98–99
PathWalking.c file (QuickDraw
GX) 101
PathWalking.h file (QuickDraw
GX) 101
PBGetFCBInfo (Macintosh
Q & A) 109–110
PConfirmName (AppleTalk),
Macintosh Q & A 108–109
PDlog Expand (LaserWriter 8.4)
34
phrase (Speech Recognition
Manager) 8
PlacMac sample program 23

- platform kinds (OpenDoc) 44, 46, 48
 - PlayMem (MacBug) 85–86
 - PLookupName (AppleTalk), Macintosh Q & A 108–109
 - point (3D geometry) 90
 - translation of 91
 - Pointing Device Manager (QuickDraw 3D) 95
 - Polaschek, Dave 34
 - POpenSkt (AppleTalk), Macintosh Q & A 109
 - PostScript LaserPrep dictionary, LaserWriter 8.4 and 36
 - PostScript part kind (OpenDoc) 43
 - PostScript Utilities library (LaserWriter 8.4) 36
 - PowerPC
 - debugging 82–86
 - setting breakpoints in PowerPC code 85
 - PowerPlant (Metrowerks), adding speech recognition to applications with 22–33
 - PPCJump (MacBug) 85–86
 - PPD files, LaserWriter 8.4 and 36
 - PPD library (LaserWriter 8.4) 36
 - PRAM, KON & BAL puzzle 121–122
 - PREC 103 mechanism (PostScript), LaserWriter 8.4 and 36
 - Preferences and Collection libraries (LaserWriter 8.4) 36
 - preferred kind (OpenDoc) 39, 45, 48
 - changing 51
 - PrGeneral opcodes (LaserWriter 8.4) 34–35
 - “Print Hints” (Polaschek), The All-New LaserWriter Driver Version 8.4 34–36
 - PrJobMerge, LaserWriter 8.4 and 34
 - ProcPtr value, routine descriptors and 84
 - proprietary part kinds. *See* part kinds
 - protoFloatNGo (Newton Q & A) 117
 - 'prts' resource (Apple Guide 2.1) 56–59, 60–61
 - 'prts' resource test (Apple Guide) 54
 - PSetSelfSend (AppleTalk), Macintosh Q & A 105–106
 - 'ptyp' resource (QuickDraw GX), Macintosh Q & A 103
 - push-to-talk key (Speech Recognition Manager) 11, 12
 - PWriteDDP (AppleTalk), Macintosh Q & A 109
- ## Q
- 'QLfy' resource (Apple Guide) 55, 56
 - quadratic curve segments (QuickDraw GX) 98–99
 - QuickDraw 3D
 - 3D geometry 90–91
 - game controls 87–97
 - Macintosh Q & A 110
 - See also* camera movements
 - QuickDraw GX
 - implicit points 99
 - quadratic curve segments 98–99
 - ShapeWalker library 99–101
 - small font sizes (Macintosh Q & A) 103–104
 - traversing paths 98–101
 - QuickTime, KON & BAL puzzle 118–122
- ## R
- RealFont (Font Manager), KON & BAL puzzle 120–121
 - recognition-begun Apple event handler, speech recognition and 29–31
 - recognition results (Speech Recognition Manager) 7, 31–32
 - notifications 11–14
 - processing 15, 18
 - recognition system (Speech Recognition Manager) 7
 - recognizer (Speech Recognition Manager) 7
 - Reeves, Arlo 6
 - reference counting (OpenDoc) 69–70
 - RefreshViews (Newton Q & A) 117
 - RegUnionSoup (Newton Q & A) 116
 - rejection word (Speech Recognition Manager) 11, 20
 - ReleaseResource, PowerPC
 - debugging and 85–86
 - Remove (OpenDoc) 49, 70
 - Rendering Acceleration Virtual Engine (RAVE), Macintosh Q & A 110
 - Revert command, speech recognition and 27, 29, 31
 - routine descriptor, displaying 84
- ## S
- SamplePathWalker.c sample application (QuickDraw GX) 100–101
 - self-send variable (AppleTalk), Macintosh Q & A 105–106
 - SetLanguageObjectState, speech recognition and 30–31
 - SetPromiseValue (OpenDoc) 50
 - shape objects (QuickDraw GX) 98
 - ShapeWalker library (QuickDraw GX) 99–101
 - ShortenDITL, Mac OS 8
 - assistants and 79
 - ShowFoundItem (Newton Q & A) 113
 - Show/Hide Details button (Mac OS 8 assistants) 78
 - single-layer text shapes (QuickDraw GX), Macintosh Q & A 103
 - SOMObjects™ for MacOS
 - dynamic class objects 69
 - static class objects 69
 - unloading OpenDoc part editors 69, 71
 - soupDef (Newton Q & A) 116
 - “Speakable Menus?” 23
 - Speech control panel 11
 - speech-done Apple event handler, speech recognition and 31–32
 - speech objects (Speech Recognition Manager) 7–9
 - adding to an application framework 22–33
 - shutting down 33
 - starting up 26
 - See also* Speech Recognition Manager
 - Speech Recognition extension 6, 7
 - Speech Recognition Manager 6–21
 - building language models 11, 13, 16–18, 27–29
 - determining version 10
 - feedback services 7, 12, 14
 - hardware requirements 6–7
 - initializing 10–11

- initializing speech
 - recognition 9–11
- manipulating language models 18–20
- processing recognition results 15, 18
- recognition result notifications 11–14
- releasing object references 9
- speech objects 7–9
- terminating speech recognition 17
 - See also* language models
- “Speech Recognition Manager Revealed, The” (Pallakoff and Reeves) 6–21
- speech recognition objects, creating custom 24
- “Speech Recognition Tips” 12
- speech source (Speech Recognition Manager) 7
 - telephone 7
- Speech Synthesis Manager, Speech Recognition Manager and 14
- SRAddLanguageObject 11, 29
- SRAddText 11, 29
- SRCloseRecognitionSystem 8
- SRGetIndexedItem 9
- SRGetProperty 9
- SRLanguageModel 8, 9
- SRLanguageModeler tool 16, 20, 21, 27
- SRLanguageObject 8
- SRNewLanguageModel 27
- SRNewLanguageObjectFromFile 20
- SRNewLanguageObjectFromHandle 20
- SRNewPhrase 11
- SRNewRecognizer 8, 9
- SRNewWord 8, 9
- SROpenRecognitionSystem 8
- SRPath 8, 16
- SRPhrase 8, 9
- SRPutLanguageObjectIntoHandle 20
- SRRecognitionResult 9
- SRRecognitionSystem 8
- SRRecognizer 8
- SRReleaseObject 7, 9, 11
- SRSample sample application 6, 21
- SRSetLanguageModel 14
- SRSetProperty 9, 11, 18–20
- SRSpeakAndDrawText 14

- SRSpeechObject 7–9
 - class hierarchy 7–8
- SRStartListening 8, 9, 16
- SRStopListening 8
- SRWord 8, 9
- standard part kinds. *See* part kinds
- Standard Type I/O utilities (OpenDoc) 39
- static class objects (SOM), OpenDoc and 69
- stdloginto** macro (MacsBug), Macintosh Q & A 102
- 'STR#' resources
 - and the 'prts' resource 61
 - specifying 24–25
- Stuff..., Newton Q & A 115

T

- T_ACCEPTCOMPLETE (Open Transport), Macintosh Q & A 105
- TApplication, Mac OS 8 assistants and 79
- TAssistant, Mac OS 8 assistants and 79
- TCP server connections
 - Open Transport and (Macintosh Q & A) 106–108
 - passive (Macintosh Q & A) 107–108
- T_DATA (Open Transport), Macintosh Q & A 104–105, 108
- T_DISCONNECT (Open Transport), Macintosh Q & A 104
- Technical Q & A 3
- TextureEyes, Macintosh Q & A 110
- “The OpenDoc Road” (Lo), Facilitating Part Editor Unloading 69–71
- 3D acceleration hardware, Macintosh Q & A 110
- “3D Geometry 101” 90–91
- 3D geometry (QuickDraw 3D) 90–91
- T_IDLE (Open Transport), Macintosh Q & A 108
- T_LISTEN (Open Transport), Macintosh Q & A 104, 106
- TLOOK error (Open Transport), Macintosh Q & A 104
- TLookupRequest (Open Transport), Macintosh Q & A 109

- T_ORDREL (Open Transport), Macintosh Q & A 104
- T_OUTCON (Open Transport), Macintosh Q & A 108
- T_PASSCON (Open Transport), Macintosh Q & A 105
- transformation matrix (3D geometry) 91
- transition vector (TVector)
 - MaxApplZone and 82, 84
 - ReleaseResource and 85–86
- traversing paths (QuickDraw GX) 98–101
- two cursors (Newton Q & A) 113–114

U

- UDesktop::FetchTopRegular (PowerPlant) 30
- Unicode
 - Newton Q & A 115
 - OpenDoc and 38
- union soups (Newton Q & A) 116
- universal file commands (File menu), speech recognition and 27–29
- URL part kind (OpenDoc) 43
- “Using Apple Guide 2.1 with OpenDoc” (Commons) 53–68

V

- vector (3D geometry) 90
 - rotation of 91
 - translation of 91
- “Veteran Neophyte, The” (Johnson), Your Friend The Drill Sergeant 111–112
- Video Settings dialog (QuickTime), KON & BAL puzzle 119–122
- vRefNum (Macintosh Q & A) 109–110

W

- window coaches (Apple Guide 2.1) 63
- WindowRecord, file system information (Macintosh Q & A) 109
- “Working With OpenDoc Part Kinds” (Çelik and Curbow) 37–52

X

- X/Open Transport Interface (XTI), Macintosh Q & A 104, 106, 108

RESOURCES

Apple provides a wealth of information, products, and services to assist developers.

The Apple Developer Catalog and Apple Developer University are open to anyone who wants access to development tools and instruction. Additional information and services are available through Apple's Developer Programs.

The Apple Developer Catalog

offers worldwide access to development tools, resources, training products, and information for anyone interested in developing applications on Apple platforms. This complimentary catalog features hundreds of Apple and third-party development products and offers convenient payment and shipping options, including site licensing.

Apple Developer University

(DU) provides courses to get you started programming on Apple platforms, as well as advanced, in-depth training on new technologies such as QuickTime VR, QuickDraw 3D, OpenDoc, Apple Guide, and Newton. In addition to classroom training, self-paced courses are available through the *Apple Developer Catalog*, and free introductory tutorials are provided on the Web at <http://dev.info.apple.com/du.html>.

The Macintosh Developer

Program provides members with ongoing Macintosh-related technical information and services. It includes:

- The monthly Apple Developer Mailing, which includes the *Developer CD Series*.
- Macintosh technology seeding.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Newton Developer Program

provides ongoing Newton-related technical information and services. It includes:

- The monthly Newton Developer Mailing.
- The quarterly Newton Developer CD.
- Newton development class discounts.
- Programming-level technical support via e-mail. Apple offers a number of options for varying levels of technical support.

The Apple Multimedia Program

(AMP) provides resources to keep multimedia developers up-to-date on Apple's offerings for authoring and playback. It includes:

- The quarterly Apple Multimedia Information Mailing.
- Access to a special members-only area on the AMP Web site (<http://www.amp.apple.com>).
- Invitations to special events and participation in Apple events such as trade shows.
- Seeding opportunities.
- The Interactive Music Track, an extension of the AMP designed specifically for musicians, music industry members, and interactive music developers.

Apple Developer Catalog To order a product or receive a catalog, call 1-800-282-2732 in the U.S., 1-800-637-0029 in Canada, (716)871-6555 internationally, or (716)871-6511 for fax. You can also send e-mail to order.adc@applelink.apple.com, or write *Apple Developer Catalog*, P.O. Box 319, Buffalo, NY 14207-0319. The *Apple Developer Catalog* is also on the Web at <http://www.devcatalog.apple.com>.

Apple Developer University Course descriptions and schedules can be found at <http://dev.info.apple.com/du.html> on the Web. You can also call (408)974-4897, fax (408)974-0544, send e-mail to devuniv@applelink.apple.com, or write Developer University, Apple Computer, Inc., 1 Infinite Loop, M/S 305-1TU, Cupertino, CA 95014.

Apple Developer Programs These programs vary on a country-by-country basis. For more information on any of Apple's developer support programs worldwide, call (408)974-4897, fax (408)974-7683, send e-mail to devsupport@applelink.apple.com, or write Developer Support, Apple Computer, Inc., 1 Infinite Loop, M/S 303-2T, Cupertino, CA 95014.