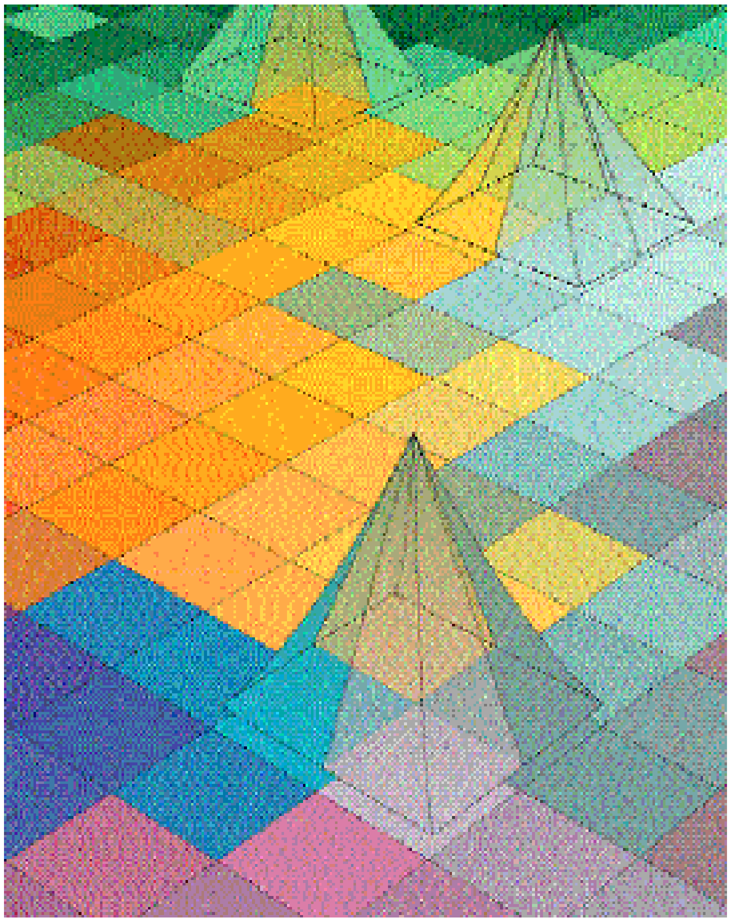
develop

The Apple Technical Journal



Issue 10 May 1992

Apple Computer, Inc.

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Review Board *Pete ("Luke") Alexander, Neil Day, C. K. Haun, Jim Reekes, Bryan K. ("Beaker") Ressler, Larry Rosenstein, Andy Shebanow, Gregg Williams*

Managing Editor *Monica Meffert*

Assistant Managing Editor *Ana Wilczynski*

Contributing Editors *Lorraine Anderson, Toni Haskell, Judy Helfand, Rebecca Pepper, Rilla Reynolds*

Indexer *Ira Kleinberg*

Manager, Developer Support Systems and Communications *David Krathwohl*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Nurit Arbel, Geoff McCormack, Dave Olmos, John Ryan*

Formatting *Forbes Mill Press*

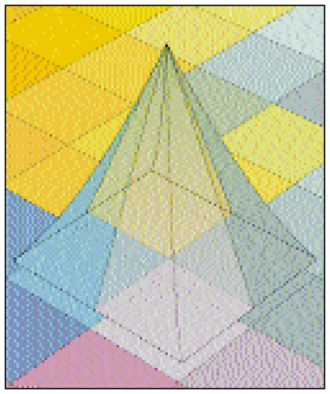
Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

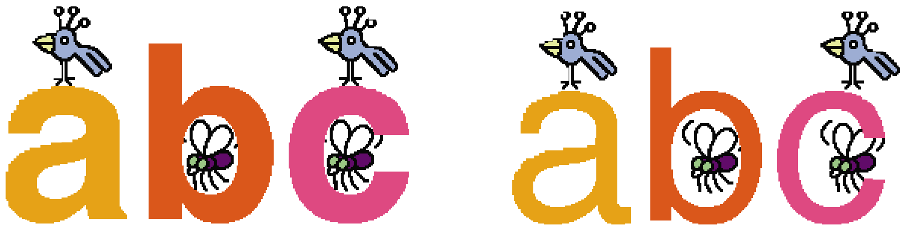
Photography *Sharon Beals*

Online Production *Cassi Carpenter*


Hal Rucker of Rucker Huggins used Adobe Illustrator and Adobe Photoshop to create this cover. The colorful tents illustrate subpixel sampling using a "tent filter," but we just think they're beautiful to look at.

develop, The Apple Technical Journal, is a quarterly publication of the Developer Support Systems and Communications group.

The *Developer CD Series* disc for May 1992 or later contains this issue and all back issues of *develop* along with the code that the articles describe. The contents of this disc, which includes other handy software and documentation, can also be found on AppleLink.


See Macintosh Q & A, page 110.

CAROLINE ROSE

Dear Readers,

We've made a change to *develop* that I'd like to draw to your attention for two reasons: (1) observant long-time readers may wonder why we're reverting to an old, abandoned practice, and (2) we've learned a lesson that also applies to the products you're developing.

Issues 1 through 4 of *develop* were "subtitled" January 1990 through October 1990. Our fifth issue started out 1991 not only with the since-abandoned "Vol. 2" designation, but also with the subtitle "Winter 1991." When I started this job in February 1991, I learned that the change to seasons was made because of the uncertainty of just when an issue would fall into developers' hands. But now we're switching back to months with this, our May 1992 issue.

There's the problem that it's not real clear which year any given winter belongs to, since that season spans two years. But the worst offense is that winter doesn't hit all parts of the globe at the same time. So we were confusing and offending some of our Australian developers, for example (see the first letter in the Letters section).

Localization is something that's of course critical to any products you hope to sell in other countries—sometimes even other cities or states. And it applies to both code and documentation. Way back when we were writing the first Macintosh user manual, we were surprised to get feedback that we shouldn't refer to pizza in our sample text because it was too regional. (Somehow the problem of having only white males in all the photographs was overlooked, but that's another story.)

If you're thinking that people in other countries never use your software anyway, consider this: In the song "Talkin' Wheelchair Blues" by folk singer Fred Small, a woman in a wheelchair has extreme difficulty getting into a restaurant. She tells the owner that there are things he can do to make it easier for folks in wheelchairs. The owner replies, "Oh, it's not necessary. Handicapped never come here anyway." The moral is that if you essentially shut some potential users out, of course they won't use your software.

Apple can offer some new resources to you in your quest for localization. The *Guide to Software Localization* will be replaced by a *Guide to Macintosh Software Localization*, to be published by Addison-Wesley and available by around late July. Besides the

CAROLINE ROSE (AppleLink: CROSE) has been writing computer documentation for more than half her life. She harks back to the days when using "you" in a manual was controversial. Caroline hasn't stood still: she switched from writing and programming (at Tymshare, R.I.P.) to writing and editing (*Inside Macintosh*, at Apple) to managing and editing (at NeXT) to editing *develop* (couldn't stay away). Besides her job at Apple, Caroline loves almost all kinds of music, dancing, and reading. Her eclectic tastes are typified by the three books she's involved in at the moment: a history of *MAD* magazine, an annotated *Hamlet*, and an Italian language textbook. A New York City transplant, Caroline thinks that nothing compares to the *New York Times Book Review* or *real* New York pizza. She's occasionally nostalgic for knishes.•

subtle title change, the difference between these two is that the new book will cover third-party script support as well as script systems directly supported by Apple. And by the time you read this, there should be a new APDA product, called *Localization for Japan*, that covers the business and technical aspects of getting software into the Japanese market. You'll of course get all the latest technical details on worldwide software in the forthcoming new, improved *Inside Macintosh*.

Another change starting with this issue is that there's no longer an Apple II Q & A section. Instead, we'll pass on those Q & A's to the Apple II journal *A2-Central* (published by Resource Central). We think this is a better way of getting the information out to the majority of Apple II developers. All current Apple II development products previously sold through APDA are now sold through Resource Central. For more information, you can phone them at (913)469-6502.

Now on to Issue 9's trivia question. Here it is again: The original hardcover *Inside Macintosh* Volumes I-III had a running pattern of Macintosh computers across its endpapers . . . what broke this pattern, and why? The first two correct replies came from Tom Bernard of Bersearch Information Services and Bill Lipa of CODAR Ocean Sensors. The answer is that the screen of the last Macintosh in the back of the book contains my favorite character in the Cairo font, a rose, and it represents yours truly, the editor and principal author of that tome. The tricky part of the answer is that the rose was a surprise to me. My boss arranged for it to be put there; I didn't know about it until he handed me my first hardcover copy, hot off the press, and slyly asked me to lift up the inside end flap.

This time I've got a puzzle for you font fiddlers out there—a test of just how good your font-observing eye is: What character in *develop*'s body font is upside-down (not just one-time-only, but defined that way)? Clue: There are lots of them in this issue, but none in this editorial. Get out those magnifying glasses!

C Rose

Caroline Rose
Editor

SUBSCRIPTION INFORMATION
To subscribe to *develop*, use the subscription card in the back of this issue. Please address all subscription-related inquiries to *develop,* Apple Computer, Inc., P.O. Box 531, Mt. Morris, IL 61054 (or AppleLink DEV.SUBS).•

BACK ISSUES
For information about back issues of *develop* and how to obtain them, see the last page of this issue. Back issues are also on the *Developer CD Series* disc.•

LETTERS

PROVINCIALISM

I would be pleased if Apple (and APDA, etc.) would cease dating publications by what would seem to be the season in the northern hemisphere (for example, "Winter 1992"). This practice is inconsistent with Apple's dedication to Worldwide Software and the avoidance of cultural values. In addition to it being ignorant and arrogant, it is both confused and confusing—the seasons in the U.S.A. are not, I believe, simply the opposite of ours.

Furthermore, what the heck is "Winter 1992"? Which end of the year is it? Is it sometimes also known as "Winter 1991"?

—Dr. Ross L. Richardson, Australia

Thanks for the kick in the pants—and for inspiring the theme of this issue's editorial. The change (back) to using months has been made in this, our May issue. I wish we could have done it for Issue 9 (February, a.k.a. Winter 1992), because I agree that "Winter" is particularly confusing, but the timing was too tricky. In fact, just after hearing from you about this, I received a "Winter 1991" catalog in the mail!

—Caroline Rose

WHERE'S THE SKB STACK?

Wow, a real Developer CD! At first I thought it was one of those errors that are in my favor (a rarity for sure!). But no, I will continue to receive Developer CDs. Last year I would have considered murder to gain access to one of these.

Needless to say I've been over it as much as possible. There's one thing that I can't find. I can't seem to locate the Search Knowledge Base stack. I was able to select which stacks to search and get the report of all Tech Notes containing what I was looking for. I don't mind telling you that it saved my butt on a couple of contracts. I've looked in the logical places on the CD for it. Where is it?

—J. R. Hughson

Yes, since Issue 8, subscribers to develop *have been getting the entire Developer CD Series disc and not just the Developer Essentials subset. Truly the bargain of the century (tell your friends). We're glad you didn't have to resort to foul play to get one.*

The Search Knowledge Base (SKB) stack was modified slightly and integrated into the Developer Info Assistant (DIA) stack—so it was only hiding, not gone. We apologize for not communicating this change to you. To go directly to the SKB utility, just open the DIA stack (in the Start Here folder) and click the "Search..." button.

—Caroline Rose

ON LOCATION (MOOF!)

It was nice to get more goodies on the CD that came with Issue 8, but the CD has a couple of deficiencies that I think should have been detected/eliminated.

1. Unless something is wrong with my copy of On Location, the On Location index on the CD can't be used to view the files it references. On Location says that the index is out of date. I don't know about all files, but this was true of the sample I tried to look at. To rebuild the index on my hard disk will take a big chunk of my limited hard disk space.

2. The time I spent with the CD from *develop* Issue 7 getting accustomed to

PLEASE WRITE!
We welcome timely letters to the editors, especially from readers reacting to articles that we publish in *develop*. Letters should be addressed to Caroline Rose (or, if technical *develop*-related questions, to Dave Johnson) at Apple Computer, Inc., 20525 Mariani Avenue, M/S 75-2B, Cupertino, CA 95014 (AppleLink: CROSE or JOHNSON.DK). All letters should include your name and company name as well as your address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did).•

the Search Knowledge Base stack was for naught since SKB is not present on the CD that came with Issue 8. The old version does not work on the new CD without substantial modification. I found Issue 7's SKB very useful.

By the way, what does "Moof!" mean?

—Pete Roberts

The On Location index was indeed broken on the Developer CD Series disc Volume X. Our apologies, and thanks for alerting us. It works fine on later CDs.

As for the apparent disappearance of the Search Knowledge Base (SKB) stack, please see the answer to the previous question.

Moof is a sacred tradition among Apple's Developer Technical Support engineers. It means several things. First, it's the sound that the dogcow makes. (The story of the dogcow is "hidden" in Macintosh Technical Note #31 on the CD; if you can find it, you'll be considered a Moof initiate.) The dogcow is integrated into our artwork in a number of places. Take a look at the cover card of the Technical Notes stack or Q & A stack, for example. There's also a guest appearance of the dogcow in this issue of develop *(see the Print Hints column).*

Second, the word is used to indicate any software that's a hack, something untested and on the edge. Certain folders on the CD are marked Moof, such as "Tools & Apps (Moof!)" and "Development Platforms (Moof!)." This is to let you know that these folders contain software that's not fully tested or sanctioned by the powers that be. When you open these folders you cross the boundary into hackerland.

—Sharon Flowers

QUICKTIME MOVIES ON THE CD?

I want to thank you and your staff for a truly wonderful publication. I currently hold a dual position as a Macintosh/Windows developer and as director of a research lab using UNIX® boxes with Motif; of all the trade magazines I get (and let me tell you, I get a zillion of them), *develop* shines above all the rest. The articles addressed aren't always the ones I'd have chosen, but with as large an audience as you probably have, you still manage to print more than enough for me to get plenty out of each issue.

The CD has also provided me with hours of exploration each time. I would really like to see you include QuickTime movies made by Apple employees and maybe even readers.

Keep up the good work—and treat yourselves to pizza on me (send the bill to my mailing address).

—Robert H. Zakon

Thanks for the glowing comments. You have no idea (or maybe you do) how much these things mean to us.

Regarding QuickTime movies on the CD, I agree that would be great, sorta like the old days when we had an audio track on every CD: our chance to get creative and add some entertainment value. Unfortunately, it's unlikely that it will happen, for a few reasons. The main one is that we'd have to get rid of a lot of useful stuff to make room for the movies (which, as I'm sure you know, tend to be largish). Even though we have 600 MB of space on the disc, it's always burstingly full.

So despite the fun we could all have with movies, the Developer CD is just not the place for them, alas. The QuickTime CD is, *of course, chock full of fun movies, if you haven't seen it yet. It's available from APDA as part of the QuickTime Developer's Kit, APDA #R0147LL/A.*

Thanks again for writing, and if you have ideas for articles you'd like to see (or want to write one yourself!), please don't hesitate to let us know.

—Dave Johnson

P.S. The pizza was delicious. All 114 of us thank you heartily. The bill is in the mail.

NOT READY FOR PAPERLESS

I'm more concerned about the environment than most, but the race to achieve the paperless office, without determining what people's needs really are, will only end up convincing them that the electronic office has not come of age! Apple isn't considering that we've spent most of our lives looking at the printed page, so the productivity loss now in using on-line electronic media is high. The fact is most people relate more easily and absorb information more quickly from the printed page than from any existing on-line method.

A case in point is *develop.* Scanning Issue 8 on-line took more than 30 minutes, just for a quick scan. An equal amount of information could have been obtained from the printed magazine in less than 1 minute. That's a factor of 30!

I applaud the effort to save trees, but how many trees are being saved when developers merely print the information themselves (single-sided, I might add)? If Apple wants to make an impact in this area of environmental concern I suggest they provide a set of integrated on-line tools that allow retrieval of information in a way that's more human oriented. Apple held a carrot under our noses at the 1991 Worldwide Developers Conference called BlueNote, but they've been unable to deliver. In the meantime we're expected to make do with the existing (poor) HyperCard® utilities. Come on, Apple—now is the time to make a difference.

—Todd Stanley

I couldn't agree with you more that Apple needs to develop an electronic publishing strategy that advances the art of access to information. But Apple is many people and consensus is somewhat of an endangered species! Current strategies, electronic develop *included, do make some advances. Regretfully, in some ways they are more difficult to use than the tried-and-true print media.*

But beyond regret and apologies we actually have some solutions in the works. The "BlueNote" carrot that was dangled before you is in fact now in use on the Developer CD. It's being used to present chapters from the new, improved Inside Macintosh. Unfortunately, like most solutions to date, it creates as many problems as it solves. So please be patient as we try to come up with a solution that does the electronic medium justice.

As a side note: This discussion should send a message to the developer community. There's clearly a need in the marketplace for generic electronic publishing tools. And where there is a need, there is opportunity.

—Corey Vian

WHERE'S PRINTED DEVELOP?

I'm an Apple Associate and lately I have not been receiving the printed version of *develop* magazine. Issue 7 was the last printed one I received. I don't know why this happened, but I would like to receive the printed version again.

The CD-ROM paperless approach is admirable, but it doesn't work for me because I do all my reading while commuting on CalTrain.

—Joe Zuffoletto

Letters like yours are all too common. What's happened is that, starting with Issue 8, printed develop *was removed from the mailing to Apple Associates and Partners. In various places (including in the mailing itself), it was announced that developers would have to explicitly subscribe to keep getting* develop *in printed form—but apparently many developers didn't see this announcement. Since then we've received a lot of less-than-positive feedback about the decision to drop the printed magazine out of the mailing and about the difficulties in trying to read it on the CD (see, for example, the previous letter). The paperless approach isn't working for a lot of people, as it turns out.*

For now, you'll have to subscribe in order to get printed develop*. If you don't want to go through the trouble of finding the subscription form in* develop *on-line, you can place a phone order at 1-800-877-5548—or you can subscribe through APDA.*

As for the future, rest assured that Apple is paying attention to the feedback and has learned a lot from it. I can't say what will happen, but I know that the more developers tell us what they want, the more likely they are to get it. So thanks for writing to us.

—Caroline Rose

UP TO HIS EARS IN CDS

I agree 105% with your editorial comment in Issue 8 (about preferring to read *develop* in printed form). I guess I'll cast my vote by subscribing to *develop*, but I need another copy of the CD like I need a typewriter. I sometimes wonder whether I should still keep all the old CDs that compete for room in my limited collection space. I want the hard copy *develop*, but it bothers me to waste that CD that comes with it.

—Bruce Radford

Apple Associates and Partners who subscribe to develop *to get it in hard copy do end up with an extra CD. At one time I was concerned about the extra cost of this as well as the waste. It turns out that the CD doesn't add much at all to the cost of* develop*. But as for the waste, I don't have an answer; maybe some of our readers do?*

I wouldn't recommend holding on to the old CDs (which only compounds your disposal problem, but oh well). For the most part they're cumulative—we've only deleted old versions of international system software and a few other things that we had pressing reasons to remove. In particular, the develop *code on the CD is kept up-to-date, and any bugs are fixed, with each new CD.*

—Caroline Rose

APPLE EVENT OBJECTS AND YOU

With Apple events, Apple has opened the door for applications to control each other and work collaboratively. However, before applications can communicate, they have to agree on the commands and data they'll support. Apple event objects form the basis of such a protocol—the Apple event object model. The object model is powerful, but still a source of confusion for many developers. This article provides an overview of the object model and answers several commonly asked questions, including "What is the Apple event object model?" and "How do I support it?"

RICHARD CLARK

One of the greatest strengths of the Macintosh—its graphical user interface—is also the basis of one of its greatest weaknesses—the difficulty of automating routine or repetitive tasks. "Give us batch files!" many users cried. The developers responded with macro programs such as QuicKeys and Tempo, which handle many of the routine tasks but can't always make a program do *exactly* what the user wants.

The problem is that macro programs are generally limited to manipulating an application's human interface and have limited information about the state of the application. This means that if some setting has been changed or something has been moved, running a particular macro might not have the desired effect. In other words, one Macintosh application cannot control another application reliably through the target application's human interface.

For one application to control another application reliably, all of the following must happen:

- The two applications must agree on a protocol for sending commands and data and agree on the specific information to be sent across this connection.
- The controlled application needs to provide a rich enough set of commands and sufficient access to its data so that meaningful work can be done.

RICHARD CLARK, an instructor and course designer in Apple's Developer University, is no stranger to projects both large and small. (He claims that both of his recent projects—the new Advanced System 7 class and Daniel Guy Clark—took around nine months and developed a life of their own.) When he's not playing with his new son, you can find him dancing in local Renaissance Faires, stunt kite flying, searching for the ultimate chocolate recipe, and dreaming up horrible new puns.•

- The protocols and command sets should be standardized so that many different applications can work together.

On the Macintosh, Apple events and the *Apple Event Registry* provide the standards that allow applications to control each other reliably. The Apple event, a standard protocol for sending commands and data between applications, was introduced as part of System 7. The *Apple Event Registry* defines standard Apple event commands and two standard data types—*Apple event object* and *primitive.* Apple event objects describe an application's internal data, and primitive types describe the data that can be sent between applications. In essence, the Registry forms the basis for a standard language that applications can use when sending or receiving Apple events.

One of the challenges in creating the *Apple Event Registry* was to keep the set of commands small while providing an adequate level of control between applications. The Registry does this by allowing the same command to apply to different Apple event objects within an application. The application of Apple events to Apple event objects is commonly referred to as the *Apple event object model.*

This article provides an overview of the object model and then discusses how you can add object model support to your application. The fundamentals of Apple events are given in *Inside Macintosh* Volume VI.

OBJECT MODEL BASICS

The *Apple Event Registry* defines an application's programmatic interface as a series of Apple event objects, where each object belongs to a particular object class. Each Apple event object is comprised of some data and a set of Apple event commands that operate on that data. In a traditional object-oriented fashion, new classes are defined by taking an existing class and adding new data and/or commands. Related classes are grouped together into *suites.*

The most commonly used objects (and their associated commands) are grouped together into the Apple event *core suite.* The commands in the core suite, which include Create Element, Delete, Get Data, and Set Data, cover the basic operations for any given object. The Apple event objects defined within the core suite include documents, windows, and the application itself. The core suite also includes some primitive classes such as long and short integers, Boolean values, and text. Every object model–aware application should support the core suite, and all Apple event objects defined within your application should support the core suite events.

The data portion of an Apple event object is broken into two parts: the object's *properties* and its *elements:*

- The *properties* of an object contain the attributes of the object—for example, its name and a 4-byte code designating its class.

Concepts from object-oriented programming (notably inheritance—the process of defining new classes in terms of other classes) are used in defining the Apple event object model, but supporting Apple event objects does not require the use of an object-oriented language or class library. You can use any language or implementation technique you want, as long as your application can understand the Apple events sent to it.•

- The *elements* of an object are the other objects (in other words, data) that it contains. For example, a drawing application contains one or more documents, and each document may contain several rectangles and a picture or two. When the Registry describes an object, it lists all the element classes of an object, but a particular object may contain only some (or no) elements of each class at run time. (The number of elements can change during run time. For example, the number of words in a window could increase due to user typing or an incoming Apple event.)

For more detail on the difference between a property and an element, see "Properties and Elements."

Figure 1 shows three object classes that we'll use throughout the article; they've been derived from the *Apple Event Registry* and simplified for the purpose of illustration.

	cDocument	**cRectangle**	**cWord**
Properties	pClass	pClass	pClass
	pDefaultType	pDefaultType	pDefaultType
	pName	pBounds	pFont
	pIsModified		pSize
			pStyle
Elements	cFile	(None)	cCharacter
	cRectangle		
Apple Events	Create Element	Create Element	Create Element
	Get Data	Get Data	Get Data
	Set Data	Set Data	Set Data
	Delete	Delete	Delete
	Open		
	Close		
	Print		
	Save		

Figure 1
Some Hypothetical Apple Event Object Classes

OBJECT SPECIFIERS

Most of the Apple events defined in the *Apple Event Registry* contain one or more *object specifiers* as parameters. An object specifier is similar to the instructions you might give someone who's looking for a particular house: turn left at the first signal, then look for Jones Street and turn right, then travel down to the third house on the right. Object specifiers can also be used to specify a group of objects—for example, every green house on Jones Street.

In addition to the core suite, the *Apple Event Registry* includes other specialized suites for text processing, database manipulation, manipulating QuickDraw graphics, and the like. Application developers can define their own custom Apple event object classes and suites and submit them to the Apple Events Developer Association for standardization.•

PROPERTIES AND ELEMENTS

Each Apple event object contains exactly one of each of its properties (each of which has a name), so you might ask for the "Bounds of the frontmost Window" and receive back the pBounds property of the specified window. An object can contain zero or more of each of its element classes (each of which has a name), so you could ask for "every Paragraph in Document 1," where Paragraph is a valid element class for the document.

Many developers want to know when you should declare something as a property and when you should declare it as an element. You should make something (call it *x*) a property of an object when *x* describes something about that object. You should make something else (call it *y*) an element of an object if *y* is contained within the object.

Some developers use the rule "If there's only going to be one *y* in the object, make it a property." Alas, this rule isn't always correct. Let's assume that an application could display only one document window at a time. Should that document be an element or a property? According to the Registry's definition of an element, since the document is contained within the application, you should make it an element. If you make something an element based simply on the Registry's definition, your new classes will be consistent with the existing classes.

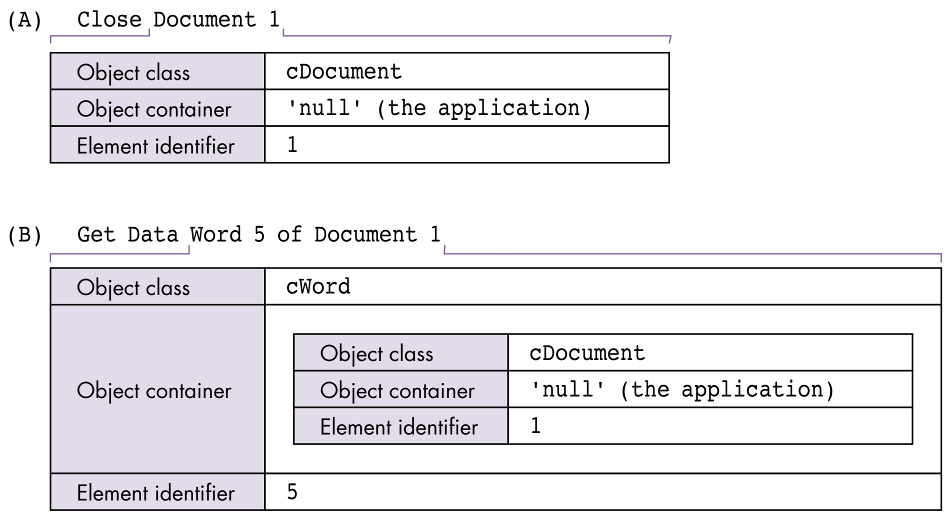
Another useful test is to ask "Can I delete this item?" If you can, it's not a property. (You can delete a window from within an application, so a window is an element of that application, not a property. But since you cannot delete the bounds of the window, the bounds is a property.)

Or imagine you send an Apple event–aware word processor the object specifier "every Paragraph in the current Document that contains the Word 'Apple'." The application would search in stages, first finding the current document and then searching through the paragraphs one at a time to see if they contained the word "Apple." Object specifiers provide a powerful general mechanism for locating a particular object in an application.

The Apple event's direct parameter typically contains the object specifier, yielding such commands as "Close Document 3" and "Delete Word 3 of Document 'fred'." Passing an object specifier as part of a command allows the same command to be reused for different objects (New *window*, New *document*, or New *rectangle*) instead of inventing a unique command for each action-object pair (NewWindow, NewDocument, or NewRectangle).

Internally, an object specifier consists of a series of recursive "get a particular element of class *x* from object *y*" commands. For example, in the command "Close Document 1," the object specifier (Document 1) is represented as "the first object of class Document contained within the Application." Another way of looking at this is "(the first object of class Document in (the Application))" where the parentheses represent one object specifier embedded within another. In addition to specifying a single element, an object specifier can refer to a property of some object or to a set of objects. For example, your application may receive the object specifier for "the Bounds of Window 1" or "every Icon contained within Rectangle 1 of Window 5."

Figure 2 shows a simplified representation of two object specifiers. Object specifiers are stored as Apple event records, with one field each for the object class and the object's container (stored as a handle) and two fields for the *element identifier*. The two fields of the element identifier together represent the specific element to be selected. In part A of Figure 2, the desired object class is cDocument, the container is 'null' (in other words, a descriptor that has type typeNull and a nil handle), and the element identifier is 1. The null container typically represents the application. In part B, the desired object class is cWord, the container is a handle to the object specifier from part A, and the element identifier is 5.



(A) `Close Document 1`

Object class	`cDocument`
Object container	`'null' (the application)`
Element identifier	`1`

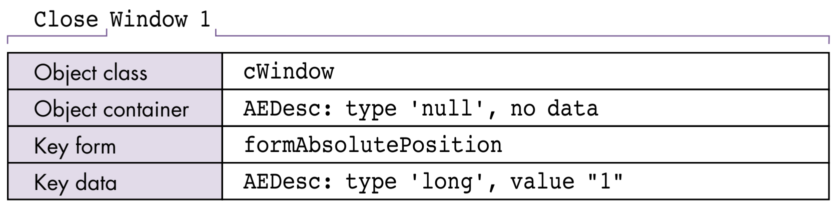
(B) `Get Data Word 5 of Document 1`

Object class	`cWord`
Object container	Object class: `cDocument`; Object container: `'null' (the application)`; Element identifier: `1`
Element identifier	`5`

Figure 2
Simplified Representation of Object Specifiers

An actual object specifier is slightly more complicated than the ones shown in Figure 2. In the examples given above, we've consistently referred to elements by number. However, you might want to refer to some object, such as a document, by name. In that case you would need to know that the two fields of the element identifier contain a *key form* and some *key data*.

Each different way you can refer to an element uses a different key form. When we refer to an element by number, we're using the "absolute position" key form. We could also specify a "name" key form, a "property" key form (to get a property of an object instead of one of its elements), and so on. A complete object specifier is shown in Figure 3. A list of all standard key forms is given in the *Apple Event Registry* and in the Apple Event Manager chapter of the new, improved *Inside Macintosh* (preliminary draft) on the *Developer CD Series* disc.


Close Window 1

Object class	`cWindow`
Object container	`AEDesc: type 'null', no data`
Key form	`formAbsolutePosition`
Key data	`AEDesc: type 'long', value "1"`

Figure 3
The Four Fields of an Object Specifier

HOW DO I DISPATCH AN APPLE EVENT CONTAINING OBJECT SPECIFIERS?

One of the side effects of the object model is that the same command will be executed differently depending on the type of object involved. Therefore the object class, event class, and event ID are required before you can dispatch an Apple event. Since the Apple Event Manager uses only two of these values when dispatching an Apple event (the event class and event ID), you'll need to write some additional dispatching logic.

We'll discuss three major ways of dispatching object-model Apple events: an event-first approach, an object-first approach, and a method that uses a lookup table to dispatch the events. These approaches all serve the same function—extracting an object specifier and using the combined object class, event class, and event ID to select one of the application's routines. They differ only in the way you structure your code.

AN EVENT-FIRST APPROACH

The event-first approach allows the Apple Event Manager to do most of the work. The Apple Event Manager calls a different handler for each event—for example, Get Data and Set Data—and that handler calls different routines depending on the object class given by the object specifier. Figure 4 and the following sample code illustrate this approach.

```
pascal OSErr AESetDataHandler (AppleEvent *message, AppleEvent *reply,
   long refCon)
{
   OSErr    err;
   AEDesc   theObject, theToken;

   err = AEGetKeyDesc(message, keyDirectObject, typeObjectSpecifier,
      &theObject);
   if (err != noErr) return err;
```

For source-code samples that use the event-first technique, see the samples Quill and AEObject-Edition Sample in the Apple Events and Scripting Development Kit on the *Developer CD Series* disc.•

```
    err = AEResolve(&theObject, kAEIDoMinimum, &theToken);
    AEDisposeDesc(&theObject);
    if (err != noErr) return err;

    /* The token is an Apple event descriptor. For now, we can */
    /* assume that the token's descriptor type is the class of the */
    /* object that should handle this event. */
    switch (theToken.descriptorType) {

        case cWindow: case cDocument:
            err = Win_SetData(&theToken, message, reply);
        break;

        case cRectangle:
            err = Rect_SetData(&theToken, message, reply);
        break;

        case cWord:
            err = Word_SetData(&theToken, message, reply);
        break;

        default:
            err = errAEEventNotHandled;
    }
    AEDisposeDesc(&theToken);
    return err;
}
```

An application that processes events using the event-first approach goes through the following steps after it receives an Apple event and calls AEProcessAppleEvent (the numbers correspond to the numbers in Figure 4):

1. The Apple Event Manager locates the event in its dispatch table.
2. The appropriate handler routine is called by the Apple Event Manager—in this example, it's Set Data. This handler routine needs to determine the object class before it can perform the appropriate action, so it calls AEResolve to convert the object specifier into a reference to a particular object.
3. AEResolve takes an object specifier as input, and calls one or more accessor routines to convert this object specifier into a token that refers to some object. (See the section "How Do I Resolve an Object Specifier?" for more information.)
4. The token is returned to the handler.

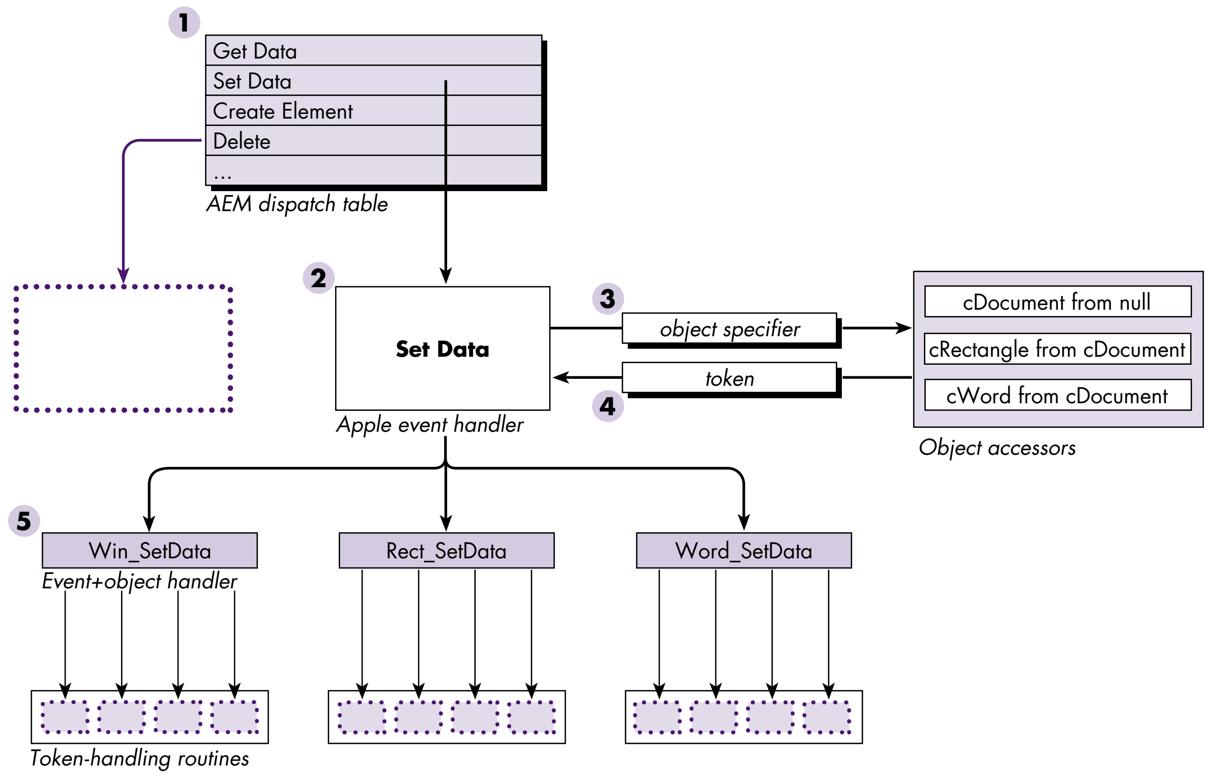


Figure 4
Event-First Approach to Dispatching Apple Events

5. Once the handler knows the object class, it can call the appropriate object-specific routine. This routine typically accepts the token as one of its parameters.

Since many of the things you can do with a token fall into a few basic operations, such as reading, writing, inserting, or deleting the information represented by a token, you can choose to write a set of token-handling routines for each token type that you define. Token-handling routines are not required, but they are useful. (See the section "What Are Token-Handling Routines?" for more information.)

Due to its simplicity, the event-first approach is recommended for all applications written in a procedural programming style (as is typically done in C or Pascal). Its only real drawback is that if you add a new object class to your application, you have to modify a number of Apple event handlers to recognize the new class (one handler per event that the new object class supports).

If you have code spread across several source files, consider whether this could present a code maintenance problem. If so, the object-first approach might work better for your application.

AN OBJECT-FIRST APPROACH

You can limit the amount of work required when adding a new object class by making each object class a self-contained unit. In this approach, an individual file (or group of files) contains all the code required to implement a single object class, including the event-dispatching code, object accessors, and token handlers. (For more information on token handlers, see the section "What Are Token-Handling Routines?")

Since the object includes its own event-dispatching code, you don't usually install a separate handler for each individual Apple event. Instead, you install one or more wild-card handlers that route the event to the appropriate object using the following algorithm:

1. Extract the parameter containing the object specifier.
2. Call AEResolve to convert this object specifier into a token.
3. Extract the object class from the token.
4. Call the event dispatcher within the appropriate object.

Since most Apple events carry their object specifiers in the direct parameter, a single wild-card handler works for all of these Apple events. However, there are some events that carry their object specifiers in different places, so you need to install specific handlers for these events. (For example, the Create Element event carries its object specifier inside an insertionLoc structure.) Using a single handler that uses the first object specifier it finds is inadequate, since some events use multiple object specifiers and an object specifier can appear anywhere another parameter can.

The handler that extracted the object specifier passes the token, the message, and the reply event to the object's central event dispatcher. This dispatcher then calls the appropriate routine, which typically calls one or more token-handling routines. This approach is illustrated in Figure 5 and in the following sample code.

```
/* This is a typical Apple event handler that you install using */
/* a wild card (in this case, the class = 'core', and the event */
/* ID = '****'). This would go in a "common area" file, separate */
/* from the individual object implementation files. */
pascal OSErr AECoreSuiteHandler (AppleEvent *message, AppleEvent *reply,
   long refcon)
{
   OSErr    err;
   AEDesc   directParam, theToken;
   /* The following code works for all core Apple events except */
   /* Create Element. Either this routine would need to be modified */
   /* for Create Element, or a specific handler installed. */
   err = AEGetKeyDesc(message, keyDirectObject, typeWildCard,
      &directParam);
   if (err != noErr) return err;

   if (directParam.descriptorType == 'null') {
   /* AEResolve doesn't like null descriptors, so skip it. */
      theToken = directParam;
   }
   else {
      err = AEResolve(&directParam, kAEIDoMinimum, &theToken);
      AEDisposeDesc(&directParam);
      if (err != noErr) return err;
   }
   /* We assume the token's type is the class that handles this event. */
   switch (theToken.descriptorType) {
      /* Include one entry for each object class. */

      case 'null':
         /* This is the application object's token class. */
         err = AppEventDispatcher(&theToken, message, reply);
      break;

      case cDocument:
         /* See the example of this routine below.*/
         err = DocumentEventDispatcher(&theToken, message, reply);
      break;

      /* And so on for cRectangle, cWord, etc. */

      default:
         err = errAEEventNotHandled;
   }
   AEDisposeDesc(&theToken);
   return err;
} /* AECoreSuiteHandler */

/* ===In the Document Object file...=== */

OSErr DocumentEventDispatcher (AEDesc *theToken, const AppleEvent *message,
   AppleEvent *reply)
{
   OSErr       err = noErr;
   AEEventID    eventID;
   OSType       typeCode;
   Size         actualSize;

   /* Get the event ID. */
   err = AEGetAttributePtr(message, keyEventIDAttr, typeType,
      &typeCode, (Ptr)&eventID, sizeof(eventID), &actualSize);
   if (err != noErr) return err;

   switch (eventID) {

      case kAECreateElement:
         err = Doc_CreateElement(theToken, message, reply);
      break;

      case kAEGetData:
         err = Doc_GetData(theToken, message, reply);
      break;

      /* And so on for Set Data, Delete, Open, Close, Print, etc. */

      default:
         err = errAEEventNotHandled;
   }
   return err;
} /* DocumentEventDispatcher */
```

When an event is processed using the object-first technique, the application takes the following steps after it receives an Apple event and calls AEProcessAppleEvent (the numbers correspond to the numbers in Figure 5):

1. The Apple Event Manager locates a handler routine in its dispatch table. The handler is usually installed with a wild-card value so that it's passed all (or most) events.
2. The appropriate handler routine is called. This routine acts as an object dispatcher—it determines the type of object involved and calls the code in the appropriate object's source file. This handler routine needs to determine the object class, so it calls AEResolve to convert the object specifier into a reference to a particular object.
3. AEResolve takes an object specifier as input, and calls one or more accessor routines to convert this object specifier into a token that refers to some object. (See the section "How Do I Resolve an Object Specifier?" for more information.)

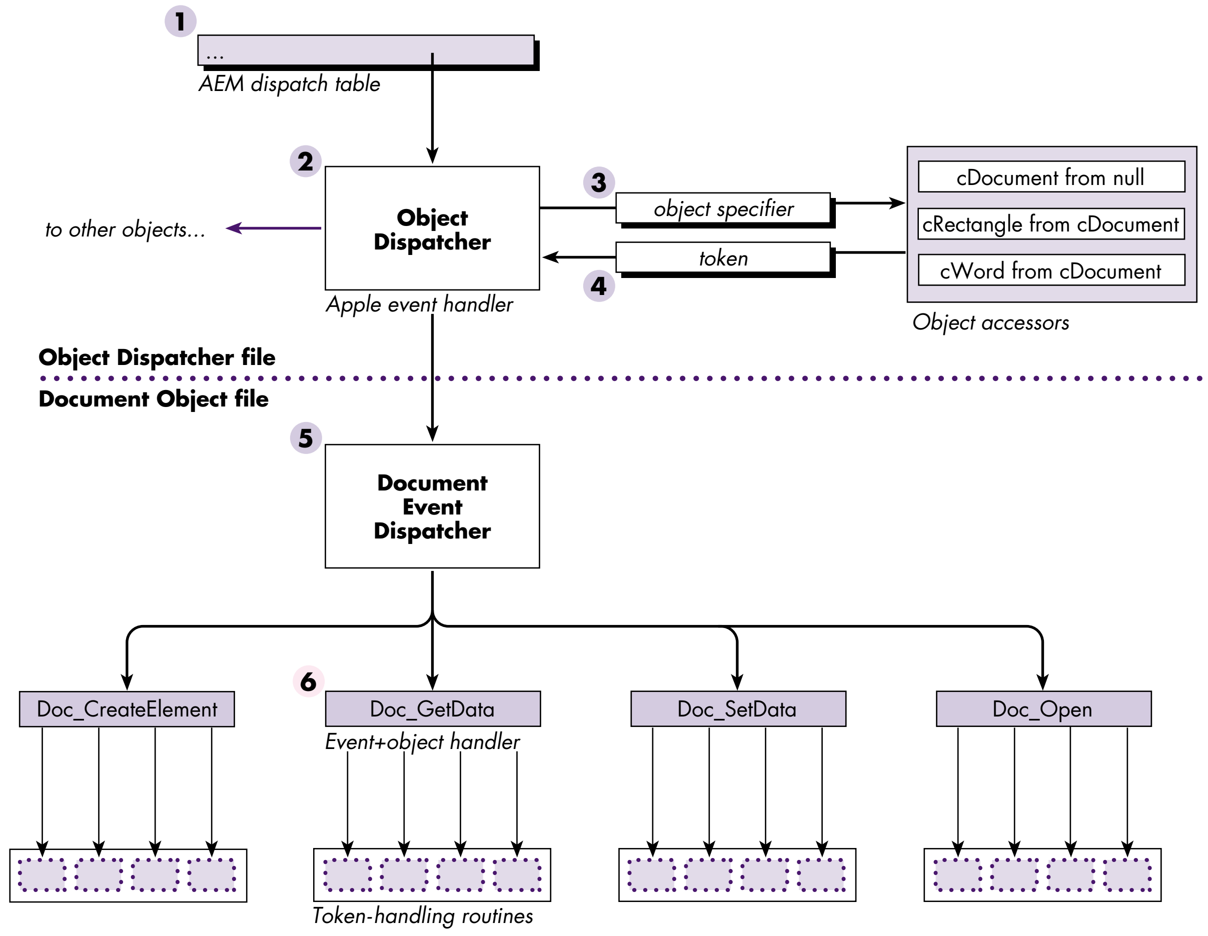


Figure 5
Object-First Method for Dispatching Apple Events

4. The token is returned to the handler.
5. Once the handler knows the object class, it can call the appropriate object's event dispatcher. The dispatcher looks at the event's class and ID and calls the appropriate routine.
6. The called routine performs a task specific to the event class, event ID, and object class. It typically accepts the token as one of its parameters.

This approach, or some variant of it, could be implemented using object-oriented programming and is recommended for object-oriented applications.

If you use the object-first approach in a procedural application, you can still get some of the benefits of object-oriented programming, since this technique can be used to implement a simple form of inheritance for Apple event objects. If a particular object's event dispatcher doesn't recognize an event, it can pass the event to its superclass's event dispatcher. If that dispatcher doesn't recognize the event, the request can be passed up the chain until the topmost dispatcher is reached (typically cObject). This minimizes the code required for adding a new object, since an object only needs to implement its unique events (and any standard events that it handles differently) and can pass all other events to its superclass.

One drawback to this approach is the overhead involved in dispatching the event. Each event goes through the Apple Event Manager, AEResolve, a pair of switch statements (one in the top-level Apple event handler, and another in the object's dispatch routine), and possibly a couple of superclass event dispatchers. Still, each of our approaches requires the initial use of the Apple Event Manager and a call to AEResolve, so the added overhead lies primarily in the switch statements.

Another drawback is that each Apple event typically has several parameters, and each Apple event handler needs to extract the set of Apple event–dependent parameters for that Apple event. This can lead to redundant code.

TABLE-BASED DISPATCHING

One way to lower the overhead associated with dispatching object-model Apple events involves building a dispatch table of your own to replace the Apple Event Manager's. The Apple Event Manager constructs a two-way hash table based on the event class and event ID. Since this isn't enough information to properly dispatch an object-model Apple event (you also need to know which object class will be responsible for handling the event), the solution is to construct your own table using a three-part index (event class, event ID, and object class) that contains the addresses of the appropriate routines.

As in the object-first example, this dispatcher should be "attached" to the Apple Event Manager through a wild-card handler in the Manager's regular dispatch table. (This is necessary since there's no other robust way to "unpack" an Apple event when it arrives from the outside world.) This handler would extract the event class and event ID attributes and would get the object specifier from the direct parameter. The handler would then call AEResolve and pass the object class (along with the event class and event ID) to your table lookup routine.

The only real problem occurs when the object specifier isn't contained in the direct parameter. The solution here is to install handlers for any events that don't contain their object specifiers in their direct parameters, and have these handlers call AEResolve and then jump directly into your table lookup routine.

The implementation of such a table-based dispatcher is left to you.

HOW DO I RESOLVE AN OBJECT SPECIFIER?

When an object-model Apple event is received, such as “Close Document 1,” the object specifier (Document 1) is usually contained in the direct parameter of the event. Before the event can be processed, the object specifier needs to be *resolved*. Resolving an object specifier involves locating the specified information in memory so that the Apple event can act on this information.

While it’s possible to parse an object specifier directly, object specifiers can be much more complicated than the simple examples shown here. The Apple event Object Support Library (OSL) helps you resolve an object specifier through a set of *object accessor routines*, which you write and then install. One type of accessor routine extracts one or more types of element from a given object, while other accessor routines extract a property from an object. When you ask the OSL to resolve an object specifier, it calls the appropriate accessor routines in the necessary order.

Figure 6 shows how the OSL resolves the object specifier “Word 5 of Window 1.” First, the accessor for the innermost specifier (Window 1) is called. This accessor returns a token, which is an Apple event descriptor (AEDesc) referring to some data in your application. The returned token and the next part of the object specifier to be processed are then passed to the appropriate accessor. This process is repeated until the object specifier has been fully resolved, and the final result is returned to your application.

HOW DO I IMPLEMENT AN OBJECT ACCESSOR?

Each accessor routine should accept one part of an object specifier and return a token. An accessor routine has the form

```
pascal OSErr MyAccessor (DescType desiredClass, const AEDesc *container,
   DescType containerClass, DescType keyForm, const AEDesc *keyData,
   AEDesc *value, long refCon);
```

and is passed the desiredClass, containerClass, keyForm, and keyData fields directly from the part of the object specifier being resolved. The container is either the token returned from the last accessor called or an AEDesc of type 'null' containing a null handle (if this is the first accessor in the series to be called).

All accessors have to perform essentially the same functions:

1. Check that the specified key form is valid.

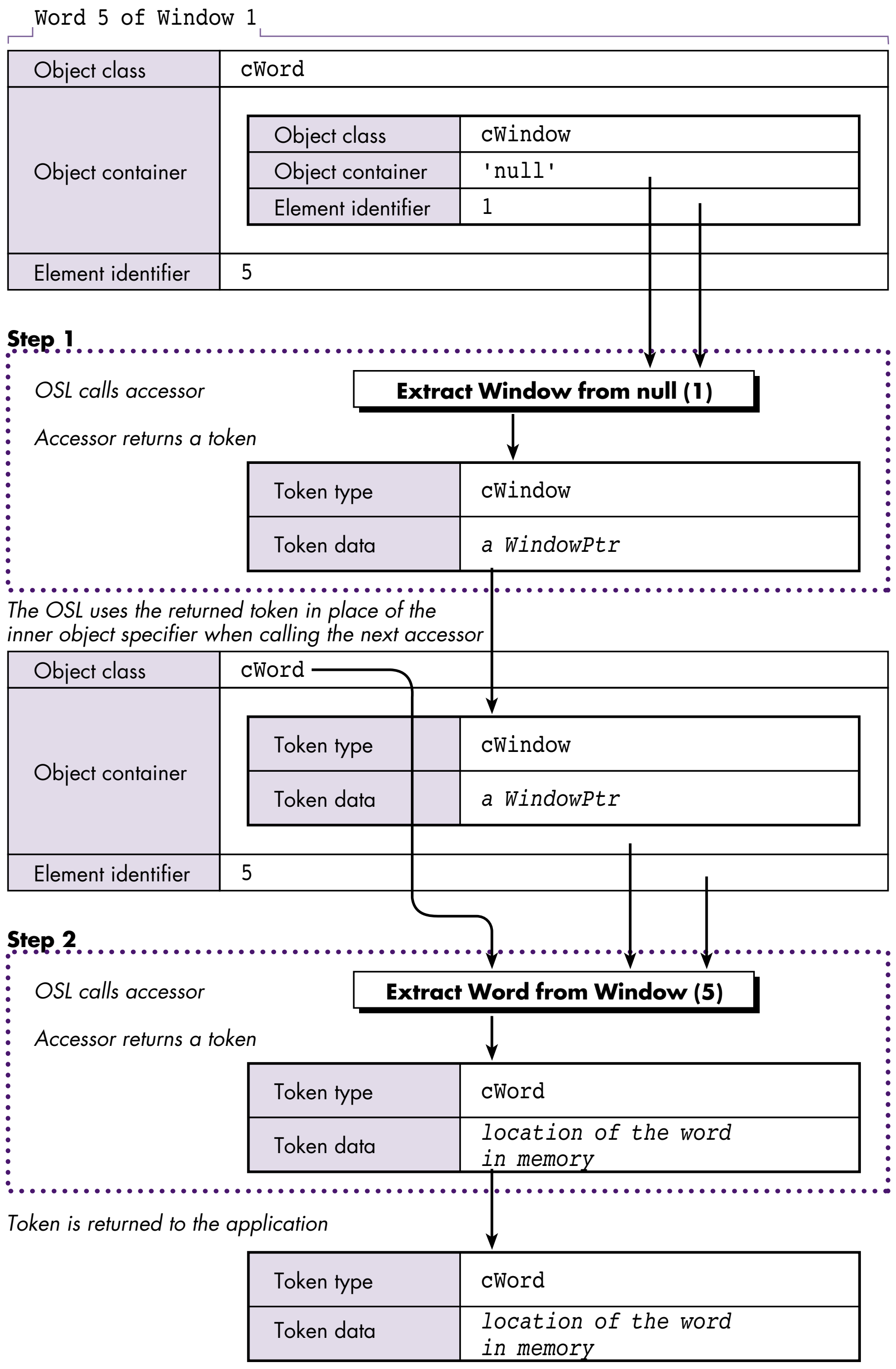


Figure 6
Resolving an Object Specifier

2. Locate the requested information.
3. Construct a return token.

The following code illustrates this process using a simple "extract a Window from a null container" accessor. (In most applications, this accessor extracts both windows and documents from the null container since most applications maintain a one-to-one correspondence between documents and windows.)

```
pascal OSErr WindowFromNull (DescType desiredClass,
   const AEDesc *containerToken, DescType containerClass,
   DescType keyForm, const AEDesc *keyData, AEDesc *theToken,
   long theRefcon)
{
   WindowPtr    wp;
   long         count;

   /* 1. Make sure we can handle this request. We only handle */
   /* object specifiers of the form "Window 1", "Window 2", etc. */
   if ((keyForm != formAbsolutePosition) return errAEBadKeyForm;

   /* 2. Extract the window number and find the window. */
   count = **(long**)(keyData->dataHandle);
   wp = FrontWindow();
   while (count > 1) {
      if (wp == 0L) return errAENoSuchObject;
      wp = (WindowPtr)((WindowPeek)wp)->nextWindow;
      --count;  /* Count down by 1. */
   };

   /* 3. Create the token. */
   /* The token is an AE descriptor of type 'cwin' (window). */
   /* The AEDesc contains a handle to a WindowPtr. */
   return AECreateDesc(desiredClass, (Ptr)&wp, sizeof(wp), theToken);
} /* WindowFromNull */
```

While the above code contains many of the features of an object accessor, it's far from complete. For example, it doesn't handle formName, which is one of the more common key forms. It also assumes that the value for a formAbsolutePosition parameter will be a positive integer. In fact, the value could be a negative number (with -1 signifying the last element of the container, -2 signifying the next to the last element, and so on), or one of the special constants representing the first, last, middle, any, or every element of the container.

To make the formAbsolutePosition code complete, you need to add a routine that looks at the key data for one of the special values and converts the key data into a positive integer or returns a flag indicating that every element should be returned. Such a routine would look something like this:

```
OSErr GetWindowIndex (const AEDesc *keyData, long *index, Boolean *getAll)
{
   long     numWindows;
   long     rawIndex;

   /* There are three flavors of formAbsolutePosition key: */
   /* typeLongInteger/typeIndexDescriptor, typeRelativeDescriptor, */
   /* and typeAbsoluteOrdinal. */

   /* 1. Initialize some values. */
   *getAll = false; *index = 1;
   numWindows = CountUserWindows(); /* A private routine */

   /* 2. Get the number out of the key. If it's not an absolute */
   /* value, convert it to one. */
   rawIndex = **(long**)(keyData->dataHandle);
   switch (keyData->descriptorType) {

      case typeLongInteger:
         if (rawIndex < 0)
         /* A negative value means "the Nth object from the end," */
         /* i.e., -1 = the last object. */
            rawIndex = numWindows + rawIndex + 1;
         /* A positive value is an absolute value, so do nothing. */
      break;

      case typeAbsoluteOrdinal:
         /* kAEFirst, etc. are special 4-byte constants. */
         if (rawIndex == kAEFirst)       rawIndex = 1;
         else if (rawIndex == kAELast)   rawIndex = numWindows;
         else if (rawIndex == kAEMiddle) rawIndex = numWindows / 2;
         else if (rawIndex == kAEAll)    *getAll = true;
         else if (rawIndex == kAEAny) {  /* Select a random window. */
            if (numWindows <= 1)  /* 0 or 1 */
               rawIndex = numWindows;
            else
            /* Get a random number between 1 and numWindows. */
               rawIndex = 1 + ((unsigned long)Random() % numWindows);
         }
         else return errAEBadKeyForm;
      break;
   }
   return noErr;
} /* GetWindowIndex */
```

To install an accessor, use the AEInstallObjectAccessor routine:

```
pascal OSErr AEInstallObjectAccessor (DescType desiredClass,
      DescType containerType, accessorProcPtr theAccessor,
      long accessorRefcon, Boolean isSysHandler)
```

In the “extract a Window from a null container” example, the call to the AEInstallObjectAccessor routine would look like this:

```
err = AEInstallObjectAccessor(cWindow, 'null',
      (accessorProcPtr)WindowFromNull, 0, false);
```

You can also install accessor routines to get one of the properties of an object (use the special constant 'prop' in specifying the desired type), or you can supply a wild card for either the container or the desired type. Most developers install one accessor routine for each of the element types supported by a particular object, and one accessor routine to handle all of the properties of that object.

WHAT SHOULD I PUT INTO A TOKEN?

As noted earlier, accessors communicate with each other and with the application using application-specific *tokens*. Most Apple events that contain an object specifier end up resolving the object specifier into a token and then manipulating the data represented by that token. Since the format of each object class is different, you’ll typically write Read Token Data and Write Token Data routines for each object class that your application supports. (You might also choose to write Create Token Data (Create Element) and Delete Token Data routines if more than one Apple event in a given object needs to create or delete information.) What you put into these token-handling routines depends completely on the contents of your tokens.

Each token is stored in an Apple event descriptor—a data structure containing a 4-byte type code and a handle to some data, where the contents of the handle are completely up to you. While this raises the question of what should go into the handle, many developers decide to invent a different token data type for each object class or set of related object classes.

In this approach, a window token would contain a WindowPtr, a text token would contain a handle to some text, and so on. Since tokens are used for both elements and properties, each token might also contain a 4-byte property code.

Here’s how the tokens might look for the object classes defined in Figure 1:

```
struct DocumentTokenBody {
   WindowPtr    theWindow;
   Boolean      useProperty;
   DescType     propertyCode;
};

struct RectTokenBody {
   Rect         *theRect;          /* Use a pointer so we can read */
                                   /* and write the rectangle. */
   long         elementNumber;     /* See token-handling examples */
                                   /* below. */
   Boolean      useProperty;
   DescType     propertyCode;
   WindowPtr    parentWindow;      /* The window that holds this */
                                   /* rectangle. */
};

struct WordTokenBody {
   Handle       theText;
   long         startingOffset;    /* How many bytes in does the text */
                                   /* start? */
   long         textLength;        /* How many bytes long? */
   Boolean      useProperty;
   DescType     propertyCode;
   TEHandle     parent;            /* The location from which we took */
                                   /* this text. */
};
```

These three sample tokens demonstrate several things you should keep in mind when designing your own tokens:

- Each token contains a reference to the data—not a copy of the data itself. This allows the same token to be used for both reading and writing the data.
- Each token contains a field for the property code. If the application received the object specifier “the Name of Document 1,” the returned token would contain a pointer to the document’s window and the Name property code—'pnam'. The token-handling routines have to include code to support property tokens.
- Since each token format is different, you’ll need to write the token-handling routines (Read/Write and, optionally, Insert/Delete) for each token type.

- The Word and Rectangle tokens contain references to the objects that contain them. This is important, since changing the text or the rectangle could affect the document containing the information and there's no way to get either a partially resolved object specifier or the intermediate products of the resolution. *Therefore, if you need to know the parent of a particular token, you must store a copy of that information in the child token yourself, since the OSL may dispose of the original parent token.* (You may need to supply a custom DisposeToken callback if your tokens contain handles or pointers to other data.)

The guidelines given above cover the contents of the token's handle, but they don't say anything about the descriptorType field. When you return a token from an accessor routine, you must put the proper type code into the descriptorType field of the AEDesc. This is required because the OSL uses the returned token type from one step of the resolution process to guide the next step. Having the accessor routines control the resolution process actually insulates outside Apple event sources from having to know about your specific implementation details.

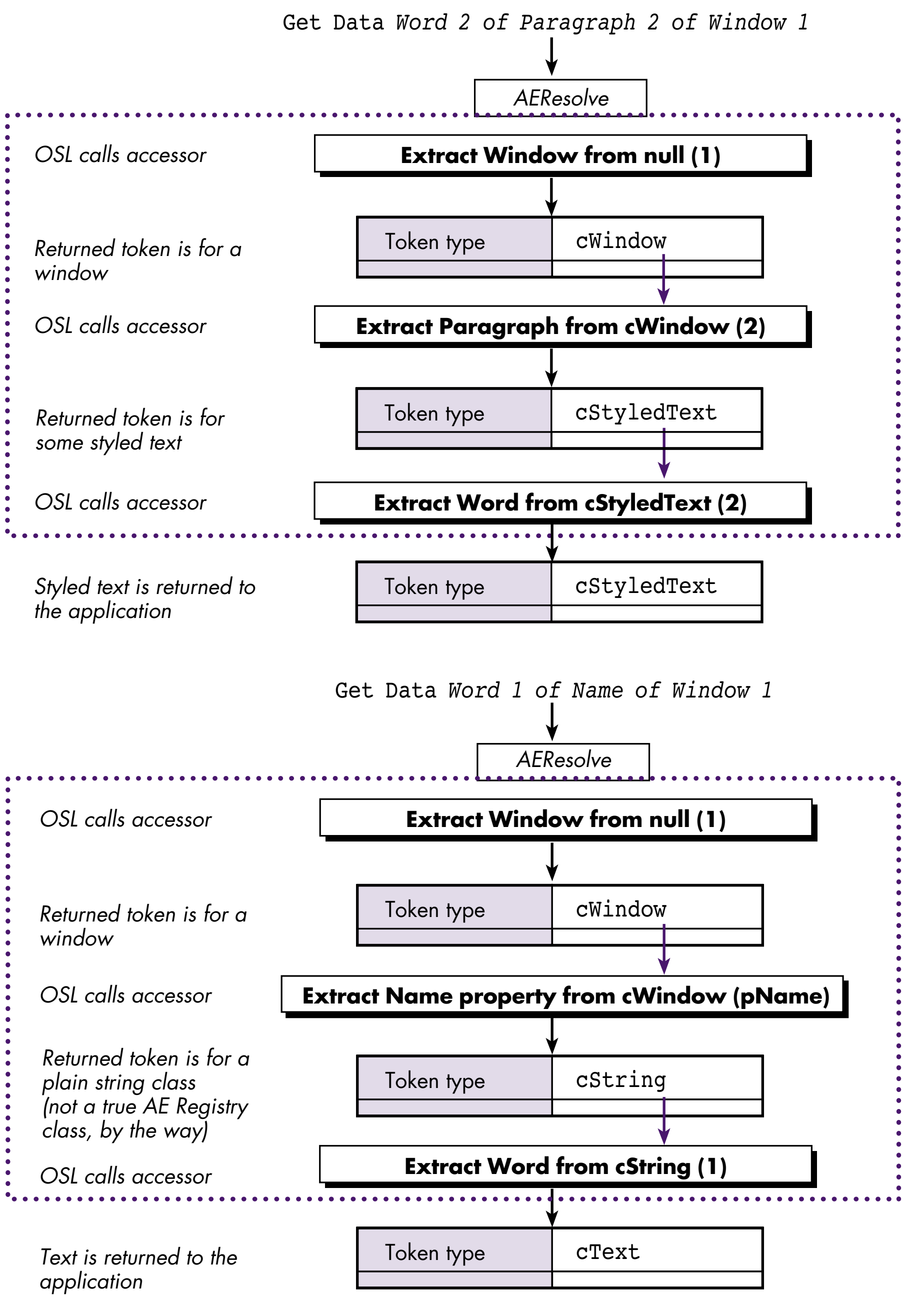
Throughout the article, we've assumed that the token type in the token is the same as the external data type specified by the object specifier. However, your code can put anything in the token type field as long as you write the matching object accessors for those token types.

For example, let's say that you've written a word-processing program, and another application sends the request "Get Data *Word 2 of Paragraph 2 of Window 1*" where the italicized part is an object specifier. The returned type would probably be some styled text. However, if the requester had sent "Get Data *Word 1 of the Name of Window 1*," your application would have to access a completely different form of text (a simple Str255) and might return some nonstyled text.

Internally, the data type that represents text within a document can be different from the data type representing a simple string. Instead of forcing the user to use two different terms for the same thing (documentWord and plainTextWord, perhaps), the application can make this determination at run time. Figure 7 shows how an application might resolve the two examples given above.

WHAT ARE TOKEN-HANDLING ROUTINES?

Token-handling routines are optional routines (in other words, routines not explicitly required by the object model or OSL) that perform common editing operations on the data referred to by a token. Generally, when you have a token, you want to read, write, insert, or delete the data the token refers to. Here are Read Token Data and Write Token Data handlers for the cRectangle object class:



Get Data Word 2 of Paragraph 2 of Window 1
AEResolve
OSL calls accessor
Extract Window from null (1)
Returned token is for a window
Token type
cWindow
OSL calls accessor
Extract Paragraph from cWindow (2)
Returned token is for some styled text
Token type
cStyledText
OSL calls accessor
Extract Word from cStyledText (2)
Styled text is returned to the application
Token type
cStyledText
Get Data Word 1 of Name of Window 1
AEResolve
OSL calls accessor
Extract Window from null (1)
Returned token is for a window
Token type
cWindow
OSL calls accessor
Extract Name property from cWindow (pName)
Returned token is for a plain string class (not a true AE Registry class, by the way)
Token type
cString
OSL calls accessor
Extract Word from cString (1)
Text is returned to the application
Token type
cText

Figure 7
Controlling Object Specifier Resolution with Returned Tokens

```
struct RectTokenBody {
   Rect        *theRect;      /* Use a pointer so we can read */
                              /* and write the rectangle. */
   long        elementNumber;
   Boolean     useProperty;
   DescType    propertyCode;
   WindowPtr   parentWindow;  /* The window that holds this rectangle. */
};

typedef struct RectTokenBody RectTokenBody;
typedef RectTokenBody *RectTokenPtr, **RectTokenHandle;

OSErr ReadRectToken (const AEDesc *theToken, AEDesc *result)
{
/* This routine gets called by the Get Data Apple event handler (or any */
/* other handler that needs to read some data and possibly return it to */
/* the user). If the useProperty flag is true, we return the requested */
/* property, otherwise we return the default representation for this */
/* class (we'll use the cQDRect primitive type for this). */

   RectTokenPtr    tokenPtr;
   DescType        descCode;
   OSErr           err;

   HLock(theToken->dataHandle);
   tokenPtr = (RectTokenPtr)*theToken->dataHandle;
   if (tokenPtr->useProperty) {
      switch (tokenPtr->propertyCode) {

         case pClass:
            /* Tell the world that this is a rectangle. */
            descCode = cRectangle;
            err = AECreateDesc(typeType, (Ptr)&descCode,
                sizeof(descCode), result);
         break;

         case pBounds:
            /* Return the bounds of this rectangle, as a QuickDraw */
            /* rectangle. */
            err = AECreateDesc(typeQDRectangle,
               (Ptr)&tokenPtr->theRect, sizeof(Rect), result);
         break;

         /* More property codes go here... */
```

The cRectangle class used in this code is simplified. Remember that if you're implementing the real cRectangle class from the *Apple Event Registry*, you'll need to support many more properties and a more complex default representation.•

```
         default:
            err = errAENoSuchObject;
      }
   }
   else {
      /* Return the default representation. In this simple example, */
      /* it's a QuickDraw rectangle. */
      err = AECreateDesc(typeQDRectangle, (Ptr)&tokenPtr->theRect,
         sizeof(Rect), result);
   }
   return err;
}

OSErr WriteRectToken (const AEDesc *theToken, const AEDesc *theData)
{
/* This routine gets called by the Set Data Apple event handler (or */
/* any other handler that needs to change a property or some value */
/* of the object). If the useProperty flag is true, we check to see */
/* if the property is writable and modify it, otherwise we change */
/* the contents of this object. */

   RectTokenPtr    tokenPtr;
   AEDesc          thisRectDesc;
   OSErr           err;

   HLock(theToken->dataHandle);
   tokenPtr = (RectTokenPtr)*theToken->dataHandle;
   if (tokenPtr->useProperty) {
      switch (tokenPtr->propertyCode) {

         case pClass: /* This is a read-only property. */
            err = errAEWriteDenied;
         break;

         case pBounds: /* Set the bounds of this rectangle. */
            /* Make sure we have a QuickDraw Rectangle. */
            err = AECoerceDesc(theToken, typeQDRectangle,
               &thisRectDesc);
            if (err != noErr) return err;
            /* Copy the data into our rectangle. */
            BlockMove(*thisRectDesc.dataHandle, &tokenPtr->theRect,
               sizeof(Rect));
            AEDisposeDesc(&thisRectDesc);
         break;
         /* More property codes go here... */

         default:
            err = errAENoSuchObject;
      }
   }
   else {
      /* Change the default representation (the bounds of this */
      /* rectangle). */
      err = AECoerceDesc(theToken, typeQDRectangle, &thisRectDesc);
      if (err != noErr) return err;
      /* Copy the data into our rectangle. */
      BlockMove(*thisRectDesc.dataHandle, &tokenPtr->theRect,
         sizeof(Rect));
      AEDisposeDesc(&thisRectDesc);
   }
   return err;
}
```

The contents of the Create Element and Delete Token Data routines are completely application-specific and are not illustrated here. Typically, the Create Element routine takes a token for the element's container and an index position within that container, and returns an object specifier describing the new element. (This object specifier may be returned as the result of a Create Element Apple event, or may be resolved so that you can insert some data into the newly created element.)

COMBINING OBJECTS AND EVENTS

Once you've created the object event dispatcher code, the object accessor routines, the token formats, and the token handlers, your last task is to write the actual event-handling routines. (These are different from the routines that you install into the Apple Event Manager's dispatch table; event-handling routines do the work for a specific event as handled by a specific object class.) While the exact content of these routines is application dependent, they do have some features in common:

- Routines that need to return something to the outside world can use a Read Token Data handler to convert an internal token into an externally usable form, and can use the other token manipulation routines as needed.
- Each routine should accept both the event and the reply record as parameters. The results from an event are typically placed into the direct parameter of the reply record. When your event has finished execution, the Apple Event Manager will send the reply back to the client application.

MOVING ON

Writing an object model application isn't difficult; once you've implemented an object or two (including the accessors and tokens) and a couple of events, you should have a good understanding of the issues. I hope that this article has given you a good idea of where and how to begin adding the object model to your application. If you still need help there are several options: reading the related documentation (see the box below); looking at the sample code on the *Developer CD Series* disc (the samples Quill and AEObject-Edition Sample in the Apple Events and Scripting Development Kit and the sample code provided with this article); talking with other programmers; training through Apple's Developer University; and using the on-line support available through AppleLink, CompuServe, and other means.

Good luck! We all look forward to seeing the exciting things that can be done when applications can work both cooperatively and under the control of scripting environments.

RELATED READING

- *Inside Macintosh* Volume VI (Addison-Wesley, 1991) provides fundamental information about Apple events. Chapter 1 gives an overview of interapplication communication and explains the relationship of the Apple Event Manager to other parts of System 7. Chapter 6 provides a complete description of Apple events, explains how to send and receive Apple events, and includes reference information for all Apple Event Manager routines.
- The Apple Event Manager chapter of the new, improved *Inside Macintosh* (preliminary draft) on the *Developer CD Series* disc provides information about Apple event objects and object classes.
- *Apple Event Registry: Standard Suites,* on the *Developer CD Series* disc, describes standard Apple events, Apple event data types, and Apple event object classes. A printed version of the *Apple Event Registry* is available from APDA (#R0130LL/A).

THANKS TO OUR TECHNICAL REVIEWERS
Kevin Calhoun, Donn Denman, C. K. Haun, Eric House, Bennet Marks •

PRINT HINTS

TOP 10 PRINTING CRIMES

PETE ("LUKE") ALEXANDER

In this issue, we're going to take a slightly different tack. Instead of dealing with one printing hint, we're going to give you ten. We'll take a look at the "Top 10 Printing Crimes" that I've seen during my three and a half year adventure in Apple's Developer Technical Support Group. I'll start by listing these crimes, and then I'll discuss the solution to each one.

Here's the list:

10. Loading PDEFs directly from within your application.
9. Poor memory management at print time.
8. Assuming the grafPort returned by PrOpenDoc is black and white.
7. Not saving and restoring the grafPort or resource file in your application's pIdle procedure.
6. Not using PrGeneral when you should to determine and set the resolution of the current device.
5. Not reading Macintosh Technical Note #91, "PicComments—The Real Deal," before you start using PicComments in your application.
4. Opening the Printing Manager when your application starts up.
3. Mixing high-level and low-level printing calls.
2. Accessing private and unused fields in the print record.
1. Adding printing to your application two weeks before going final.

All of these crimes are very easy to avoid. Let's take a look at the solution to each one.

SOLUTIONS TO THE PRINTING CRIMES

10. Loading PDEFs directly from within your application.

A PDEF is a printer driver's CODE resource definition. Each printer driver contains multiple PDEFs, which implement the various functions of the driver (such as displaying the Print dialogs, opening the connection with the printer, and supporting PrGeneral). A few applications load and call these PDEFs directly, probably because they feel this will improve printing performance. Instead, this approach will usually cause serious compatibility problems and headaches for printer driver developers. Also, it's very difficult for printing utilities (for example, utilities that count the number of pages printed) to patch into printing if an application isn't using the printing trap (PrGlue). Finally, this approach could cause some serious compatibility problems for users when a new printer and its associated driver software are released.

Solution: The main function of the Printing Manager is to load the printer driver PDEFs in a device- and driver-independent manner. Using the Printing Manager to load the PDEFs is the simplest and most compatible method.

9. Poor memory management at print time.

Poor memory management at print time will cause some interesting problems with various printer drivers. Usually, some object in your document won't print or you'll receive a blank page. The problem is that each printer driver available on the Macintosh requires a different amount of memory; some require very little memory, while others require a lot. For example, the

PETE ("LUKE") ALEXANDER Inquiring minds want to know: Does Luke have a life beyond these weird Print Hints he dishes out occasionally? The answer is a resounding YES! This happy hacker likes to keep his head in the clouds—literally. The proud owner of an ASW-20 sailplane, Luke's other passion (besides working at Apple) is soaring 10,000 feet above ground, while observing eagles, mountain goats, and wild horses in exotic outposts of California and Nevada. Luke has the "funnest time" when he's gliding like a bird, suspended in time with the air rushing past him. For him, it's pure, unparalleled excitement and enjoyment.•

LaserWriter SC is one of the piggier drivers. What's an application to do?

Solution: Since each printer driver uses a different amount of memory, there's not a magic amount of memory that will always ensure the success of a print job. The best solution to this problem is to unload *all* unnecessary code and data segments at print time. The more memory available, the better. In addition to ensuring that printing will work OK, more memory can improve printing performance significantly, which your users will thank you for.

8. *Assuming the grafPort returned by PrOpenDoc is black and white.*

Yes, PrOpenDoc can return a color grafPort, if the printer driver you're using supports color. Unfortunately, not all printer drivers are capable of returning a color grafPort. This feature caused compatibility headaches for us when we released LaserWriter driver version 6.0, which was the first printer driver from Apple that could return a color grafPort. Many applications assumed that the grafPort it returned was black and white, and this assumption caused quite a few applications to die when printing to LaserWriter driver 6.0. This assumption can also have some very ugly results if your user is printing to a color printer and you're only sending black-and-white data.

Solution: A good rule of thumb when printing: never assume anything. Usually there are methods available to enable your application to determine the environment it's in. Printing isn't any different; in fact, this is probably even more important for printing. You should check the grafPort returned by PrOpenDoc to see whether it's color or black and white: if the high bit in the rowBytes of the grafPort is set, you have a color grafPort.

7. *Not saving and restoring the grafPort or resource file in your application's pIdle procedure.*

Many applications install a pIdle procedure at print time. This procedure allows the application to present the print job status to the user. This is a very good idea—but you must be a little defensive to keep a printer driver happy.

Solution: When your application enters its pIdle procedure, you should save the current grafPort and resource file (that is, the printer driver's). When you exit your pIdle procedure, you should restore the grafPort and resource file back to the original. This is extremely important, because the printer driver assumes that the current grafPort and resource file are always its own. If they're not, when you exit your pIdle procedure you won't be drawing into the correct grafPort, and when the printer driver makes the next Resource Manager call, it will have the wrong resource file. Technical Note #294, "Me And My pIdle Proc (or how to let users know what's going on during print time . . .)," describes the details of creating and using a pIdle procedure within your application.

6. *Not using PrGeneral when you should to determine and set the resolution of the current device.*

The PrGeneral trap allows a developer to determine the supported resolutions of the current printer, and also to set the resolution, determine the page orientation selected by the user, and force draft printing. Many developers who want resolution information don't use the power of this trap, but instead use a device-dependent method, which is *bad*. PrGeneral allows you to determine the resolution in a device-independent manner, so that you'll be able to print to *all* printers connected to the Macintosh without knowing about the printer you're talking to. There are now over 130 printer drivers available on the Macintosh. It would be a real shame if your application couldn't maximize its output to a device just because you made a bad assumption.

Solution: This is a case where you can be *completely* device independent in your print code without sacrificing anything. You can obtain outstanding results if you use the PrGeneral trap correctly. Any time you're interested in the available resolutions for the current printer, you should use the GetRsl opcode supplied by PrGeneral. For details about getting and setting the resolution, see the "Meet PrGeneral" article in Issue 3 of *develop.* If you don't have the article handy, it's available on the *Developer CD Series* disc. Accompanying the article on the CD is an application named PrGeneralPlay that contains complete sample code for PrGeneral. You should probably also take a look at *Inside Macintosh* Volume V, pages 410-416.

5. Not reading Macintosh Technical Note #91, "PicComments—The Real Deal," before you start using PicComments in your application.

Many developers have tried to use PicComments in their applications before understanding their function, with very mixed results. If you don't follow the recommendations in Technical Note #91, you'll definitely receive some undesirable results—especially if you don't match all "open" calls with a "close" call.

Solution: Read Technical Note #91 *before* you start using any PicComments in your application. This Note has been rewritten with new pictures, sample code, and descriptions to help developers properly use PicComments in their printing code. It will help you avoid many of the pitfalls and misuses of PicComments. It's also helpful to look at pictures generated by other applications, to see what they're doing.

4. Opening the Printing Manager when your application starts up.

In the early Macintosh days, it was recommended that you always call PrOpen at application startup. This hasn't been the recommendation for a long time. Why? When you open the Printing Manager, it loads some of the printer driver's resources into memory. This means that less memory is available for your application. However, the real problem is that other applications or DAs cannot print until you close the Printing Manager, since the Printing Manager is *not* reentrant. Unfortunately, there isn't a reliable method for determining whether the Printing Manager is open, nor is there a method for closing it if it's already open. This isn't much of a problem any more because the majority of applications today no longer call PrOpen at startup.

Solution: Do *not* open the Printing Manager until you're ready to print or perform some other printing-related task (for example, initializing a print record when your application starts up). You should close the Printing Manager when the print job is complete or when you've accomplished the task at hand. You should never allow a user to switch your application out with the Printing Manager open (that is, never call WaitNextEvent between PrOpen and PrClose).

3. Mixing high-level and low-level printing calls.

This is one of the classic printing problems. You should *never* mix the high-level and low-level printing calls. This approach will usually cause instant death at print time, because the high-level and low-level calls do very similar things. One of the common mistakes is calling PrDrvrClose after calling PrClose. Printer drivers are not designed to use both interfaces simultaneously.

Solution: In general, all applications should be using the high-level printing calls. Please follow the advice in Technical Note #161, "A Printing Loop That Cares . . . ," which describes the use of the high-level calls. Always match each "open" printing call with its corresponding "close" call. Also, check the PrError function for a printing error before making the next printing call.

The only advantage gained by using the low-level calls would be when you're text streaming, which is easier with those calls. Technical Note #192, "Surprises in LaserWriter 5.0 and Newer," describes the use of the low-level interface.

As you might expect, there's a minor exception to this rule. If you've read the Printing Manager chapter of *Inside Macintosh* Volume II, you may have noticed that the PrDrvrVers function is defined in the "Low-Level Driver Access Routines" section (page 162). This function can also be used with the high-level interface (it's the *only* low-level call that can be called in the high-level interface). PrDrvrVers is very useful for determining the version of a printer driver, which will enable you to work around bugs that may exist in a specific version of a printer driver.

2. Accessing private and unused fields in the print record.

Many of the print record fields should not be accessed by an application because they're used by the printer driver as storage locations, which means the information in them *will* change during a print job.

Solution: You should *never* use any information from fields in the print record that have "PT" at the end of the field name. All of them have corresponding "public" fields in the print record for application use. For example, you should use the information stored in rPage, and not rPagePT. Printer drivers store some of their private information in the fields with "PT" at the end of the field name. During printing, the values in these fields will change. Furthermore, different printer drivers use these fields differently, so accessing one of them might work on one driver but not another. Use the public fields!

1. Adding printing to your application two weeks before going final.

This one might be a slight exaggeration, but it's definitely in the ballpark. Believe it or not, I've talked to quite a few developers who have left printing as the last feature they add to their application (or maybe next to last, just before Undo). This can cause some serious problems in your development schedule.

Solution: There *are* a few pitfalls in printing, but they can be avoided if you start early in the design phase of your application. My advice to avoid this problem is to start printing from your application as soon as possible. When you have an early prototype running, send some output to the printer. Usually you can tell very early if you'll have any problems.

One more thing: I created this list in order from the least printing crime to the worst. Actually, if you commit any of the printing crimes mentioned, you'll probably receive some undesirable results with various printers. I would suggest testing your application on at least one PostScript® printer and a QuickDraw printer.

Finally, if we take a look out onto the documentation horizon, we can see something new peeking through. What is it, you ask? It's the new and improved *Inside Macintosh* chapter on printing. Yes, after years of waiting, it's finally coming. I believe you'll find the new printing chapter useful and informative. It will unlock additional information about printing on the Macintosh.

REFERENCES

- *Inside Macintosh* Volume V, Chapter 22, "The Printing Manager," pages 410–416 (Addison-Wesley, 1988).
- *Inside Macintosh* Volume II, Chapter 5, "The Printing Manager," page 162 (Addison-Wesley, 1985).
- "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager," Pete "Luke" Alexander, *develop* Issue 3, July 1990.
- "Me And My pIdle Proc (or how to let users know what's going on during print time . . .)," Macintosh Technical Note #294.
- "Surprises in LaserWriter 5.0 and Newer," Macintosh Technical Note #192.
- "A Printing Loop That Cares . . . ," Macintosh Technical Note #161.
- "PicComments—The Real Deal," Macintosh Technical Note #91.

Thanks to Dave Hersey and Scott "Zz" Zimmerman for reviewing this column.•

POSTSCRIPT ENHANCEMENTS FOR THE LASERWRITER FONT UTILITY

For System 7, the LaserWriter Font Utility was given the ability to handle drop-in enhancements, called UTILs. These hybrid Macintosh-and-PostScript utilities are provided with a rich parameter block and many useful callbacks. They offer a straightforward method for putting useful tidbits of PostScript code to work—with a real user interface.

BRYAN K. ("BEAKER") RESSLER

The LaserWriter Font Utility is an obscure system software application that isn't even installed by the System 7 Installer. Its main mission is to facilitate the downloading of TrueType, PostScript Type 1, and PostScript Type 3 fonts to PostScript (and PostScript-compatible) printers and printer hard disks. With System 7, however, the innocuous LaserWriter Font Utility has been endowed with an extensible Utilities menu and the ability to handle drop-in enhancements, called *UTILs*. UTILs can be used for a variety of interesting applications. For example:

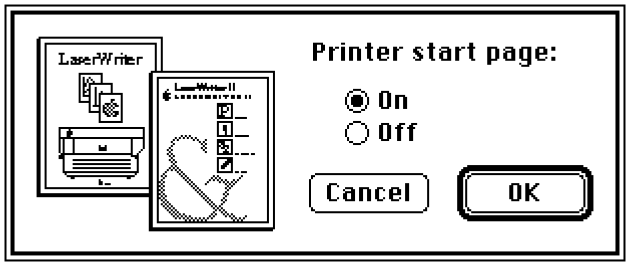
- downloading a PostScript language file or restarting a PostScript printer with special-purpose PostScript utilities
- setting the resolution or printing an alignment page on a particular model of typesetter with device-specific applications
- putting little snippets of PostScript code to work

INTRODUCING UTILS

UTILs are resources that are stored in the LaserWriter Font Utility's application resource file. When the Font Utility is run, UTILs are collected into the Utilities menu, listed by their resource name.

UTILs are generally modal and very task-specific. For example, one of the UTILs that are distributed as part of the Font Utility, Start Page Options, allows users of PostScript printers to decide whether or not the printer produces a "start page" when turned on. When the user chooses this UTIL from the menu, the dialog box shown in Figure 1 appears.

BRYAN K. RESSLER, a.k.a. "Beaker," is a bloodstained "binary vivisectionists" who regularly cuts into live code just to see what happens. He terrorized the University of California, Irvine for four years, and just to get rid of him, they gave him a B.S. in computer science. Beaker did the System 7 revision of the LaserWriter Font Utility. Now, when the medication wears off and he's allowed out of his cell, he writes sound and MIDI applications, composes marginal music, and sharpens his "binary scalpel."•



Printer start page:

[✓ On] Off

Cancel OK

Figure 1
Example of a UTIL Dialog Box

A UTIL performs its task via PostScript code embedded in the UTIL or in its owned resources. Therefore, UTILs are provided with routines that ease two-way communication with the PostScript printer.

UTILs may own resources and allocate a block of private “global” memory. UTILs get printer configuration information from the Font Utility and may also query the printer directly for configuration information. This allows for device-specific UTILs.

Since most UTILs are expected to be implemented similarly, many application facilities are provided to UTILs so that common code, like the bold outline for the default button in Figure 1, is not duplicated in every UTIL. As a result, most UTILs are very small (under 2K).

UTIL RESOURCES

UTILs are stored as resources of type 'UTIL'. Their IDs start at 128. However, the range 128 through 149 is reserved by Apple, so you should use an ID of 150 or higher for the UTILs you write. The UTIL's resource name defines the text of the menu item that's appended to the Utilities menu.

The UTIL resource format is shown in Figure 2.

The first two bytes of the resource specify the version of the UTIL resource format, which is currently $0001. Next comes resSpace, the first ID in the UTIL's resource space. A UTIL's *resource space* is the range of IDs that the UTIL may use for its owned resources. The UTIL has 100 consecutive IDs, starting with resSpace. In general, to calculate a given UTIL's resource space ID, use the formula

resSpace = 16000 + (UTILID - 128) * 100

where UTILID is the resource ID of the UTIL resource itself. For example, if your UTIL resource were numbered 158, your UTIL's resource space would be calculated as follows:

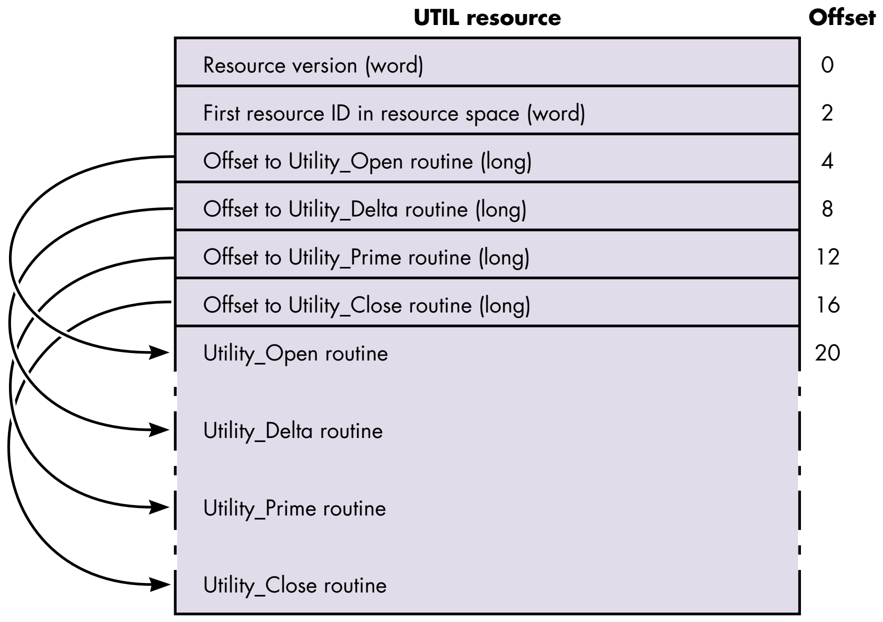

UTIL resource
Offset
Resource version (word)
0
First resource ID in resource space (word)
2
Offset to Utility_Open routine (long)
4
Offset to Utility_Delta routine (long)
8
Offset to Utility_Prime routine (long)
12
Offset to Utility_Close routine (long)
16
Utility_Open routine
20
Utility_Delta routine
Utility_Prime routine
Utility_Close routine

Figure 2
UTIL Resource Format

resSpace = 16000 + (158 -128) * 100 = 19000

In your UTIL's code, it's important to use *relative* resource IDs, in case your UTIL is renumbered at installation time. There are several examples of relative resource IDs in the code we'll be examining.

Following the version and resSpace, the UTIL resource contains offsets to the four UTIL entry points (described in the next section). These offsets are from the beginning of the UTIL resource. The beginning and order of entry points is flexible (see "Variations on UTIL Entry Points" for details).

UTIL ENTRY POINTS

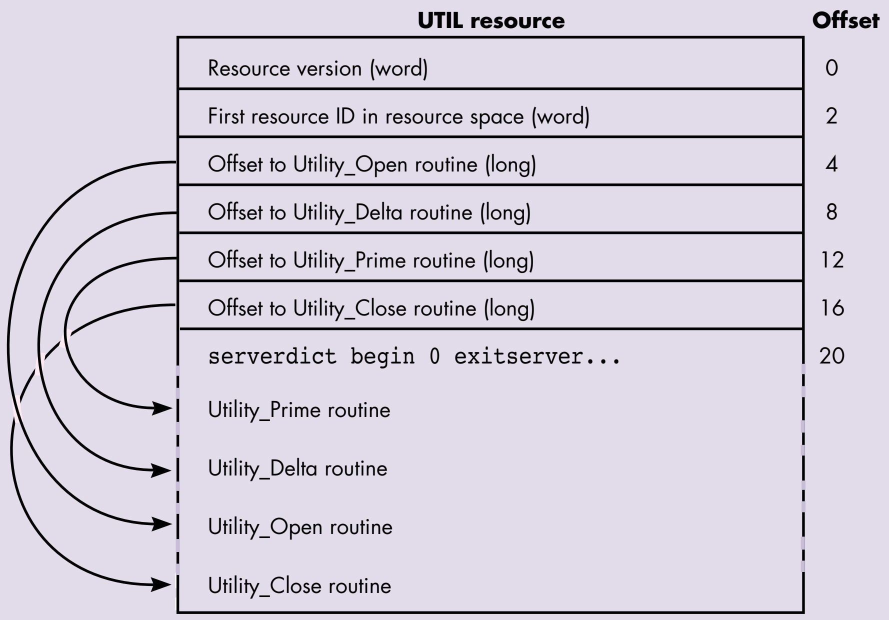
Let's take a quick look at the four UTIL entry points. Later, in the section "The Script: NamerUTIL.c Code," we go into more detail.

Utility_Open. This routine is called by the Font Utility at startup, right after the UTIL is loaded into memory. Utility_Open initializes the UTIL and allocates any memory it requires. Utility_Open needs to return a Boolean result which, if true, tells the Font Utility to install the UTIL in the Utilities menu. A false result from Utility_Open, which might occur if there were insufficient memory, tells the Font Utility not to install this UTIL.

VARIATIONS ON UTIL ENTRY POINTS

There's no requirement that the Utility_Open routine start at offset 20, nor is there any specification of the order in which the routines are stored within the resource. This allows noncode data to be stored in the UTIL resource along with the code. Figure 3 shows an example of PostScript code stored inside a UTIL resource, and the four routines rearranged as desired.

While this method of storage works well for embedded PostScript code, remember that other textual data (that's *not* PostScript code) should be stored in resources to facilitate localization.



UTIL resource
Offset
Resource version (word)
0
First resource ID in resource space (word)
2
Offset to Utility_Open routine (long)
4
Offset to Utility_Delta routine (long)
8
Offset to Utility_Prime routine (long)
12
Offset to Utility_Close routine (long)
16
serverdict begin 0 exitserver...
20
Utility_Prime routine
Utility_Delta routine
Utility_Open routine
Utility_Close routine

Figure 3
Arranging UTIL Routines

Utility_Delta. This routine is called by the Font Utility once after Utility_Open and subsequently any time the user selects a different printer with the Chooser. Utility_Delta is the means by which a UTIL informs the Font Utility whether the UTIL's menu item should be dimmed or not (usually based on the characteristics of the currently selected printer). The Font Utility provides a host of useful printer configuration information, but should your UTIL require different or more specific information, it may download PostScript code at this point and parse the response from the printer. Utility_Delta needs to return true if the UTIL's menu item is to be available, or false if it's to be dimmed.

Utility_Prime. This routine is called by the Font Utility to carry out the basic function of the UTIL. It's called when the user chooses your UTIL's menu item from the Utilities menu. The Start Page Options UTIL described earlier downloads some PostScript code to determine the current state of the **dostartpage** flag in the printer, puts up a dialog box, and then downloads more PostScript code to set the flag to the new setting. The Utility_Prime routine needs to return a long word composed of (possibly multiple) return codes. These codes tell the Font Utility of any special behavior it should take upon return, such as refreshing its font lists.

Utility_Close. This routine is called by the Font Utility at quit time. Normally, at this point your UTIL releases any memory it has allocated.

GOODIES IN THE LWFUPARMBLK STRUCTURE

When the LaserWriter Font Utility is starting up, it allocates one LWFUParmBlk for each installed UTIL. That means that each UTIL's parameter block is unique, and the same block is always passed to it. Each of the entry points described above takes as a parameter a pointer to an LWFUParmBlk structure. This structure is discussed in detail in the section "The Script: NamerUTIL.c Code," but here are the high points.

General information. This part of the structure includes the version of the LWFUParmBlk structure, the base resource ID for this UTIL's resource space, and a storage field into which the UTIL may place a handle to some global storage space.

Driver information. You're provided with an FSSpec pointing to the currently selected printer driver. Tempting as this might be, you should use this only to retrieve the driver's version, allowing your UTIL to put the driver version in a dialog box.

Printer information. This includes the name of the current printer and a host of printer configuration information. Also included is a handle to the Font Utility's own standard Macintosh print record, which allows you to print via the Printing Manager if you wish.

Callback information. This is a rich set of callbacks into the Font Utility. There's also a pointer to the Font Utility's QuickDraw globals and pointers to two large I/O buffers you can use for printer communication. The callback routines can be grouped into three major categories:

- 3 PAP routines that assist in printer communication
- 18 dialog utility routines, many of which you would have probably had to include anyway

- 4 Pascal-string utility routines that aid in the construction of PostScript language strings

ON STAGE: NAMERUTIL

Now that we've got an overview of how UTILs fit into the LaserWriter Font Utility, let's take a closer look at a specific example, NamerUTIL, provided on the *Developer CD Series* disc. NamerUTIL, which appears in the Utilities menu as Rename Printer, allows the user to rename the currently selected printer.

Figure 4 shows part of the Utilities menu, including the Rename Printer UTIL.

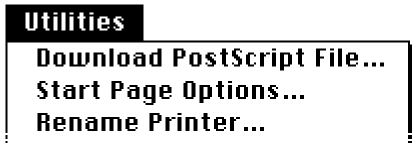


Figure 4
UTILs in the Utilities Menu

Rename Printer's Utility_Prime routine presents the user with the dialog box shown at the top of Figure 5. The dialog is smart enough to limit the length of the new name to 30 characters and disallow various illegal characters. If the user clicks Rename, the UTIL transmits a PostScript program to rename the printer as specified. The UTIL then puts up one of the two alerts shown in Figure 5, depending on whether the printer was successfully renamed or not.

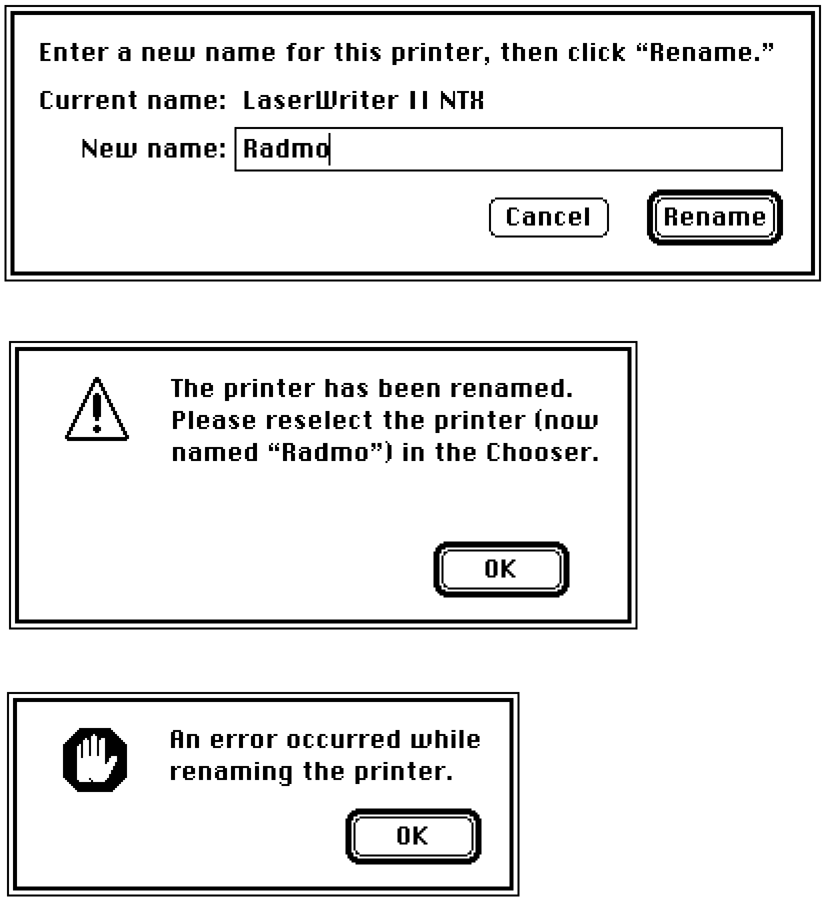
That's the basic user interface design for the Rename Printer UTIL. So how do we make it work? First, let's take a look at how you rename a printer in PostScript.

RENAMING A PRINTER IN POSTSCRIPT

The PostScript code to rename a printer is trivial.

```
serverdict begin 0 exitserver
   statusdict begin
      (NewPrinterName) setprintername
   end
```

The first line enters **serverdict**, a dictionary of operators for controlling the PostScript server, then exits the server loop with the **exitserver** operator. The 0 is the system administrator password, which is almost universally 0. (In fact, the LaserWriter driver won't work correctly if the password has been changed, so it's OK for us to assume it's 0 and hard-code the password.) The net effect of exiting the server loop is to allow us to change persistent parameters, like the printer name.


Enter a new name for this printer, then click "Rename."

Current name: LaserWriter II NTX

New name: Radmo

The printer has been renamed. Please reselect the printer (now named "Radmo") in the Chooser.

An error occurred while renaming the printer.

Figure 5
Rename Printer Dialog Box and Alerts

The next line enters **statusdict**, a dictionary containing machine- and configuration-dependent operators. The printer is renamed with the **setprintername** PostScript operator. The UTIL replaces *NewPrinterName* with the new name provided by the user.

NAMERUTIL AND ITS SOURCE FILES

NamerUTIL, the Rename Printer UTIL, is made up of five source files. Two of them, UTIL.h and UTILHead.a, are provided on the CD as general interface files and are the same for all UTILs. Figure 6 gives an overview of the source files and their relationships. We'll look at the source files whose names begin with "Namer" and at the makefile.

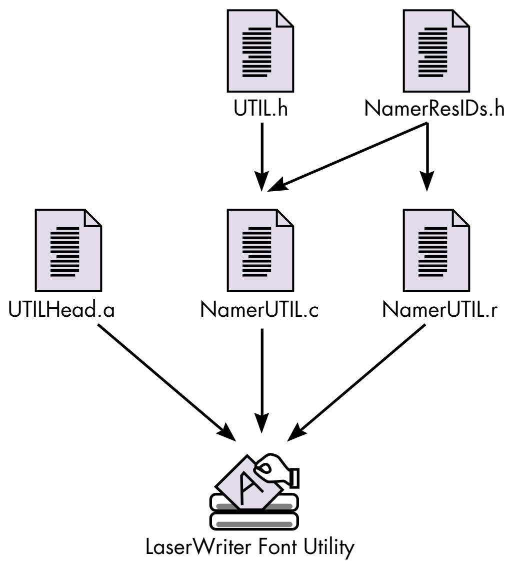


Figure 6
Source File Relationships

NamerUTIL.c. This C language source file is the bulk of the code for NamerUTIL. It contains the four entry points, and three other routines that carry out NamerUTIL's job. Thanks to all the Font Utility's callbacks, it's fairly straightforward.

NamerUTIL.r. This Rez input file contains descriptions of all the NamerUTIL's owned resources. Since the owned resources' IDs depend on the ID of the UTIL resource itself, the resource IDs are specified *relatively*, that is, by an offset from a variable named ResSpace, which is defined in the makefile.

NamerResIDs.h. This file contains #define statements for all the resource IDs. It's included by both NamerUTIL.c and NamerUTIL.r.

MakeFile. MakeFile ties all the pieces together. For development, it's easiest to use Rez to place NamerUTIL's resources into a copy of the Font Utility. Use Rez's **-a** option to append the resources to those already present in the LaserWriter Font Utility application file. Then link NamerUTIL's UTIL resource directly into the Font Utility as well. Finally, by launching the Font Utility, you can test your UTIL. That's the approach of the makefile we'll be looking at. The CD contains an alternate makefile that generates a standalone file and an application called UTILInstall for installing UTILs into the Font Utility.

Let's dive right into NamerUTIL.c.

THE SCRIPT: NAMERUTIL.C CODE

Here's the first part of NamerUTIL.c in MPW 3.2 C.

```
/* NamerUTIL.c - a UTIL that allows the LaserWriter Font Utility to rename
   PostScript printers. */

/* --- Includes ----------------------------------------------------*/
#include <Types.h>              /* Macintosh includes */
#include <Memory.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <Dialogs.h>
#include <Printing.h>
#include <ToolUtils.h>
#include <Errors.h>

#include "UTIL.h"               /* Standard UTIL constants and */
                                /* structures */
#include "NamerResIDs.h"        /* "Relative" resource IDs */

/* --- Defines ------------------------------------------------ */
#define kMinVersion      1      /* Minimum version we'll run */

#define kPSErrStr        1      /* Strings in 'STR#' kNamerStrs */
#define kExitVerStr      2
#define kRenameStartStr  3
#define kRenameEndStr    4

#define kNDDummy         0      /* Item #s in Namer dialog box */
#define kNDRename        1
#define kNDCancel        2
#define kNDNewName       6
#define kNDBoldOutline   7

#define kReturnKey       0x0d   /* Key and char code constants */
#define kEnterKey        0x03
#define kBackspaceKey    0x08
#define kAtChar          '@'
#define kColonChar       ':'
#define kLowASCII        0x7f

#define kMaxNameLength   30     /* Maximum printer name length */
#define kNameBufLen      40     /* Size of printer name buffers */
#define kCompStrLen      80     /* Size of parse string buffers */
```

```
/* --- Prototypes -------------------------------------------- */
pascal Boolean          Utility_Open(LWFUParmBlk *pb);
pascal Boolean          Utility_Delta(LWFUParmBlk *pb);
pascal unsigned long    Utility_Prime(LWFUParmBlk *pb);
pascal void             Utility_Close(LWFUParmBlk *pb);
pascal short            ExitBufferRtn(short length, LWFUParmBlk *pb);
pascal Boolean          NamerFilter(DialogPtr TheDialog,
                           EventRecord *TheEvent, short *ItemHit);
short                   RenamePrinter(LWFUParmBlk *pb);
```

At the beginning are the includes. Besides the usual Macintosh Toolbox and OS includes, we include UTIL.h, a header file used by all UTILs, and NamerResIDs.h. The latter, as mentioned earlier, contains constants that we'll add to the resSpace field of the LWFUParmBlk to form valid resource IDs. Sharing this file with NamerUTIL.r makes maintenance easier.

The #defines are for indices into NamerUTIL's owned STR# resource, item numbers for the dialog box, various keyboard and character constants, and buffer sizes.

Below the constant definitions are prototypes for NamerUTIL's routines. You can see the standard four entry points, Utility_Open, Utility_Delta, Utility_Prime, and Utility_Close, plus three other routines, which will be called from the Prime routine.

THE SUPPORTING CAST: UTILITY_OPEN, UTILITY_DELTA, AND UTILITY_CLOSE

The Utility_Open routine is exceedingly simple. It checks that the parameter block passed in pb is equal to or newer than the minimum version. In the future, the LWFUParmBlk structure may be extended—for instance, to add new fields or callbacks. Since subsequent versions of the UTIL parameter block are defined to be *extensions*, a UTIL can function with any version of the LWFUParmBlk greater than or equal to the version provided when the UTIL was written. The current version is $0001. If the parameter block is new enough, Utility_Open returns true and we're on our way.

```
pascal Boolean Utility_Open(LWFUParmBlk *pb)
{
   return(pb->version >= kMinVersion);
}
```

Some UTILs might want to allocate memory in their Utility_Open routine and store a handle to the storage in pb->uStorage. If the allocation failed, Utility_Open could return false, indicating that it should not be installed into the Utilities menu.

If you thought NamerUTIL's Utility_Open was trivial, take a look at Utility_Delta. This entry point is called for each UTIL, once at startup and subsequently every time the user chooses a different printer with the Chooser. Since NamerUTIL can rename any PostScript printer, its Utility_Delta always returns true, indicating to the Font Utility that the Rename Printer menu item should always be available (as opposed to dimmed). Utility_Delta is provided to facilitate device-specific UTILs that might, for instance, want to send out PostScript code to determine the printer's specific make and dim the menu item if the UTIL isn't applicable to the chosen printer.

```
pascal Boolean Utility_Delta(LWFUParmBlk *pb)
{
#pragma unused(pb)

   return(true);
}
```

The #pragma keeps the C compiler from barking at us about not using the pb parameter.

Let's save the best for last and dispatch with Utility_Close, so we can get on with the meat of the UTIL, Utility_Prime. Since Utility_Open didn't allocate any storage, the Utility_Close routine can simply return without doing anything. Even though the routine may do nothing, *it must be included*, and *will* be called at quit time.

```
pascal void Utility_Close(LWFUParmBlk *pb)
{
#pragma unused(pb)
}
```

THE STAR OF THE SHOW: UTILITY_PRIME

All the work NamerUTIL performs is handled by the Utility_Prime routine, shown here:

```
pascal unsigned long Utility_Prime(LWFUParmBlk *pb)
{
   if (RenamePrinter(pb))
      return(urCheckPrinter | urCheckFeatures | urEraseLists);
   else return(urNoAction);
}
```

Utility_Prime calls RenamePrinter to do most of its work. RenamePrinter returns a Boolean result that indicates, if true, that the printer was successfully renamed or, if false, that there was an error or the user canceled the process. If RenamePrinter returns true, Utility_Prime returns a conglomerate return code that indicates to the Font Utility that it should recheck its connection with the chosen printer, recheck the printer's features, and forget any font lists it might have for the printer. If the user canceled or there was an error, Utility_Prime tells the Font Utility to take no special action.

Let's look at the beginning of RenamePrinter:

```
short RenamePrinter(LWFUParmBlk *pb)
{
   DialogPtr          nameDlg;               /* The Rename dialog */
   short              itemHit;               /* From ModalDialog */
   char               psBuffer[150];         /* Buffer for PostScript */
   char               newName[kNameBufLen];  /* New printer name */
   char               blankStr[1];           /* Handy empty string */
   short              status;                /* ExitBufferRtn's status */
   Point              where;                 /* For positioning dialog */
   LWFUCallBackInfo   *cb;                   /* Pointer to callbacks */
   Boolean            doneFlag = false;      /* Flags to dismiss */
   Boolean            renameFlag = false;    /* Flags to rename */
   Boolean            retVal = false;        /* True/false on success */

   *blankStr = 0;
   cb = pb->callBacks;
```

Here we declare the local variables. The most important locals are nameDlg, the dialog pointer; status, which tests the success of the rename operation; and cb, a cached pointer to the callback array. We set blankStr to be an empty string (for use in ParamText calls and such) and initialize cb's value.

Creating the set. Now, we set up our Namer dialog box.

```
nameDlg = GetNewDialog(pb->resSpace + kNamerDlg, nil, (WindowPtr)-1);
cb->CenterDialog(nameDlg, &where);
MoveWindow(nameDlg, where.h, where.v, true);

ParamText(pb->printerInfo->currentPrinterName, blankStr, blankStr,
   blankStr);

cb->SetPText(nameDlg, kNDNewName, blankStr);
cb->UserItem(nameDlg, kNDBoldOutline, cb->BoldOutlineItem);

(LWFUParmBlk *)((DialogPeek)nameDlg)->window.refCon = pb;
ShowWindow((WindowPtr)nameDlg);
```

RenamePrinter gets the Namer dialog box from the resource file with a call to GetNewDialog. Notice the way the resource ID is specified as pb->resSpace + *constant*. This relative resource ID convention makes the code flexible in case the UTIL was renumbered for some reason. You'll see this convention throughout NamerUTIL.

The design for the Font Utility tried to encompass the most common types of operations UTIL writers would be performing and provided those routines as callbacks. One example is centering a dialog box. The CenterDialog callback returns a point—the top left coordinates for a MoveWindow call.

RenamePrinter then installs the current printer name as parameter ^0, so the user can see the current printer name in the dialog box. RenamePrinter presets the new printer name to blank and installs a userItem to put the bold outline around the dialog box's default button. Again, notice the handy callbacks.

The dialog filter, NamerFilter, needs access to the LWFUParmBlk. To allow this, RenamePrinter installs a pointer to the block in the dialog window's refCon field.

The dialog template resource for the Namer dialog box specifies a hidden window, so all the fuss we've just gone through hasn't been disturbing the poor user. Once finished, RenamePrinter puts the dialog box on the screen with ShowWindow.

Behind the scenes. Next comes the ModalDialog loop.

```
do {
   ModalDialog(NamerFilter, &itemHit);
   switch(itemHit) {
      case kNDRename:    /* Rename */
         renameFlag = doneFlag = true;
         break;
      case kNDCancel:    /* Cancel */
         doneFlag = true;
         break;
   }
} while (!doneFlag);

HideWindow((WindowPtr)nameDlg);
```

This is your average modal dialog hit-loop. RenamePrinter waits for the user to dismiss the dialog box, setting the renameFlag appropriately, then hiding the dialog window (but not disposing of it yet). RenamePrinter filters the dialog events with NamerFilter, shown here:

```
pascal Boolean NamerFilter(DialogPtr TheDialog, EventRecord *TheEvent,
   short *ItemHit)
{
   unsigned char   theKey;                  /* From the event record */
   short           retVal = false;          /* The return value */
   char            newName[kNameBufLen];    /* The new name */
   LWFUParmBlk     *pb;                     /* The parameter block */
   /* Retrieve a pointer to the parameter block. */
   pb = (LWFUParmBlk *)((WindowPeek)TheDialog)->refCon;

   /* Trap keyDown and autoKey events. */
   if (TheEvent->what == keyDown || TheEvent->what == autoKey) {

      /* Grab the ASCII character code from the event record. */
      theKey = TheEvent->message & charCodeMask;

      if (theKey == kReturnKey || theKey == kEnterKey) {
         /* Return or Enter? Hit the default button. */
         retVal = true;
         *ItemHit = kNDRename;
      } else if (theKey == kAtChar || theKey == kColonChar ||
            theKey > kLowASCII) {
         /* "@", ":", or a high ASCII char? Beep and tell */
         /* ModalDialog to ignore the event. */
         SysBeep(5);
         retVal = true;
         *ItemHit = kNDDummy;
      } else if (theKey != kBackspaceKey) {
         /* Key other than Backspace? Check length to decide. */
         pb->callBacks->GetPText(TheDialog, kNDNewName, newName);
         if (*newName >= kMaxNameLength) {
            SysBeep(5);
            retVal = true;
            *ItemHit = kNDDummy;
         } else retVal = false;
      }
   } else retVal = false;

   return(retVal);
}
```

This filter keeps users from entering an illegal printer name. Specifically, it disallows ampersand (@) and colon (:) characters (reserved for forming NBP network names) and high ASCII characters (because PostScript is technically 7-bit ASCII), and it limits the length of the name to kMaxNameLength (30). The filter can make use of callbacks, because we put a pointer to our LWFUParmBlk into the window's refCon field.

Performance time. If renameFlag is true, RenamePrinter performs the rename operation.

Here's the code:

```
if (renameFlag) {
   GetIndString(psBuffer, pb->resSpace + kNamerStrs, kRenameStartStr);
   cb->GetPText(nameDlg, kNDNewName, newName);
   cb->Pstrcat(psBuffer, newName);
   cb->GetAndAppend(psBuffer, pb->resSpace + kNamerStrs, kRenameEndStr);
```

In constructing an appropriate PostScript program to rename the printer, we get the first part of the PostScript code from our STR# resource, append the new name the user provided, and finally tack on the end of the PostScript code. The handy Pascal-string utility callbacks were included for just this situation. We end up with our little PostScript renamer program in psBuffer, as a Pascal string.

Now things start getting a little more interesting. The code that actually downloads psBuffer to the printer and checks the results is as follows:

```
   if ((status = cb->OpenPrinter()) == noErr) {
      status = cb->DoWrite(psBuffer + 1, (short)*psBuffer, sendEOF, pb,
         ExitBufferRtn);
      cb->ClosePrinter();
   }
```

RenamePrinter opens a connection to the printer with the callback OpenPrinter. If there's no error, RenamePrinter uses DoWrite to write psBuffer to the printer. The sendEOF argument tells DoWrite to send an end-of-file indication to the printer after writing the specified text to the printer. Notice that ExitBufferRtn, the "output parser," is included as a parameter to DoWrite.

Here's the code for ExitBufferRtn:

```
pascal short ExitBufferRtn(short length, LWFUParmBlk *pb)
{
   Handle   dataHandle;                 /* "Handlized" response */
   short    status;                     /* The return code */
   char     psErrorText[kCompStrLen];   /* Buffer for "fail" test */
   char     exitText[kCompStrLen];      /* Buffer for "success" test */

   GetIndString(psErrorText, pb->resSpace + kNamerStrs, kPSErrStr);
   GetIndString(exitText, pb->resSpace + kNamerStrs, kExitVerStr);

   PtrToHand(pb->callBacks->PAPReadBuffer, &dataHandle, length);

   if (Munger(dataHandle, 0, psErrorText + 1, *psErrorText,
         nil, 0) >= 0)
      status = printerError;
   else if (Munger(dataHandle, 0, exitText + 1, *exitText,
         nil, 0) >= 0)
      status = noErr;
   else status = printerError;

   DisposHandle(dataHandle);
   return(status);
}
```

DoWrite constantly polls the printer for some response. If anything comes back from the printer (like an error, or some verification that the operation was completed), it's sent to ExitBufferRtn. The return value from ExitBufferRtn is passed back to DoWrite, and subsequently returned as DoWrite's return value. This allows ExitBufferRtn to essentially "post" an error. The only predefined error is printerError, which is defined in UTIL.h. You may define your own error codes as well.

In order to return an error code, ExitBufferRtn needs to look for some sign of success or failure from the printer. To determine failure, it looks for "%%[Error: ", which is the beginning of all error strings that are returned by PostScript printers. We don't care about parsing the rest, since *any* error is enough for us to return printerError.

To detect success, ExitBufferRtn searches for the string "%%[exitserver: permanent", which is the beginning of "%%[exitserver: permanent state may be changed]%%". This string tells us that the **exitserver** password was correct. We should always see this response from the printer. These two search strings are stored in the resource file. Observe the relative resource IDs.

You may have noticed that a characteristic of UTILs seems to be avoiding work by using all those handy callbacks. But now we have to search for a string. Yuck! Well, let's cheat. We'll take the momentary memory hit and use PtrToHand to create a handle to a copy of the response from the printer. Then we'll search with every hacker's dream routine, Munger. After ExitBufferRtn sets status, it disposes of the temporary handle and returns.

Back in RenamePrinter, we set our local variable status from the DoWrite call. Since status was passed through from ExitBufferRtn, RenamePrinter can tell whether or not the rename operation was successful. Either way, it's really important to remember to close the connection. The rule is, *if the OpenPrinter call succeeded, do a ClosePrinter call, even if the code in between failed.*

Posting the reviews. Now that the ugly transmission-reception stuff is finished, let's tell the user what's going on.

```
   cb->UseCursor(0);
   if (status == noErr) {
      retVal = true;
      ParamText(newName, blankStr, blankStr, blankStr);
      cb->PositionAlert(pb->resSpace + kVerifyAlrt);
      CautionAlert(pb->resSpace + kVerifyAlrt, nil);
   } else {
      retVal = false;
      cb->PositionAlert(pb->resSpace + kFailAlrt);
      StopAlert(pb->resSpace + kFailAlrt, nil);
   }
}
```

Since the user needs to go out and choose the newly renamed printer with the Chooser, RenamePrinter puts up an alert if the rename operation is successful, half as a reminder to choose the printer, and half as a reminder of the new printer name. (See "Where'd That Printer Go?") If status, the local variable, was nonzero, RenamePrinter puts up a general error alert. In any case, RenamePrinter sets the return value appropriately, since Utility_Prime needs to know whether the printer is renamed, so that it can return the appropriate flags to the Font Utility.

Finally, RenamePrinter disposes of the (currently hidden) Namer dialog and returns.

```
   DisposDialog(nameDlg);
   return(retVal);
}
```

That's basically it! Now let's glance at the resource file, NamerUTIL.r.

NAMERUTIL'S RESOURCES: NAMERUTIL.R

The only remarkable feature of NamerUTIL's resource file, NamerUTIL.r, is its use of relative resource IDs. For example, here's NamerUTIL's main dialog box DLOG resource:

```
resource 'DLOG' (ResSpace + kNamerDlg, "Namer", purgeable) {
   {40, 20, 162, 412},
   dBoxProc, invisible, noGoAway, 0x0,
   ResSpace + kNamerDlg,
   ""
};
```

Note that both the resource ID of the DLOG resource itself and the ID of the associated DITL resource are specified relatively. NamerUTIL.r also contains the following resource definition:

WHERE'D THAT PRINTER GO?

We rename the printer behind the Font Utility's back (and the Chooser's, for that matter). The name and zone information for the currently chosen printer is stored in some string resources in the System file. After we rename the printer, the strings in the System file still contain the old printer name.

The return value urCheckPrinter causes the Font Utility to attempt to open a PAP connection with the "currently chosen printer." Since the name of the currently chosen printer has not been updated, the PAPOpen call fails, and the user gets the alert shown in Figure 7. This is an unfortunate side effect of renaming the printer.

The solution is to tell the Chooser that we have chosen a printer in the same zone with our new name, but there's no officially sanctioned way to do this. Future versions of system software will remedy this situation.

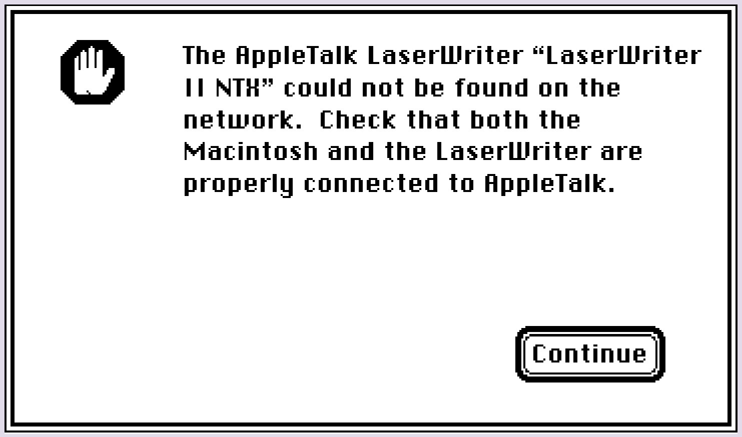

The AppleTalk LaserWriter "LaserWriter II NTX" could not be found on the network. Check that both the Macintosh and the LaserWriter are properly connected to AppleTalk.

Figure 7
A NamerUTIL Side Effect

```
resource 'uvrs' (ResSpace + kVersion) {
   0x01, 0x00, final, 0x00,
   verUS,
   "1.0",
   "1.0, © 1991 Apple Computer, Inc."
};
```

The 'uvrs' resource is encouraged but not required. A 'uvrs' resource is structured exactly like a 'vers' resource. It stands for **U**TIL **vers**ion, and should have the same ID as the UTIL resource. The 'uvrs' resource provides a handy way to attach your copyright information to the UTIL.

BUILDING NAMERUTIL: MAKEFILE

Here's NamerUTIL's makefile:

```
# makefile - make rules for NamerUTIL

# Set up vars that describe UTIL's ID and resource space base
ID UTILID = 150
ResSpace = 18200
UTILName = "Rename Printer…"

COptions = -b -mbg full -sym off -r
AOptions = -d ResSpace={ResSpace}
LOptions = -sym off
ROptions = -a -d ResSpace={ResSpace}
Objects = UTILHead.a.o NamerUTIL.c.o "{Libraries}Interface.o"

LWFU ƒ LWFU.bak {Objects} NamerResources.rsrc MakeFile
   Duplicate -y LWFU.bak LWFU
   Rez {ROptions} -o LWFU NamerUTIL.r
   Link {LOptions} ∂
      -rt UTIL={UTILID} ∂
      -sn Main={UTILName} ∂
      -o LWFU {Objects}
   Beep 1c,5 1e,5 1g,5 2c,5 2c,5,0 1g,5 2c,10
```

This makefile assumes you have a copy of the LaserWriter Font Utility named LWFU.bak in your target directory. It makes a copy, called LWFU, and Rezzes and links your UTIL directly into LWFU. Naturally, you can't distribute a modified version of the LaserWriter Font Utility (see "Distribution of UTILs"), but for development, this is a really handy way to test your UTILs. The Beep command to play "CHARGE!" at the end is optional. I put it there to remind me that I'd rather be at the ball game.

BUT WAIT! THERE'S MORE!

Our little NamerUTIL is a useful example of what you might do with a LaserWriter Font Utility UTIL. It does not, however, use every callback and exercise every option available to it. The CD contains a UTIL specification that lists prototypes for all the callbacks and more detailed descriptions of the fields of the LWFUParmBlk. There's also a version of the NamerUTIL sources that's set up for use with the UTILInstall application, for those who are hoping to write the first modal-word-processor-spreadsheet-communications-package UTIL.

DISTRIBUTION OF UTILS

Since you can't distribute modified Apple system software, how can you distribute UTILs? The answer is to distribute them as UTIL files along with the UTILInstall application. UTILInstall (included with the source code on the *Developer CD Series* disc) is specifically designed for installing UTILs into the LaserWriter Font Utility. It requires that you generate a file, containing the UTIL and all its owned resources, that has a creator of 'UtIn' and a type of 'UTIL'. Also, you must supply a resource of type 'USPC' (**U**TIL **spec**ification) that tells the UTILInstall program what resources you own. This is the Rez format of the USPC resource:

```
type 'USPC' {
   integer = $$Countof(ResourceList);
   array ResourceList {
      unsigned longint;  /* Resource type */
      integer;           /* Resource ID */
   };
};
```

You provide a USPC resource with the same ID as your UTIL. The USPC resource lists the resource type and ID for every resource your UTIL owns. This tells the UTILInstall program exactly what resources to move, plus it facilitates the renumbering of your owned resources should there be an ID conflict.

For instance, suppose you've built your UTIL with UTIL ID 150 and ResSpace 18200. If the user attempts to install your UTIL into a copy of the Font Utility that already contains a UTIL with the ID 150, UTILInstall renumbers your UTIL and all its owned resources as specified in the USPC list.

Given a little thought, you might wonder what happens to resources that refer to other resources by ID, like DLOG and ALRT resources. Unfortunately, UTILInstall doesn't provide a comprehensive renumbering facility. It does, however, have special cases for renumbering DLOG and ALRT resources. Other resource types that refer to other resources by ID should be avoided if you plan to distribute your UTIL with UTILInstall.

For a complete example of the use of USPC resources, see the alternate NamerUTIL sources and the UTILInstall source code on the CD.

SO GO FOR IT!

A great aspect of UTILs is their hybrid nature—a unique combination of Macintosh code and PostScript code. So where you would have had to analyze a text file that came back from your PostScript code, you can write a UTIL that actually does something useful with the output. UTILs are also a great chance for hardware manufacturers to make those device-specific test and calibration pages accessible to common users.

So go for it! If you've got a collection of really cool PostScript hacks sitting around, here's your chance to give them a shiny faceplate and loose them on the unsuspecting world!

THANKS TO OUR TECHNICAL REVIEWERS
Richard Hu, Dave Johnson, Scott "Zz" Zimmerman•

GRAPHICAL TRUFFLES

MULTIPLE SCREENS REVEALED

FORREST TANAKA AND BILL GUSCHWAN

One very neat feature of the Macintosh is that you can connect more than one screen to the computer and use them as if they were one big screen. Better still, applications take advantage of multiple screens automatically. But the screens that are attached to your system can have different sizes, depths, and color tables, and you might want to optimize your application for each screen, or you might want to find the best screen to display something on. Both these things are easy to do, but not necessarily in the ways that you might think at first. In this column, we'll uncover a few important truths about QuickDraw's handling of multiple screens, and we'll talk about a few ways to deal with multiple screens if you want to go beyond what QuickDraw gives you for free.

It's important to understand that if you're just drawing items to a window and want to stay completely above the specifics of different screens, don't do anything special—just draw to your window as if there were one screen. QuickDraw was designed to make multiple screens look like one, so you should take advantage of this valuable abstraction if you can. Note too that machines with original QuickDraw can also have multiple screens, but we don't describe that here.

Truth #1: Windows don't change their depth or color table when they're moved to different screens.

One of the most common misconceptions about multiple screens is that a window's pixMap holds the size, depth, and color table of the screen that the window is on. That seems logical enough at first glance, especially considering that each screen has its own pixMap. But it's not true, because a window can cross more than one screen. Instead, the pixMap of a window always holds the depth, color table, and bounds rectangle of the main screen (the one with the menu bar) even if the window is nowhere near the main screen. The pixMap of a window is, in essence, a copy of the main screen's pixMap, except for one detail: the bounds rectangle of a window's pixMap is in the local coordinates of the window while the bounds rectangle of the main screen's pixMap is in global coordinates. In fact, any screen's pixMap has a bounds rectangle that's in global coordinates, indicating that screen's position relative to the main screen.

To find the sizes, depths, or color tables of the screens your window is on, you should use the list of GDevices that the system maintains (usually called the *device list*), which gives you the pixMap of each screen. We'll describe a method of using the device list later.

Truth #2: There are exactly two coordinate systems.

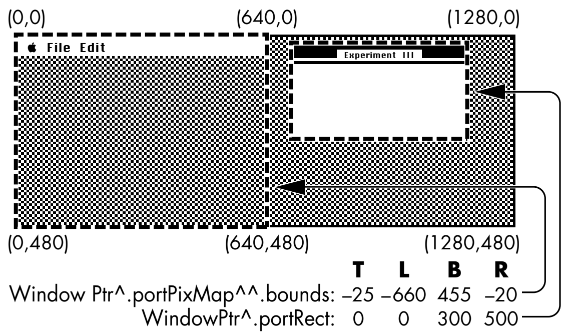
With multiple screens, it's easy to get confused by what looks like many coordinate systems, but there are only two: the local coordinate system of the current port and the global coordinate system. QuickDraw has no concept of a coordinate system for each screen. Global screen coordinates are always relative to the main screen—the global coordinate (0,0) is always at the extreme upper left corner of the menu bar.

All coordinates in a graphics port are local coordinates, including the bounds rectangle of the port's pixMap. This bounds rectangle has two purposes. First, it defines the area of a pixel image that QuickDraw can draw into. Second, the top left point of the bounds rectangle is the horizontal and vertical distance from the origin of the local coordinate system to the origin of the global coordinate system. Specifically, if you subtract the coordinate of the top left corner of the bounds rectangle from all the other coordinates in a port, you convert those coordinates into the equivalent global coordinates.

FORREST TANAKA Just before fastening that buckle on his bike helmet and snapping into those pedals, Forrest whispered, "Howdy, my name is Forrest; I don't drink, and I hate nicknames and terms of endearment. But I firmly believe that real life is more exciting and fantastic than the best fiction, except for Antoine de Saint-Exupéry's *The Little Prince*."•

BILL ("ANGUS") GUSCHWAN Stopping between moguls after some maney fakey shredding on his snowboard, Bill borrowed a few clock hands to say "Hi, my name is Angus; I like tacos, '71 Cabernet, and my favorite color is magenta." His favorite philosopher, Ludwig Wittgenstein, would be proud of his brevity.•

An example of the relationship between the portRect of a window and the bounds rectangle of its pixMap is shown in the following figure. The two screens in the example are next to each other and are both 640 pixels across and 480 pixels down, with the main screen on the left, and the window is contained entirely on the second screen. Global coordinates are marked around the corners of the screens and the portRect and bounds rectangle are marked with a dashed outline. Notice that the bounds rectangle circumscribes the main screen, and it's in the local coordinates of the window. If you subtract the components of the bounds rectangle's top left corner from the coordinates of the portRect, you get the rectangle [T:25 L:660 B:325 R:1160], which is the portRect in global coordinates.


(0,0)
(640,0)
(1280,0)
File Edit
Experiment III
(0,480)
(640,480)
(1280,480)
T L B R
Window Ptr^.portPixMap^^.bounds: –25 –660 455 –20
WindowPtr^.portRect: 0 0 300 500

Truth #3: QuickDraw switches to the GDevice of each screen your drawing crosses as it's drawn.

When you draw something to a window, QuickDraw searches the device list for every GDevice whose gdRect intersects your drawing. For each intersecting GDevice, QuickDraw makes it the current GDevice and then draws the intersecting part of your drawing. Switching GDevices is important because the current GDevice provides the current color environment, which tells the system what color corresponds to each pixel value and vice versa. As QuickDraw draws across your screens, it keeps switching the current GDevice to the one for the screen it's actively drawing to.

Color environments are specific to each screen. Compare this with grafPorts and cGrafPorts, which provide the screen-independent drawing environment that tells the system things like the pattern, pen size, and color to use when drawing something. Each window gets its own drawing environment, but has to share the color environments with other windows.

Therefore, you should never switch GDevices to have QuickDraw draw to a specific screen—QuickDraw switches GDevices as appropriate. Whenever you have QuickDraw draw to any screen, the current GDevice should be the main screen's GDevice, which it is by default. The only time that you should switch GDevices explicitly is to switch between on-screen and off-screen drawing.

Truth #4: On- and off-screen drawing are different.

QuickDraw distinguishes between on-screen and off-screen drawing for a couple of reasons. Starting with 32-Bit QuickDraw 1.0, video memory can only be reached in 32-bit addressing mode. If QuickDraw detects that it's drawing to a screen, it switches to 32-bit addressing mode, writes to video memory, and then switches back to the native addressing mode. QuickDraw stays in the native addressing mode for the entire operation when it draws off-screen unless bit 2 of the pmVersion field of the destination pixMap is set or unless it draws into a GWorld that's cached on a QuickDraw accelerator board. In those two cases, QuickDraw switches to 32-bit addressing mode even though it's drawing off-screen.

Another important difference between on-screen and off-screen drawing is that on-screen drawing makes QuickDraw go through the additional work of using the gdRects of the screens to determine which GDevices you're drawing to. We described this in Truth #3. When QuickDraw draws off-screen, it just uses the current GDevice.

QuickDraw senses whether it's drawing on-screen or off-screen by comparing the baseAddr field of the current graphics port's pixMap against the baseAddr of the main screen's pixMap. If they're equal, QuickDraw assumes that it's drawing on-screen (not necessarily the main screen!). Otherwise, QuickDraw assumes that it's drawing off-screen.

The device list is documented in the section "The Graphics Device Record" in Chapter 21 of *Inside Macintosh* Volume VI.•

To avoid confusing QuickDraw regarding whether it's drawing on-screen or off-screen, make sure that you always draw to a window for any on-screen drawing. The pixMap of any window is a lot like the main screen's pixMap, as we described in Truth #1, so the baseAddr of a window's pixMap is always the same as the baseAddr of the main screen's pixMap.

TRUTH IN ACTION

There are several ways to use these truths so that your applications optimize their displays for the sizes, depths, and color tables of each of the screens that are attached to the systems your application runs on. What follows are a few ways to do this.

If your window is completely contained on one screen, you might want to optimize your window's image for the screen it appears on. Usually, this means finding out the depth and color table of the screen your window is on. The device list, introduced in Truth #1, is invaluable for getting this information. For each GDevice in the list (remember, each GDevice represents a screen), compare the rectangle of its gdRect field against the rectangle of your window. The gdRect is in global coordinates while your window's portRect is in local coordinates, so you'll have to convert one or the other before doing the comparison. Once you've found the GDevice whose gdRect encompasses your window, get the GDevice's pixMap from the gdPMap field. Within this pixMap, the pixelSize field tells you the depth of the screen, and the pmTable field gives you a handle to the screen's color table. The device list is a linked list; you can get the first GDevice in the list with GetDeviceList, and you can go to the next GDevice with GetNextDevice.

What if your window intersects more than one screen? A common way to deal with this is to compromise by choosing a screen based on some criterion. You might want to choose the deepest screen that your window crosses, or the screen that intersects most of your window. The program listing at the end of this column shows a routine called FindScreenGDevice that takes a rectangle in global coordinates and a criterion, and returns the GDevice of the screen that satisfies the criterion. From this GDevice, you can get the information you need from the pixMap in the gdPMap field. If you pass kDeepestScreen for the criterion, FindScreenGDevice returns the GDevice of the deepest screen that intersects the rectangle. If you instead pass kLargestAreaScreen, the GDevice of the screen that has the largest intersection area is returned. Normally, you'd convert your window's portRect to global coordinates with the LocalToGlobal QuickDraw routine, and pass the resulting rectangle to FindScreenGDevice.

If your window displays an off-screen image and GWorlds are available, you can use GWorlds to make an off-screen image with the best depth and color table for the screens your window is on. If you pass 0 as the pixel depth to NewGWorld or UpdateGWorld and pass a rectangle defining the part of your window that displays the off-screen image in global coordinates, NewGWorld and UpdateGWorld set up an off-screen graphics environment that has the same depth and color table as the deepest screen your rectangle intersects, even if the area of intersection is as small as one pixel.

In some cases, you might want to display an image specifically to one screen, maybe for a presentations application or a game. To choose a screen, use a routine like FindScreenGDevice. Once you've chosen a screen, set up a window that fills that entire screen. Then draw to the window normally. In other words, you should again pretend that there's only one screen available, except that you have a little bit of insider information about where to put a window on that screen to make your images look or act best.

System 7 introduced the DeviceLoop routine, which is the recommended method for drawing images that are optimized for every screen they cross. For example, the highlight color can be drawn in black on a 1-bit screen, but in magenta on a deeper screen. If your application is running on a pre-7.0 system, you can simulate DeviceLoop by using a routine like DeviceLoopSim, as we show below. But to maintain future compatibility, DeviceLoop should be used if it is available.

DeviceLoop is described in Chapter 21 of *Inside Macintosh* Volume VI.•

You don't have to do anything special to let your applications work with multiple screens; QuickDraw makes multiple screens look like one screen. Use this abstraction even if you want to take advantage of specific screens. Keep using QuickDraw at a high level, and multiple-screen compatibility comes for free.

```
void DeviceLoopSim(
   RgnHandle                    drawingRgn,     /* Region to draw to */
   DeviceLoopDrawingProcPtr     drawingProc,    /* Routine to call to draw */
   long                         userData,       /* User-definable data */
   DeviceLoopFlags              flags)          /* Options; not implemented */
{
   GDHandle      aGDevice;       /* GDevice of each screen */
   RgnHandle     screenRgn;      /* Intersection of screen area and drawingRgn */
   RgnHandle     savedClip;      /* Saves the current port's clipping region */
   Rect          screenRect;     /* Rectangle of screen in global coordinates */

   /* Save the current port's clipping region */
   savedClip = NewRgn();
   GetClip( savedClip );

   /* Loop through every GDevice in the device list */
   screenRgn = NewRgn();
   aGDevice = GetDeviceList();
   while (aGDevice != nil)
   {
       /* Find region of intersection between screen and drawingRgn */
       screenRect = (**aGDevice).gdRect;
       GlobalToLocal( &topLeft( screenRect ) );
       GlobalToLocal( &botRight( screenRect ) );
       RectRgn( screenRgn, &screenRect );
       SectRgn( screenRgn, drawingRgn, screenRgn );

       /* If there is an area of intersection, call drawing proc */
       if (!EmptyRgn( screenRgn ))
       {
          SetClip( screenRgn );
          (*drawingProc)( (**(**aGDevice).gdPMap).pixelSize, (**aGDevice).gdFlags,
             aGDevice, userData );
       }
       /* Go to the next GDevice in the device list */
       aGDevice = GetNextDevice( aGDevice );
   }
   SetClip( savedClip );
   DisposeRgn( savedClip );
   DisposeRgn( screenRgn );
}
```

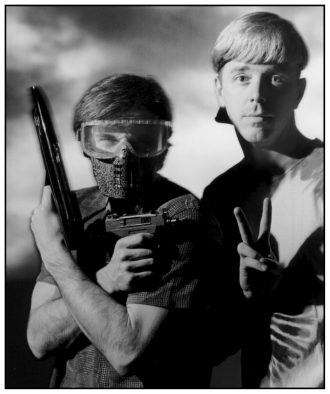
For information about using the Picture Utilities Package to find colors that are optimized for different screen depths, see the article "In Search of the Optimal Palette" later in this issue.•

```
enum { kDeepestScreen, kLargestAreaScreen };

GDHandle FindScreenGDevice(
   Rect*  bounds,        /* Global rectangle of part of screen to check */
   short  screenOption)  /* Use deepest or largest intersection area screen */
{
   GDHandle        baseGDevice;   /* GDevice that satisfies criterion */
   GDHandle        aGDevice;      /* Handle to each GDevice in the GDevice list */
   long            maxArea;       /* Largest intersection area found */
   long            area;          /* Area of rectangle of intersection */
   Rect            commonRect;    /* Rectangle of intersection */

   /* Different screen options require different algorithms */
   if (screenOption == kDeepestScreen)
       /* Graphics Devices Manager tells us the deepest intersecting screen */
       baseGDevice = GetMaxDevice( bounds );
   else if (screenOption == kLargestAreaScreen)
   {
       /* Get a handle to the first GDevice in the device list */
       aGDevice = GetDeviceList();

       /* Keep looping until all GDevices have been checked */
       maxArea = 0;
       baseGDevice = nil;
       while (aGDevice != nil)
       {
          /* Check to see whether screen rectangle and bounds intersect */
          if (SectRect( &(**aGDevice).gdRect, bounds, &commonRect ))
          {
             /* Calculate area of intersection */
             area = (long)(commonRect.bottom - commonRect.top) *
                    (long)(commonRect.right - commonRect.left);

             /* Keep track of largest area of intersection found so far */
             if (area > maxArea)
             {
                maxArea = area;
                baseGDevice = aGDevice;
             }
          }
          /* Go to the next GDevice in the device list */
          aGDevice = GetNextDevice( aGDevice );
       }
   }
   return baseGDevice;
}
```

Thanks to Edgar Lee, Guillermo Ortiz, and John Wang for reviewing this column.•

DRAWING IN GWORLDS FOR SPEED AND VERSATILITY

32-Bit QuickDraw brought system support for off-screen drawing worlds to the Macintosh, and Color QuickDraw continues this support in System 7. Using custom drawing routines in off-screen worlds can increase a program's speed and image-processing versatility. This article describes custom drawing routines that do just that.



KONSTANTIN OTHMER AND MIKE REED

It's a basic rule of Macintosh programming never to write a drawing routine that draws directly to the screen. There are two good reasons for this rule. First, multiple clients share the screen, and custom routines that draw directly to the screen may violate cooperation rules (new ones are being invented all the time). Second, support for new types of displays may be added to QuickDraw (as was the case with 32-Bit QuickDraw), and custom routines that draw directly to the screen certainly won't work when new display types are introduced.

So if your application has a drawing need that QuickDraw cannot fulfill, off-screen drawing is the only way to go. Your application draws to an off-screen copy of the application window, and the off-screen image is transferred to the screen with QuickDraw's CopyBits procedure. In the off-screen environment your application is the sole proprietor, and support for new displays will not affect how the off-screen environment behaves. In addition, using CopyBits to transfer an off-screen image onto the screen enables fast and smooth updating.

There are a couple of different ways to create an off-screen drawing environment. The old-fashioned way is to create it by hand, an arduous task that results in all the structures being kept in main memory. The new, improved way is to create it with the NewGWorld call first made available by 32-Bit QuickDraw and now supported by Color QuickDraw in System 7. When this method is used, a copy of the GWorld can be cached on an accelerator card, thus enabling improved performance by minimizing NuBus™ traffic during drawing operations. (For a full comparison of drawing operations with and without the use of GWorlds, see "Macintosh Display Card 8•24 GC" in *develop* Issue 3.)

KONSTANTIN OTHMER AND MIKE REED are on the lam again. Kon was recently spotted toting a padded canvas carrying case the size and shape of a PowerBook 170 on board a camel en route to the Pyramids of Giza. A source close to Kon reports that he neglected to bring a charger and had to spend extended hours inside a pyramid letting pyramid power recharge his PowerBook. Mike was last seen working his way across India as a caddy for the Dalai Lama. Our sources say he's picking up spiritual enlightenment, total consciousness, and a steady downstroke with his five iron. Exercise caution if you encounter either of these guys; they've corrupted system heaps in the past and could do so again.•

Given that you must certainly see the wisdom of using GWorlds in applications, we'll now move on to the good stuff—how to increase performance and create some interesting special effects with custom drawing routines. You should know the basics of creating and disposing of GWorlds to get the most from this article. If you need a review of these basics, read "Braving Offscreen Worlds" in *develop* Issue 1 or see Chapter 21 of *Inside Macintosh* Volume VI.

CUSTOM DRAWING ROUTINES TO INCREASE SPEED

Sometimes QuickDraw works too slowly for some of us. Whereas QuickDraw often trades performance for flexibility, there are times we'd just as soon trade flexibility for performance. In those cases, we can achieve tremendous gains in speed by writing custom routines to draw to off-screen worlds. Before writing such a routine, though, we need to understand what slows QuickDraw down.

WHY IS QUICKDRAW OFTEN SO SLOW?

Let's examine EraseRect to help us understand the considerable overhead QuickDraw has to deal with just to perform a simple operation. An EraseRect call is issued via a trap, so right off the bat we incur the overhead of the trap dispatcher. For a complex operation, this overhead is relatively small, but for a simple operation performed repetitively, this overhead can be significant. In the latter case, the trap dispatcher overhead can be avoided by calling GetTrapAddress and then calling the routine directly. (Note that with high-level routines, some traps take a selector.)

After we've called the routine, QuickDraw must do the following setup:

1. Check for a bottleneck procedure in the current port.
2. Check whether picture recording is enabled.
3. Calculate the intersection of the clipRgn and the visRgn and see if the drawing will be clipped out.
4. Check whether drawing is to the screen, and if so shield the cursor if the drawing intersects the cursor location.
5. Walk the device list and draw to each monitor that the clipped rectangle intersects.

Then the drawing takes place, consisting of these steps:

6. If the pixel map requires 32-bit addressing, enter 32-bit mode.
7. Determine the transfer mode to draw with.
8. Convert the pattern to the correct depth and alignment for this drawing.
9. Determine how to color the pixel pattern using the colors from the port.

10. Blast the bits.

The teardown consists of two steps:

11. Exit 32-bit addressing mode, if appropriate.
12. Unshield the cursor.

Notice that this list doesn't include error checking. QuickDraw does do some error checking, but rigorous checking slows performance further. While many of the items on this list are a simple check, others require considerable processor time. There's plenty of room here for reducing overhead by writing custom routines.

Custom routines can often skip all but step 10. For drawing operations that spend the majority of time in step 10, custom routines can't offer big wins in performance. But for operations that spend most of the time elsewhere, custom routines can achieve significant performance gains.

For example, if you copy a large image with CopyBits and the source and destination pixel depths are the same, the fgColor is black and the bkColor is white, the color tables match, the clipping regions are rectangular, and the alignment is the same, the operation is already very efficient since the majority of time is spent moving the bits rather than doing overhead. In this situation, you can't hope to gain substantial speed with a custom drawing routine. In contrast, for an operation such as setting a single pixel, the overhead involved in setting up the drawing operation eclipses the time actually spent drawing, so this is a candidate for a custom drawing routine.

OPTIMIZING A CUSTOM ROUTINE TO SET A SINGLE PIXEL

The simplest drawing to an off-screen world is setting a single pixel. Let's compare how QuickDraw sets a single pixel with how a custom drawing routine might do it. For our custom routine, we'll assume the off-screen world is 32 bits deep. This assumption gives us significant gains in speed and reduces code size and complexity.

Our sample code inverts the red and green channels. Figure 1 illustrates the transformation this accomplishes. Using QuickDraw, the code looks like this:

```
for (y = bounds.top; y < bounds.bottom; y++)
{
   for (x = bounds.left; x < bounds.right; x++)
   {
      GetCPixel(x, y, &myRGB);
      myRGB.red ^= 0xFFFF;      /* Invert the red and green channels. */
      myRGB.green ^= 0xFFFF;
      SetCPixel(x, y, &myRGB);
   }
}
```

For details of how CopyBits works in System 7, see "QuickDraw's CopyBits Procedure" in *develop* Issue 6.•

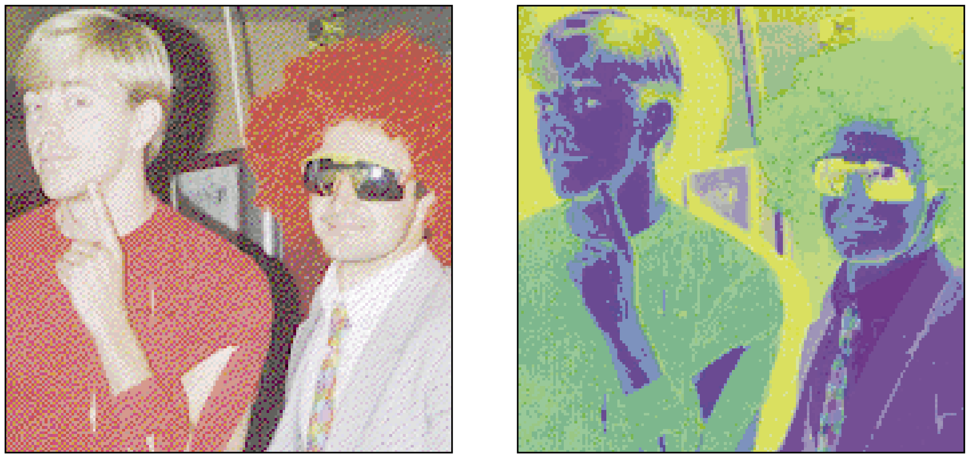


Figure 1
A Couple of Crazy Guys, Before and After Red/Green Inversion

As shown here, we use the QuickDraw routines GetCPixel and SetCPixel to get and set the color of a single pixel. SetCPixel is converted to a line-drawing command, because setting a single pixel is actually a special case of drawing a line (a *very* short line!). This way of implementing pixel setting is advantageous because line-drawing operations are saved in pictures and use the pattern and transfer mode from the port. It also simplifies QuickDraw on the bottleneck level since no separate bottleneck routine exists for setting pixels. The downside is that setting a single pixel this way is slow. To produce the transformation shown in Figure 1, the code takes 624 ticks or about 10.4 seconds to run on a Macintosh IIfx.

Faster. Now let's develop a custom routine that optimizes setting a single pixel. For a first pass, we'll eliminate the majority of the overhead and set the pixel directly rather than do line drawing. Given a GWorldPtr, an x and y position, and a 32-bit value, our routine GWSet32PixelC sets the pixel at that position to that value. The parallel call GWGet32PixelC is identical, except that where GWSet32PixelC sets the value, GWGet32PixelC returns it.

```
GWSet32PixelC(GWorldPtr src, short x, short y, long pixelValue)
{
PixMapHandle    srcPixMap;
unsigned short  srcRowBytes;
long            srcBaseAddr;
long            srcAddr;
char            mmuMode;
```

```
   srcPixMap = GetGWorldPixMap(src);
   /* Get the address of the pixels. */
   srcBaseAddr = (long) GetPixBaseAddr(srcPixMap);
   /* Get the row increment. */
   srcRowBytes = (**srcPixMap).rowBytes & 0x7fff;

   /* Make coordinates pixel map relative. */
   x -= (**srcPixMap).bounds.left;
   y -= (**srcPixMap).bounds.top;

   mmuMode = true32b;
   SwapMMUMode(&mmuMode);       /* Set the MMU to 32-bit mode. */

   /* Calculate the address of the pixel:  base + y*(row size in
      bytes) + x*(bytes/pixel). */
   srcAddr = srcBaseAddr + (long)y*srcRowBytes + (x << 2);
   *((long *)srcAddr) = pixelValue;
   SwapMMUMode(&mmuMode);    /* Restore the previous MMU mode. */
}
```

Of interest in this code is the call to SwapMMUMode before drawing to the GWorld. This is necessary since the GWorld could be cached on an accelerator card and require 32-bit addressing to access it. (See "QuickDraw's CopyBits Procedure" in *develop* Issue 6 for a complete explanation.)

If we revise our sample code to use our new calls GWGet32PixelC and GWSet32PixelC, the image shown in Figure 1 takes 398 ticks (or 6.8 seconds) to process. This is about 65% faster than QuickDraw, but is still much slower than it needs to be.

And faster. There are two major inefficiencies in our sample code: it makes four trap calls and it's at the mercy of the C compiler. Both of these problems are easily overcome, as the FastGWSet32Pixel routine demonstrates. Rather than take a GWorldPtr, FastGWSet32Pixel takes a pixMap pointer and a base address. Furthermore, the routine assumes it's being called in 32-bit mode. Note that the variables **bounds**, **top**, **rowBytes**, and **left** are defined in QuickEquate.a.

```
FastGWSet32Pixel(PixMap *srcPixMap, long *srcBaseAddr, short x,
   short y, long pixelValue)
{

   asm {
      move.l   srcPixMap,a0       ;Must be 32-bit-clean pointer
      move.w   y,d1               ;Get y
      sub.w    bounds+top(a0),d1  ;Make y bounds 0 relative
```

```
        move.w   rowBytes(a0),d0    ;Get rowBytes
        and.w    #0x7FFF,d0         ;Strip bitmap/pixMap bit
        mulu.w   d0,d1              ;Calculate offset to start of this row
        move.l   srcBaseAddr,a1     ;Must be 32-bit base address
        adda.l   d1,a1              ;Calculate address of this row
        moveq    #0,d0              ;Extend x to a word
        move.w   x,d0
        sub.w    bounds+left(a0),d0 ;Make x bounds 0 relative
        lsl.w    #2,d0              ;Convert x to pixels (4 bytes/pixel)
        adda.l   d0,a1              ;Calculate pixel address
        move.l   pixelValue,(a1)
   }
}
```

You may wonder why this routine takes both a pixMap and a base address. Can't it just get the base address from the pixMap directly? The answer is no, since the base address of a GWorld can be a handle rather than a pointer and in the future might be something different again. You must pass in a base address that's good in 32-bit addressing mode. The GetPixBaseAddr routine called by GWSet32PixelC returns the correct base address given a pixMap.

Revising our sample code isn't as trivial as it was before because of the additional assumptions made by these fast get and set pixel routines. Here's the new version of the code:

```
/* Get pixMap's 32-bit base address. */
srcBaseAddr = (long *) GetPixBaseAddr(myPixMapHandle);
myPixMapPtr = *myPixMapHandle;

/* WARNING: The pixMapHandle is dereferenced throughout these next
   loops. The code makes sure memory will not move. In particular,
   it's important that the segment containing the FastGWGet32Pixel and
   FastGWSet32Pixel routines is already loaded or is in the same
   segment as the caller. Otherwise memory might move when the segment
   is loaded. A trick to make sure the segment is loaded is to call
   a routine in the same segment (or these routines themselves) before
   making assumptions about memory not moving. */

/* Make it 32-bit clean. */
LockPixels(myPixMapHandle);
myPixMapPtr = (PixMap *) StripAddress((Ptr)myPixMapPtr);

mmuMode = true32b;
SwapMMUMode(&mmuMode);       /* Set the MMU to 32-bit mode. */
```

```
for (y = bounds.top; y < bounds.bottom; y++)
{
   for (x = bounds.left; x < bounds.right; x++)
   {
      myPixel = FastGWGet32Pixel(myPixMapPtr, srcBaseAddr, x, y);
      myPixel ^= 0x00FFFF00;    /* Invert the red and green channels. */
      FastGWSet32Pixel(myPixMapPtr, srcBaseAddr, x, y, myPixel);
   }
}

SwapMMUMode(&mmuMode);    /* Set it back. */
UnlockPixels(myPixMapHandle);
```

Note that the code calls StripAddress on the pixMapPtr since it'll be used in 32-bit mode. Also note that although we locked the pixels, we didn't lock the pixMapHandle, so no operation that could move memory is performed. You must be careful that the get and set pixel routines are in the same segment as the code that's calling them, or the Segment Loader might move memory when the routines are called. If all these restrictions are too much to keep in mind, simply call HLock on the pixMapHandle, and then HUnlock when you've finished.

Our sample image now takes only 17 ticks to process, or about .3 second. This is a nearly 37-fold improvement over the first version. But we're not done yet.

And even faster. We're still performing two subroutine calls and two multiplies per pixel, a major inefficiency. The RedGreenInvert routine gets rid of this inefficiency by walking a GWorld's pixMap to invert the red and green channels.

```
RedGreenInvert(GWorldPtr src)
{
PixMapHandle    srcPixMap;
short           srcRowBytes;
long            *srcBaseAddr;
long            *srcAddr1;
char            mmuMode;
short           row, column;
short           width, height;

   srcPixMap = GetGWorldPixMap(src);

   if(LockPixels(srcPixMap))
   {
      /* Get the base address. */
      srcBaseAddr = (long *) GetPixBaseAddr(srcPixMap);
      /* Get the row increment. */
      srcRowBytes = (**srcPixMap).rowBytes & 0x7fff;
      width = (**srcPixMap).bounds.right-(**srcPixMap).bounds.left;
      height = (**srcPixMap).bounds.bottom-(**srcPixMap).bounds.top;
      mmuMode = true32b;
      SwapMMUMode(&mmuMode);       /* Set the MMU to 32-bit mode. */
      for (row = 0; row < height; row++)
      {
         srcAddr1 = srcBaseAddr;
         for (column = 0; column < width; column++)
         {
            /* Invert the red and green. Note that for 32-bit pixels,
               pixel memory is organized as XRGB, where each component is
               8 bits. */
            *srcAddr1++ ^= 0x00ffff00;      /* Bump to next pixel. */
         }
         /* Go to the next row. */
         srcBaseAddr = (long *) ((char *) srcBaseAddr + srcRowBytes);
      }
      SwapMMUMode(&mmuMode);    /* Restore the previous MMU mode. */
      UnlockPixels(srcPixMap);  /* Unlock the pixMap. */
   }
}
```

Using the RedGreenInvert routine, our sample image now takes 1 tick or 1/60th of a second to process. This is nearly 624 times faster than the original version! Even greater performance gains can be made by rewriting this routine in assembly, but that's left as an exercise for you.

The price you pay. Note that as performance increases, the flexibility of the code decreases. In our example, the original version of the code, which calls GetCPixel and SetCPixel, works for all pixel depths, is recorded into pictures, and does the actual drawing using QuickDraw. GWSet32PixelC works only on 32-bit off-screen pixMaps. FastGWSet32Pixel has the additional restriction that it has to be called in 32-bit addressing mode. And finally, RedGreenInvert performs only one specific operation on an entire 32-bit GWorld.

We've shown you the tremendous speed improvements you can achieve by writing custom drawing routines. Now we turn our attention to some useful code examples for manipulating images.

CUSTOM DRAWING ROUTINES TO MANIPULATE IMAGES

Some image transformations lend themselves to direct manipulation of GWorld data. For example, with custom drawing routines we can transform images off-screen in various ways, find the edges of an image, quickly scale an image for use as a mask, and fill a rectangle in real time. We present these custom routines here and on the *Developer CD Series* disc.

A CUSTOM ROUTINE TO TRANSFORM IMAGES

We can obtain a variety of special effects—including rotation, stretching, perspective transformation, and sine wave warping—by applying a mapping matrix to a source GWorld, resulting in a transformed destination GWorld. This technique requires us to access an image at fractional rather than integer pixel coordinates—that is, to do subpixel sampling rather than point sampling. Let's consider subpixel sampling first and then look at the mapping routine.

Subpixel sampling. In QuickDraw, pixels are defined only for integer coordinates and undefined elsewhere. The location of each pixel is defined by the pair of integer coordinates at its upper left corner. To determine the value a pixel has at a fractional location in the pixMap, we must use a filter function. The routine we provide to do subpixel sampling, called GetFractionalPixel, can use either of two types of filters: a box filter or a tent filter. If use_box_filter is defined in the GetFractionalPixel routine, a box filter is used; otherwise a tent filter is used.

Here's how the filters work. Suppose the grid with corners at (0,0) and (6,4) represents part of an image, which we want to sample at fractional position (2.667,1.75). We can visualize each filter as a geometric solid whose base covers the pixel(s) on the grid to be sampled and whose height represents the weight to assign to the sampled pixel(s).

The box filter can be visualized as a 1-pixel x 1-pixel x 1-pixel cube that we plop down on the grid with its bottom right corner at the fractional coordinates, as shown in Figure 2. The box filter merely chooses the value of the pixel whose integer coordinates are covered by its base—in this case (2,1)—and gives this value a weight of 1, since the box is 1 pixel high above the integer coordinates. This value is then used to represent the image at the requested fractional position.

While the box filter takes the value of one integer pixel location in the source to represent a fractional pixel location, the tent filter averages the weighted values of up to four adjacent pixels in the source to represent a fractional pixel location. The tent filter can be visualized as a tent with a 2-pixel-square base that we plop down on the grid with the exact center of its base at the fractional coordinates, as shown in Figure 3. The tent filter takes the values of the one, two, or four pixels whose integer coordinates are covered by its base—in this case (2,1), (3,1), (2,2), and (3,2)—and gives each value a different weight. As with the box filter, the weighting for each value is determined by the height of the solid above each integer pixel coordinate. The average of these weighted values is then used to represent the image at the requested fractional position.

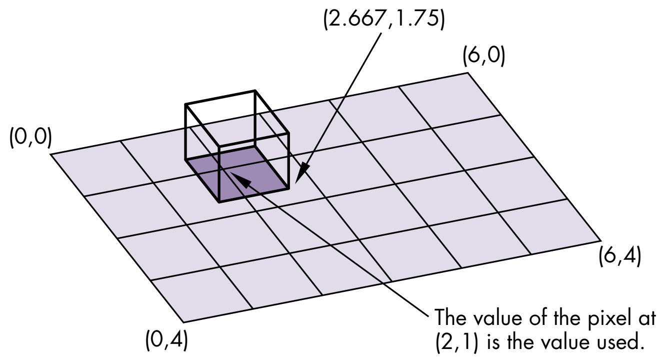


Figure 2
Subpixel Sampling Using a Box Filter

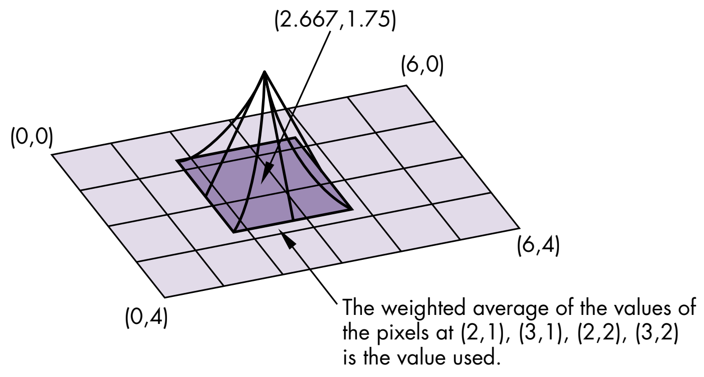
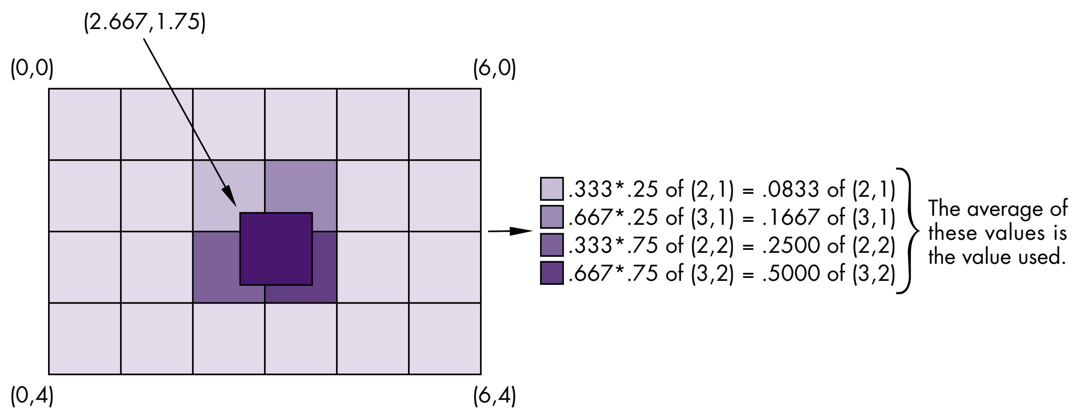

(2.667,1.75)
(6,0)
(0,0)
(6,4)
(0,4)
The weighted average of the values of the pixels at (2,1), (3,1), (2,2), (3,2) is the value used.

Figure 3
Subpixel Sampling Using a Tent Filter

Note in Figure 3 that the edge from the center of the side of the tent base to the top is linear ($z = x$), while the edge from the corner of the tent base to the top is quadratic ($z = x^2$). Note also that in two dimensions our tent filter is equivalent to imposing on the grid a pixel with its upper left corner at the fractional location, calculating what percentage of each of four integer coordinate pixels it overlaps, multiplying the value of each overlapped pixel by this respective percentage, and then averaging these values, as shown in Figure 4.

The box filter approximation makes the pixel value calculation easy, but results in some blurring of the image. The tent filter produces much better images than the box filter, but the calculation time for each pixel is considerably longer. Figure 5



(2.667,1.75)
(0,0)
(6,0)
(0,4)
(6,4)
.333*.25 of (2,1) = .0833 of (2,1)
.667*.25 of (3,1) = .1667 of (3,1)
.333*.75 of (2,2) = .2500 of (2,2)
.667*.75 of (3,2) = .5000 of (3,2)
The average of these values is the value used.

Figure 4
Two-Dimensional Equivalent of Our Tent Filter

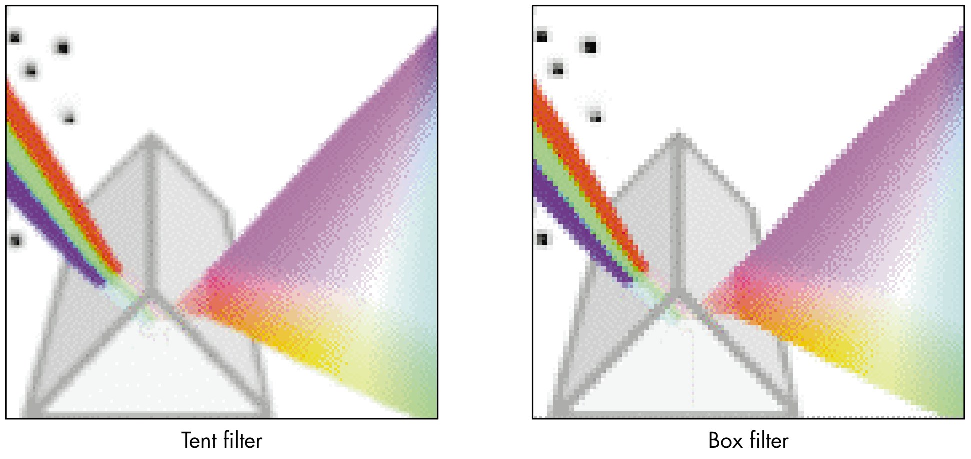


Figure 5
The 32-Bit QuickDraw Icon, Scaled With a Tent Filter Versus a Box Filter

illustrates the difference between an image scaled with a tent filter and with a box filter.

The GetFractionalPixel routine does all the work. The code first determines whether the requested pixel lies within the bounds of the source pixMap and returns false if it doesn't. Next, the code truncates pixels that lie off the right or bottom to fit wholly within the source. Finally, the pixels touched by the requested location are weighted and averaged (depending on the filter function) and the result is returned.

```
static Boolean GetFractionalPixel(long *srcBaseAddr, long srcRowLongs,
   Rect *srcBounds, Fixed fx, Fixed fy, long *dstLong)
{
unsigned long      tempBlue, tempGreen, tempRed;
long               srcPixel[2][2];
shortFrac          scale[2][2];
Point              p;

   SetPt(&p, fx >> 16, fy >> 16);
   /* ModelessPtInRect is a version of QuickDraw's PointInRect
      routine that works in either 24-bit or 32-bit addressing mode. */
   if (!ModelessPtInRect(p, srcBounds))
      return false;

   if (p.h == srcBounds->right - 1)
      fx = ff(p.h);   /* ff() is a macro that given a short,
                         returns a fixed. */
   if (p.v == srcBounds->bottom - 1)
      fy = ff(p.v);

   /* Compute the address of the first source pixel. */
   {
      long *srcAddr = srcBaseAddr + p.v * srcRowLongs + p.h;

#if use_box_filter
      *dstLong = *srcAddr;
      return true;
#endif
      srcPixel[0][0] = *srcAddr++;
      srcPixel[0][1] = *srcAddr++;
      srcAddr += srcRowLongs;
      srcPixel[1][1] = *--srcAddr;
      srcPixel[1][0] = *--srcAddr;
   }

   /* Precompute the scales for each pixel. ShortFracMul multiplies
      two short fractions and returns a short fraction. */
   {
      shortFrac xFrac = Fixed2ShortFrac((unsigned short)fx);
      shortFrac yFrac = Fixed2ShortFrac((unsigned short)fy);

      scale[0][0] = ShortFracMul(oneShortFrac - xFrac,
         oneShortFrac - yFrac);
      scale[0][1] = ShortFracMul(xFrac, oneShortFrac - yFrac);
      scale[1][0] = ShortFracMul(oneShortFrac - xFrac, yFrac);
      scale[1][1] = ShortFracMul(xFrac, yFrac);
   }

   /* Now scale each component of each corner by the percentage of the
      "real" pixel covered by the fractional pixel (the area covered
      by the filter). */
   tempBlue =      ShortFracMulByte(scale[0][0], srcPixel[0][0]) +
                   ShortFracMulByte(scale[0][1], srcPixel[0][1]) +
                   ShortFracMulByte(scale[1][0], srcPixel[1][0]) +
                   ShortFracMulByte(scale[1][1], srcPixel[1][1]);
   if (tempBlue == 256) tempBlue = 255;

   tempGreen =     ShortFracMulByte(scale[0][0], srcPixel[0][0] >> 8) +
                   ShortFracMulByte(scale[0][1], srcPixel[0][1] >> 8) +
                   ShortFracMulByte(scale[1][0], srcPixel[1][0] >> 8) +
                   ShortFracMulByte(scale[1][1], srcPixel[1][1] >> 8);
   if (tempGreen == 256) tempGreen = 255;

   tempRed =       ShortFracMulByte(scale[0][0], srcPixel[0][0] >> 16) +
                   ShortFracMulByte(scale[0][1], srcPixel[0][1] >> 16) +
                   ShortFracMulByte(scale[1][0], srcPixel[1][0] >> 16) +
                   ShortFracMulByte(scale[1][1], srcPixel[1][1] >> 16);
   if (tempRed == 256) tempRed = 255;

   *dstLong = (tempRed << 16) | (tempGreen << 8) | tempBlue;
   return true;
}
```

The mapping routine. Our custom routine to transform images by applying a mapping matrix makes use of the GetFractionalPixel routine. The mapping routine sets up a mapping matrix and then calls MapGWorld. MapGWorld takes a GWorld and applies the mapping matrix to it, resulting in a transformed GWorld. MapGWorld also takes a VAR mask parameter. If this parameter is set to nil, it's unused. Otherwise, MapGWorld returns a 1-bit mask that indicates which destination pixels were set. This mask can be passed as a parameter to CopyMask or CopyDeepMask to transfer the results onto the screen.

Figure 6 diagrams how MapGWorld works. The matrix shown there is the one that rotates an image 35 degrees from the x-axis toward the y-axis.

```
static GWorldPtr MapGWorld(GWorldPtr srcWorld, mapping *dstMapping,
   GWorldPtr *maskWorld)
{
GWorldPtr       dstWorld;
PixMapHandle    srcPixMap, dstPixMap;
```

Our matrix mapping functions use only a 3 x 2 matrix. Doing perspective transformations requires a 3 x 3 matrix.•

```
long        *srcBaseAddr, *dstBaseAddr, srcRowLongs, dstRowLongs;
Rect        srcRect, dstRect;
mapping     inverseMapping;
char        x, y, mmuMode;

   /* Create the dstWorld sized to hold the transformed srcWorld. */
   dstRect = srcRect = srcWorld->portRect;
   MapRectangle(dstMapping, &dstRect);
   if (NewGWorld(&dstWorld, 32, &dstRect, 0, 0, 0))
      return 0;

   /* Optionally, create a maskWorld with the same bounds as
      the dstWorld. */
   if (maskWorld)
   {
      if (NewGWorld(maskWorld, 1, &dstRect, 0, 0, 0))
      {
         DisposeGWorld(dstWorld);
         return 0;
      }
      EraseGWorld(*maskWorld, &dstRect);
   }

   /* Set up for fast walking of the src and dst. Need to swap
      MMU mode to look at the baseAddr. */
   srcPixMap = GetGWorldPixMap(srcWorld);
   dstPixMap = GetGWorldPixMap(dstWorld);
   /* Get the address of the pixMap. */
   srcBaseAddr = (long *) GetPixBaseAddr(srcPixMap);
   /* Get the row increment. */
   srcRowLongs = ((**srcPixMap).rowBytes & 0x7fff) >> 2;
   /* Get the address of the pixMap. */
   dstBaseAddr = (long *) GetPixBaseAddr(dstPixMap);
   /* Get the row increment. */
   dstRowLongs = ((**dstPixMap).rowBytes & 0x7fff) >> 2;

   inverseMapping = *dstMapping;
   InvertMapping(&inverseMapping);

   mmuMode = true 32b;
   SwapMMUMode(&mmuMode);    /* Set the MMU to 32-bit mode. */
   for (y = dstRect.top; y < dstRect.bottom; y++)
   {
      long *dstAddr = dstBaseAddr;
      for (x = dstRect.left; x < dstRect.right; x++)
      {
         Fixed srcX = ff(x);
         Fixed srcY = ff(y);

         MapXY(&inverseMapping, &srcX, &srcY);
         if (GetFractionalPixel(srcBaseAddr, srcRowLongs, &srcRect,
               srcX, srcY, dstAddr++) && maskWorld)
            SetGWorldPixel(*maskWorld, x, y);
      }
      dstBaseAddr += dstRowLongs;
   }
   SwapMMUMode(&mmuMode);    /* Restore the previous MMU mode. */
   return dstWorld;
}
```

The first thing MapGWorld does is call MapRectangle, which computes the size of the destination GWorld. (This and other subroutines can be found on the *Developer CD Series* disc.) It does this by applying the matrix transformation to the coordinates of the rectangle's four corners and then finding the tightest-fitting rectangle that contains the four points. It uses the same rectangle to make a 1-bit mask world, if needed.

Next, the matrix is inverted by calling InvertMapping, which establishes the inverse mapping—that is, the mapping from the destination to the source. Therefore, the matrix passed into MapGWorld must be invertible. (A commercial version of our routine would check for noninvertible matrixes and report an error.) If a transform that expands the source pixMap by 2 is used to map the source to the destination, only every other pixel in the destination is touched. So we walk the destination and use an inverse transform to find the source pixel location that maps to each destination location. This guarantees that each pixel in the destination is touched.

This brings us to an interesting observation: geometries, such as the bounding rectangle, are transformed from the source to the destination, but pixMaps use the inverse transform to walk the destination and map coordinates back to the source.

The next section of code walks the destination pixMap, performs the inverse mapping, and then calls GetFractionalPixel to put the pixel value directly in the destination GWorld. The corresponding entry in the mask pixMap is set if the requested pixel was in the range of the source pixMap and the mask GWorld exists.

A couple of the transformations that can be achieved with the MapGWorld routine are shown in Figure 7. The rotation is achieved by applying the matrix given in Figure 6. The stretching is achieved by scaling the mapping by 2 in the x direction.

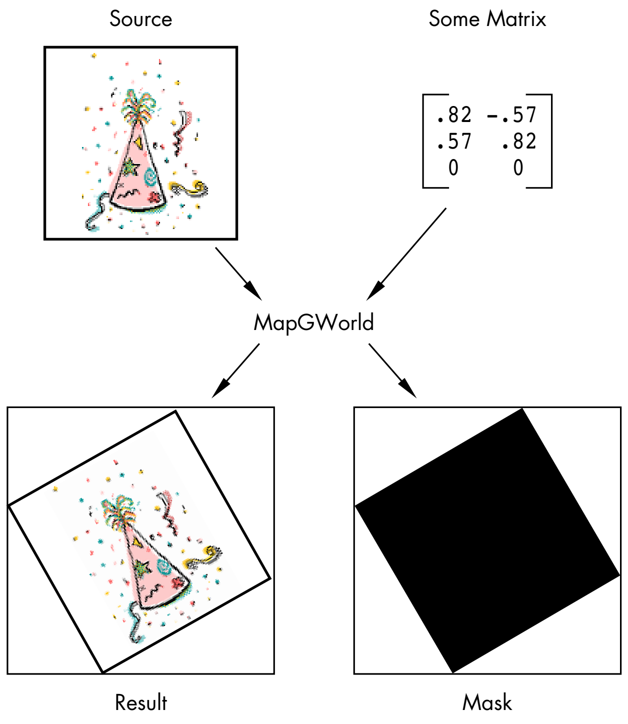


Figure 6
How MapGWorld Works

Included in the code on the *Developer CD Series* disc are routines for setting up the mapping matrix. RotateMapping sets up a matrix to perform rotation, as in

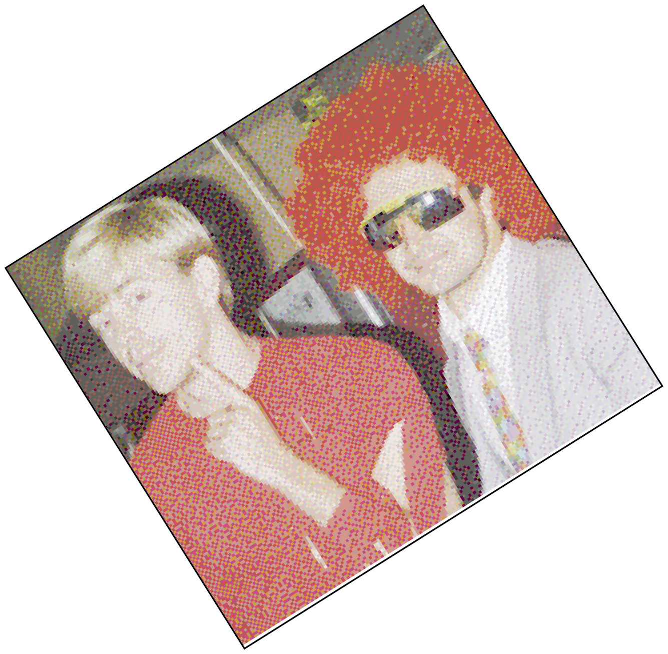
```
RotateMapping(&map, ff(35), ff(center.h), ff(center.v));
```

which rotates an image 35 degrees about some point specified by **center**. ScaleMapping sets up a matrix that performs stretching about a center, as in

```
ScaleMapping(&aMapping, ff(2), ff(1), ff(center.h), ff(center.v));
```

Note that MapGWorld is slow for the reasons discussed earlier. It's left as an exercise for you to optimize all or part of this code. Optimized assembly code for rotation can rotate a 400 x 400 8-bit pixMap at about five frames a second on a Macintosh II.

More tricks. We can perform nonlinear transformations as well. Consider the image in Figure 8. This sine wave warp was generated by transforming the coordinates of the source GWorld *after* they had been computed by sending the destination coordinates through the inverse of the mapping. The routine FancyMap performs a


Rotation

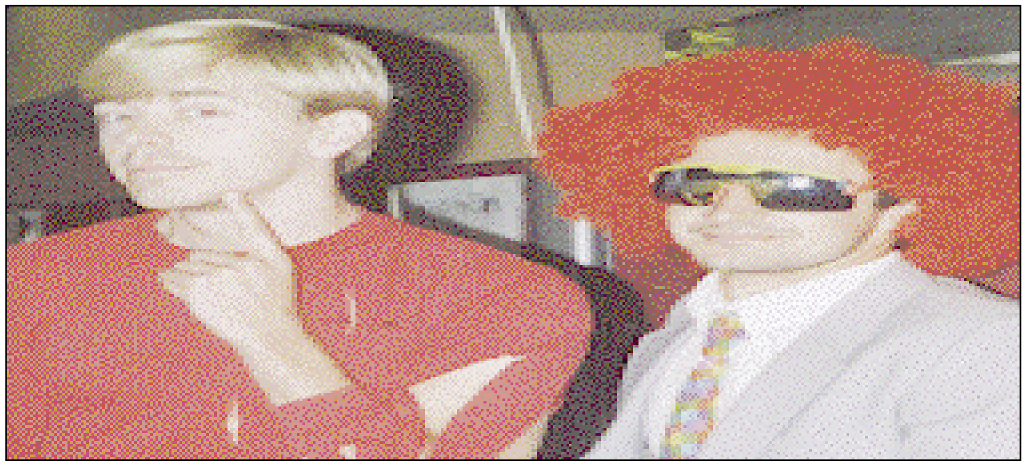

Stretching

Figure 7
A Couple of Crazy Guys, Transformed With MapGWorld

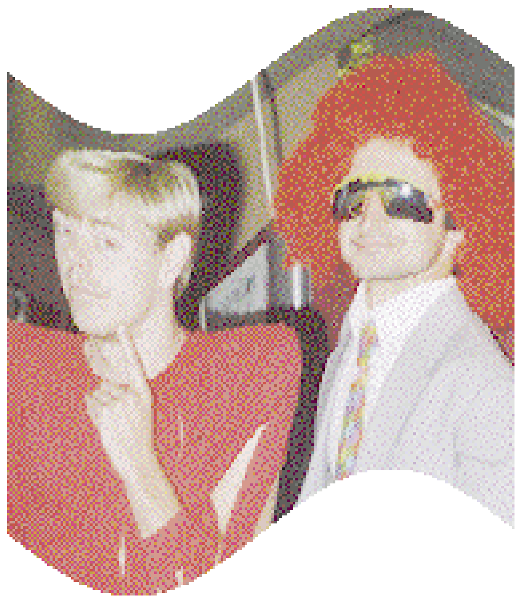


Figure 8
A Couple of Crazy Guys, Transformed Nonlinearly

simple trigonometric transformation of the y coordinate. It's not meant to be efficient, only to illustrate the flexibility provided by our MapGWorld routine.

```
static void FancyMap(const Rect *srcR, Fixed *xPtr, Fixed *yPtr)
{
   double x = *xPtr / 65536.0;
   double dx = 2 * pi() * (x - (srcR->right + srcR->left >> 1)) /
      (srcR->right - srcR->left);
   double amp = srcR->right - srcR->left >> 3;
   double delta = sin(dx) * amp;

   *yPtr += delta * 65536.0;
}
```

Other variations are easy to add. For example, sine and cosine can be used in combination to map an image onto a circle. If you expand the mapping to a full 3 x 3 matrix, you can draw images in perspective.

A CUSTOM ROUTINE TO FIND EDGES

Another example that lends itself to direct manipulation of GWorld data is finding edges by calculating the change in pixel values across a pixMap. Our CalculateDeltas routine does just that. First it copies the 32-bit pixMap down to a preallocated 8-bit gray-scale pixMap via CopyBits. This precalculates the luminance of each pixel in the source, rather than doing this individually on the fly. Next the routine calculates the difference between horizontally adjoining pixels and writes the result back out in place.

```
CalculateDeltas(GWorldPtr src, GWorldPtr dst)
{
PixMapHandle    srcPixMap, dstPixMap;
short           srcRowBytes, dstRowBytes;
long            *srcBaseAddr, *dstBaseAddr, *dstAddr;
unsigned char   *srcAddr1;
char            mmuMode;
short           row, column;
unsigned char   lum1,lum2;
unsigned long   dstLong;
short           width, height;
GDHandle        oldGD;
GWorldPtr       oldGW;

   srcPixMap = GetGWorldPixMap(src);
   dstPixMap = GetGWorldPixMap(dst);

   /* Lock the pixMaps. */
   if (LockPixels(srcPixMap) && LockPixels(dstPixMap))
   {
      GetGWorld(&oldGW, &oldGD);
      SetGWorld(dst, nil);
      CopyBits((BitMap*)*srcPixMap, (BitMap*)*dstPixMap,
         &(**srcPixMap).bounds, &(**dstPixMap).bounds, srcCopy, 0L);
      SetGWorld(oldGW, oldGD);

      srcBaseAddr = (long *) GetPixBaseAddr(srcPixMap);
      srcRowBytes = (**srcPixMap).rowBytes & 0x7fff;
      dstBaseAddr = (long *) GetPixBaseAddr(dstPixMap);
      dstRowBytes = (**dstPixMap).rowBytes & 0x7fff;
      width = (**srcPixMap).bounds.right - (**srcPixMap).bounds.left;
      height = (**srcPixMap).bounds.bottom - (**srcPixMap).bounds.top;

      mmuMode = true32b;
      SwapMMUMode(&mmuMode);       /* Set the MMU to 32-bit mode. */
      for (row = 0; row < height; row++)
      {
         srcAddr1 = (unsigned char *) dstBaseAddr;
         dstAddr = dstBaseAddr;
```

```
            lum1 = *srcAddr1++;   /* Get luminance of src pixel. */
            for (column = 0; column < ((width-1)/4); column++)
            {
               /* Do a long in the destination (4 pixels). This is OK
                  since memory blocks are always long word aligned. Thus,
                  we will never write over the right edge. */
               dstLong = 0;
               lum2 = *srcAddr1++;
               dstLong = (unsigned char) ((0x100 + lum1 - lum2)>>1);
               dstLong = dstLong << 8;
               lum1 = *srcAddr1++;
               dstLong |= (unsigned char) ((0x100 + lum2 - lum1)>>1);
               dstLong = dstLong << 8;
               lum2 = *srcAddr1++;
               dstLong |= (unsigned char) ((0x100 + lum1 - lum2)>>1);
               dstLong = dstLong << 8;
               lum1 = *srcAddr1++;
               dstLong |= (unsigned char) ((0x100 + lum2 - lum1)>>1);
               *dstAddr++ = dstLong;
            }
            /* Next row. */
            srcBaseAddr = (long *) ((char *) srcBaseAddr + srcRowBytes);
            /* Next row. */
            dstBaseAddr = (long *) ((char *) dstBaseAddr + dstRowBytes);
         }
         SwapMMUMode(&mmuMode);   /* Restore the previous MMU mode. */
         UnlockPixels(srcPixMap);
         UnlockPixels(dstPixMap);
      }
   }
```

Notice that the routine assumes the 8-bit gray-scale color table is linear. Furthermore, the routine does not deal with the right edge correctly. The problem is that if there are *n* pixels per row, there are *n*-1 deltas, so the resulting figure is one pixel narrower than the source. This routine puts garbage in the last pixel position in each row. When the image is displayed, you should shrink the image's bounds rectangle by one pixel on the right.

Figure 9 shows the result of applying the CalculateDeltas routine to one of our favorite images.

A CUSTOM ROUTINE TO SCALE IMAGES FOR MASK GENERATION

In our "Scoring Points With TrueType" article in Issue 7 of *develop*, we discuss a technique for drawing antialiased text using CopyDeepMask. There the mask for CopyDeepMask is created by (1) drawing the text at four times the target size into a

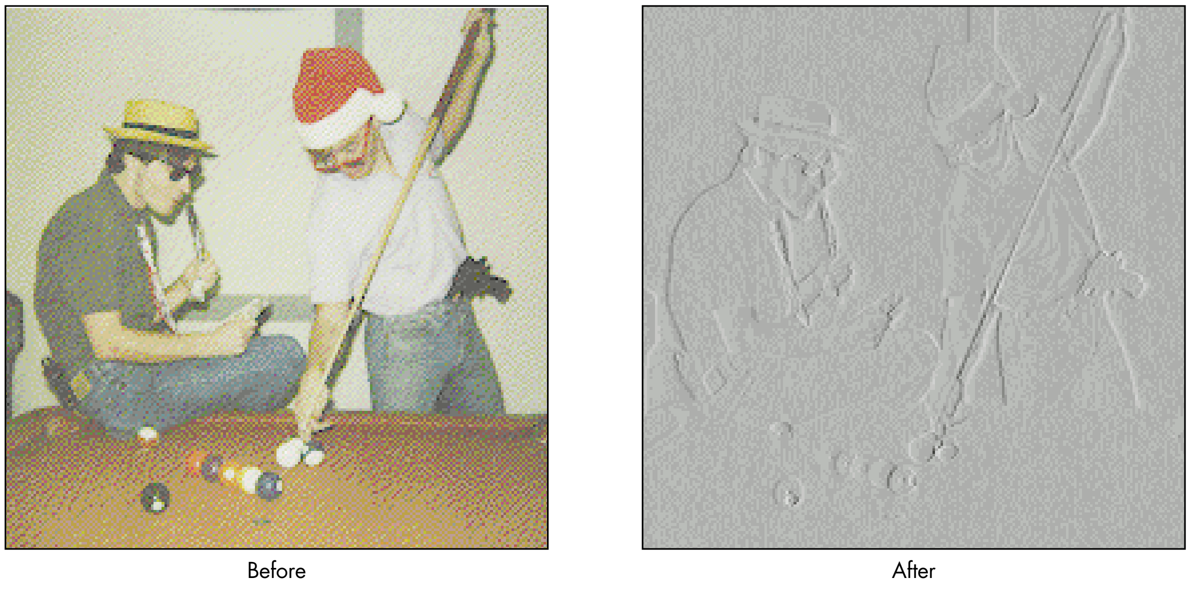


Figure 9
A Couple of Pool Sharks, Before and After Deltas Calculated

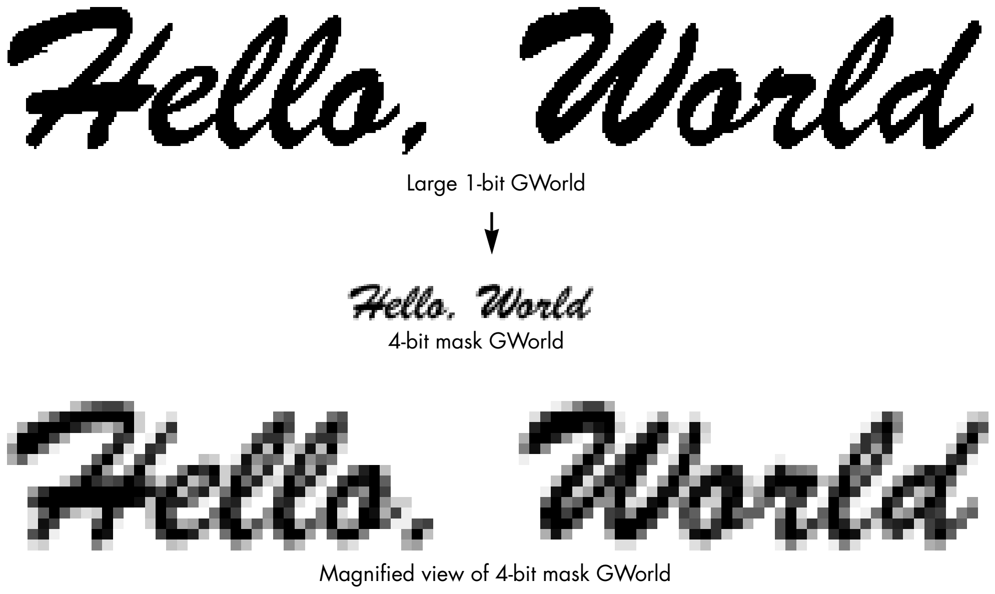


Figure 10
A "Hello, World" Mask Created With a Dithered CopyBits

1-bit GWorld and (2) using a dithered CopyBits to shrink the text to the target size in a 4-bit gray-scale GWorld.

While this technique produces very nice results, as shown in Figure 10, it's somewhat slow. One of the easiest ways to speed the routine up is to replace the dithered CopyBits with a custom routine that shrinks a 1-bit GWorld into a four times smaller 4-bit GWorld. Scale1BitTo4Bit is the routine we need.

```
Scale1BitTo4Bit(GWorldPtr src, GWorldPtr dst)
{
PixMapHandle    srcPixMap, dstPixMap;
short           srcRowBytes, dstRowBytes;
long            *srcBaseAddr, *srcAddr1, *srcAddr2, *srcAddr3, *srcAddr4;
long            *dstBaseAddr, *dstAddr;
long            thirtyTwoPixels1, thirtyTwoPixels2, thirtyTwoPixels3,
                thirtyTwoPixels4;
char            mmuMode;
short           row, column;
short           TranslationTable[256];
unsigned short  RemapTable[0x211]; /* See later comment on RemapTable. */
short           dstTenBits;
unsigned char   dstChar;
unsigned long   dstLong;
short           width, height;
short           index, index1;

   for (index = 0; index < 256; index++)
   {
      /* The translation table takes an index and counts the number of
         bits set. TranslationTable format:

         Low 5 bits contain number of bits set in low nibble of index.
         Next 5 bits contain number of bits set in high nibble.
         Top 6 bits always zero. */
      TranslationTable[index] = ((index & 1) != 0) + ((index & 2) != 0) +
                                ((index & 4) != 0) + ((index & 8) != 0) +
                                0x20 * ((index & 0x10) != 0) +
                                0x20 * ((index & 0x20) != 0) +
                                0x20 * ((index & 0x40) != 0) +
                                0x20 * ((index & 0x80) != 0);
   }

   /* The RemapTable converts a 10-bit number into an 8-bit value.
      The 10-bit number is a composite of two 5-bit numbers that
      can have values from 0-16.
      The result for each 5-bit input is:

      0-7       ->    0-7
      8         ->    7
      9-$10     ->    8-$F */

   for (index = 0; index <= 0x10; index++)
   {
      for (index1 = 0; index1 <= 0x10; index1++)
         RemapTable[(index << 5) + index1] = (char) ((index - index/8 +
            index/16) << 4) + (index1 - index1/8 + index1/16);
   }

   srcPixMap = GetGWorldPixMap(src);
   dstPixMap = GetGWorldPixMap(dst);
   /* Lock the pixMaps. */
   if (LockPixels(srcPixMap) && LockPixels(dstPixMap))
   {
      /* Get the address of the pixMap. */
      srcBaseAddr = (long *) GetPixBaseAddr(srcPixMap);
      /* Get the row increment. */
      srcRowBytes = (**srcPixMap).rowBytes & 0x7fff;
      /* Get the address of the pixMap. */
      dstBaseAddr = (long *) GetPixBaseAddr(dstPixMap);
      /* Get the row increment. */
      dstRowBytes = (**dstPixMap).rowBytes & 0x7fff;
      width = (**srcPixMap).bounds.right-(**srcPixMap).bounds.left;
      height = (**srcPixMap).bounds.bottom-(**srcPixMap).bounds.top;
      mmuMode = true32b;
      SwapMMUMode(&mmuMode);    /* Set the MMU to 32-bit mode. */

      for (row = 0; row < (height/4); row++)
      {
         /* Get addresses of first pixels in four rows of pixMap. */
         srcAddr1 = srcBaseAddr;
         srcAddr2 = (long *) ((char *) srcAddr1 + srcRowBytes);
         srcAddr3 = (long *) ((char *) srcAddr2 + srcRowBytes);
         srcAddr4 = (long *) ((char *) srcAddr3 + srcRowBytes);
         dstAddr = dstBaseAddr;
         for (column = 0; column < ((width+31)>>5); column++)
         {
            thirtyTwoPixels1 = *srcAddr1++; /* Get four longs of src. */
            thirtyTwoPixels2 = *srcAddr2++;
            thirtyTwoPixels3 = *srcAddr3++;
            thirtyTwoPixels4 = *srcAddr4++;
```

```
dstLong = 0;

/* Do eight bits of source. */
dstTenBits  = TranslationTable[thirtyTwoPixels1 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels2 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels3 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels4 & 0x000000FF];
dstChar = RemapTable[dstTenBits];
dstLong = dstChar;

/* Do second eight bits of source. */
thirtyTwoPixels1 >>= 8;         /* Move to second byte. */
thirtyTwoPixels2 >>= 8;
thirtyTwoPixels3 >>= 8;
thirtyTwoPixels4 >>= 8;

dstTenBits  = TranslationTable[thirtyTwoPixels1 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels2 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels3 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels4 & 0x000000FF];
dstChar = RemapTable[dstTenBits];
dstLong += (dstChar << 8);      /* No need to cast since C
                                   makes char into int. */

/* Do third eight bits of source. */
thirtyTwoPixels1 >>= 8;         /* Move to third byte. */
thirtyTwoPixels2 >>= 8;
thirtyTwoPixels3 >>= 8;
thirtyTwoPixels4 >>= 8;

dstTenBits  = TranslationTable[thirtyTwoPixels1 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels2 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels3 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels4 & 0x000000FF];
dstChar = RemapTable[dstTenBits];
dstLong += (long) ((long)dstChar << 16);

/* Do fourth eight bits of source. */
thirtyTwoPixels1 >>= 8;         /* Move to fourth byte. */
thirtyTwoPixels2 >>= 8;
thirtyTwoPixels3 >>= 8;
thirtyTwoPixels4 >>= 8;

dstTenBits  = TranslationTable[thirtyTwoPixels1 & 0x000000FF];
dstTenBits += TranslationTable[thirtyTwoPixels2 & 0x000000FF];
```

```
            dstTenBits += TranslationTable[thirtyTwoPixels3 & 0x000000FF];
            dstTenBits += TranslationTable[thirtyTwoPixels4 & 0x000000FF];
            dstChar = RemapTable[dstTenBits];
            dstLong += (long) ((long)dstChar << 24);

            *dstAddr++ = dstLong;
         }
         srcBaseAddr = (long *) ((char *) srcBaseAddr + 4 *
            srcRowBytes);         /* Next four rows. */
         dstBaseAddr = (long *) ((char *) dstBaseAddr +
            dstRowBytes);         /* Next row. */
      }
      SwapMMUMode(&mmuMode);   /* Restore the previous MMU mode. */
      UnlockPixels(srcPixMap);
      UnlockPixels(dstPixMap);
   }
}
```

Notice that the routine assumes the 4-bit destination pixMap is already allocated and has a linear gray-scale color table. Second, observe the use of a remapping table. This is necessary since each 1-bit x 4 x 4 patch in the source maps to a single 4-bit pixel in the destination. A 4 x 4 patch can have any value between 0 and 16, a total of 17 possibilities. Since a 4-bit pixel can hold only 16 different values, the code maps both values 7 and 8 in the source to a value of 7 in the destination via the remapping table.

A CUSTOM ROUTINE TO FILL RECTANGLES

Have you ever seen on TV the special effect of blocking out someone's face to preserve anonymity? We can create this effect ourselves with a custom drawing routine that simply undersamples the source image. While our custom drawing routine turns the whole image into blocks, it's easy for an application to block out just part of an image.

Our BlastRect routine simply fills a rectangle at a given x, y coordinate with the specified color in the prescribed GWorld. Note that this routine is a simple extension of the GWSet32PixelC routine we looked at earlier. Also note that we must make sure we don't fill beyond the right edge or the bottom of the pixMap.

```
void BlastRect(long value, Rect *rect, short x, short y, GWorldPtr dst)
{
PixMapHandle    dstPixMap;
short           dstRowBytes;
long            dstBaseAddr, dstAddr;
char            mmuMode;
short           row, column;
short           width, height;
```

```
   dstPixMap = GetGWorldPixMap(dst);

   /* Get the address of the pixMap. */
   dstBaseAddr = (long) GetPixBaseAddr(dstPixMap);
   /* Get the row increment. */
   dstRowBytes = (**dstPixMap).rowBytes & 0x7fff;

   if ((x + rect->right) < (**dstPixMap).bounds.right)
      width = rect->right - rect->left;
   else
      width = (**dstPixMap).bounds.right - (x + rect->left);

   if ((y + rect->bottom) < (**dstPixMap).bounds.bottom)
      height = rect->bottom - rect->top;
   else
      height = (**dstPixMap).bounds.bottom - (y + rect->top);

   /* Make x and y bounds relative. */
   x -= (**dstPixMap).bounds.left;
   y -= (**dstPixMap).bounds.top;

   dstBaseAddr = dstBaseAddr + (long) y*dstRowBytes + (x << 2);
   mmuMode = true32b;
   SwapMMUMode(&mmuMode);   /* Set the MMU to 32-bit mode. */
   for (row = 0; row < height; row++)
   {
      dstAddr = dstBaseAddr;
      for (column = 0; column < width; column++)
      {
         *(long *) dstAddr = value;
         dstAddr += 4;
      }
      /* Go to the next row. */
      dstBaseAddr = (long) ((char *) dstBaseAddr + dstRowBytes);
   }
   SwapMMUMode(&mmuMode);   /* Restore the previous MMU mode. */
}
```

Our UnderSampleGWorld routine, given a GWorld and a rectangle, undersamples the GWorld's pixMap at the resolution of the supplied rectangle. The performance of this routine isn't too bad for large rectangle sizes since most of the time is then spent in the inner loop of the BlastRect routine. When the image is only slightly undersampled, most of the time is spent doing overhead: recalculating the address where the fill starts, calling traps such as GetPixBaseAddr and SwapMMUMode, and calling a subroutine. The UnderSampleGWorld routine was used to hide the identity of the two people shown in Figure 11.

```
void UnderSampleGWorld(Rect *rect, GWorldPtr dst)
{
short           x, y;
PixMapHandle    dstPixMap;
long            value;

   dstPixMap = GetGWorldPixMap(dst);
   for (y = (**dstPixMap).bounds.top; y < (**dstPixMap).bounds.bottom;
      y += rect->bottom)
   {
      for (x = (**dstPixMap).bounds.left; x < (**dstPixMap).bounds.right;
         x += rect->right)
      {
         value = GWGet32PixelAsm(dst, x, y);
         BlastRect(value, rect, x, y, dst);
      }
   }
}
```

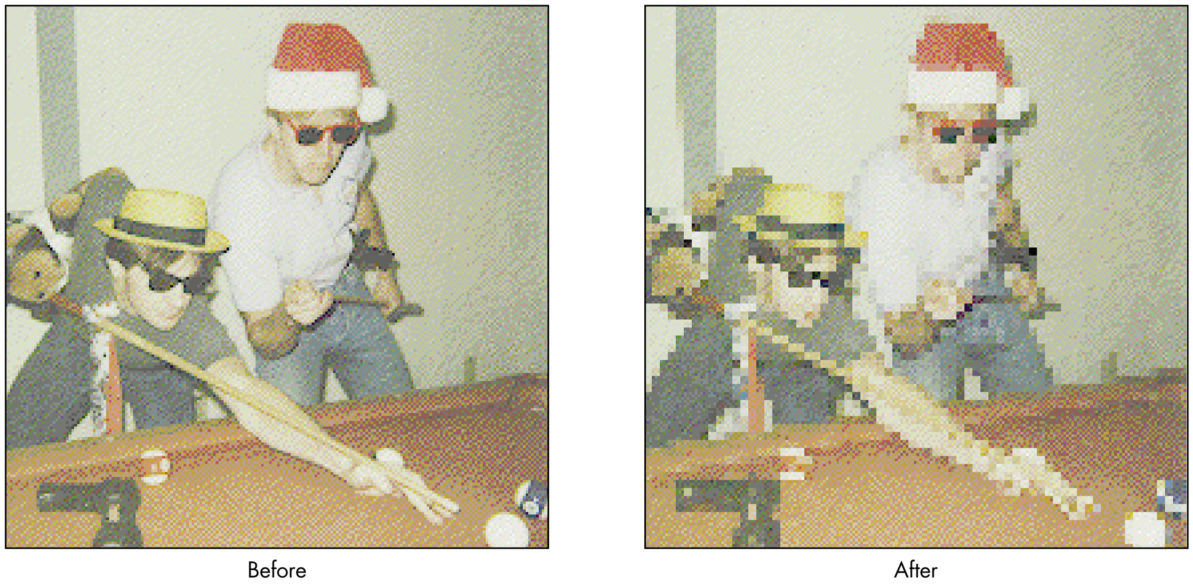


Figure 11
A Couple of Blockheads, Before and After UnderSampleGWorld

Again, it's left to you as an exercise to speed up this routine. With a little work, you should be able to munge an image in real time on a Macintosh II.

TO WRAP IT ALL UP

Now you know how to optimize code and manipulate images in various ways in GWorlds. You've learned how to design custom drawing routines to maximize speed rather than flexibility by cutting out unnecessary overhead. You've seen basic routines to do subpixel sampling, transform images by applying a mapping matrix to a source GWorld, calculate deltas, scale images for mask generation, and fill rectangles in close-to-real time. Now go exercise your new knowledge by optimizing those routines and have fun with some images of your own.

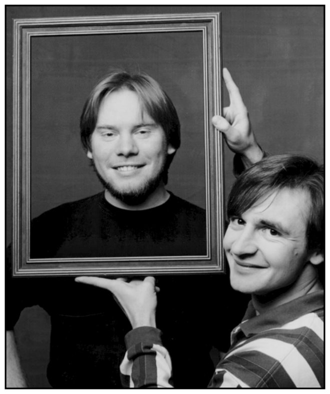
FURTHER READING

- *Computer Graphics: Principles and Practice,* 2nd ed., by J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes (Addison-Wesley, 1990). The standard text on computer graphics, offering a solid discussion of the basics.
- "Filters for Common Resampling Tasks" by Ken Turkowski, in *Graphics Gems,* Vol. 1 (pages 147–165), edited by A. S. Glassner (Academic Press, 1990). Describes filters that offer quality beyond that afforded by box and tent filters.
- *Digital Image Warping* by George Wolberg (IEEE Computer Society Press, 1990). All about different image processing effects, especially those used in movies. The discussion is very technical, very mathematical, really good on theory, and full of great ideas.
- *The Elements of Seven Card Stud* by Konstantin Othmer (Strategy One Publishing, 1989). Advanced strategy for seven card stud and psychology useful for all forms of poker. Contains numerous charts, tables, and graphs produced on the Macintosh. The strategies were developed through many hours of play as well as computer analysis of specific hands and situations. Available from Strategy One Publishing, P.O. Box 161544, Cupertino, CA 95016-1544, for $24.95 plus $2 shipping and 7% sales tax for California residents.

THANKS TO OUR TECHNICAL REVIEWERS
Keith McGreggor, Jean-Charles Mourey, Forrest Tanaka, Ken Turkowski•

IN SEARCH OF THE OPTIMAL PALETTE

When you want to display an image that contains more color information than the display device is capable of rendering, how do you pick the best colors to use? The Picture Utilities Package, new in System 7, provides two methods, which we describe here. You also have the option of developing your own color selection algorithm. This article and the accompanying code on the Developer CD Series disc will get you started.



DAVE GOOD AND KONSTANTIN OTHMER

It's tricky to display an image when the number of colors used in the source exceeds the number of colors available on the device. On an indexed device (256 or fewer colors), an application can choose, via the Palette Manager, which colors to use. But how will it know which colors are the best ones to choose, given a particular image?

To avoid this issue altogether, your application can simply draw the image and let QuickDraw use its default color palettes to make the choice. Because these palettes contain a well-dispersed set of colors, most images look pretty good. However, in the case of an image that uses an unbalanced set of colors, such as an underwater scene with many subtle shades of blue, relying on the default palette will not produce a good-looking result. In this case, you *must* tackle the issue of how to choose the optimal palette. That's when the new Picture Utilities Package can help.

Picture Utilities provides two methods—the popular method and the median method—for determining the best colors. This article describes these two methods. In addition, it describes a third method—the octree method—which, in addition to being useful in itself, makes a convenient starting point for you to develop your own algorithm for choosing the optimal color palette.

DECIDING WHICH METHOD TO USE

It would be nice if one method of selecting colors worked best for all types of images. But the truth is that the methods provided in Picture Utilities work best for some

DAVE ("KNOW") GOOD admits to never having graduated from grade school, high school, or college, though he's attended all three. Still, this didn't keep him from learning to program the Macintosh, and he's been employed by Apple for four years, working on things as varied as System Utilities 3.1 for the Apple IIc, TextEdit for the IIGS, and Picture Utilities (and other graphics-oriented work) for the Macintosh. Like most graphics programmers, he enjoys juggling, eating Chinese food, reading science fiction, and working from midafternoon till 4 or 5 in the morning. As a public service, he runs a talk-line at 976-DAVE. "If it's 3 A.M. and you're bored, just give me a call (techno-weenies only)!" On the rare occasions when Dave isn't at work, you can find him hiking in the Santa Cruz Mountains, riding his mountain bike, or skiing.•

types of images (such as those whose colors are all clustered in one small portion of the RGB cube), while QuickDraw's standard method works best for other types. Therefore, it's always important to give the user a choice of which Picture Utilities color-picking method to use and whether to use one of these at all.

The three methods we discuss here differ in how they approximate the ideal color set. The popular method bases its choices on a frequency count of colors used in the image, returning the most frequently used colors. Both the median and octree methods are algorithms that describe occupancy in a space. In this case, the space is the color cube with axes of red, green, and blue, and the occupants are the colors in the image. These algorithms differ in the way they divide up the space in order to return the correct number of colors. The median algorithm starts with one giant box covering the entire cube and splits it into successively smaller boxes; the octree algorithm starts with lots of tiny boxes and joins them into larger ones. Both methods return the weighted average of each of the boxes as the final set of colors.

The most appropriate method for your particular use depends on factors such as the type of image you want to display (real world, computer-generated, graphic, and so on), image content (perhaps the colors of items in the foreground are more important than the colors of items in the background), or even how the image will be displayed (halftoned or dithered, for example). For instance, none of these methods take dithering into account, although since we provide you with the source code, you could modify the octree method to do so.

The speed of each method also varies, with the popular method being fastest, the median method slower, and the octree method slowest, since in the latter there are more calculations involved for each color chosen. Also, the code that we supply for the octree method is intended to be easy to understand rather than blazingly fast. In fact, the current code is slower than the popular method by a factor of four, but with a little work this could probably be improved to be only twice as slow.

Another basic consideration is whether you want to represent the majority of colors in the image or the range of colors present. For example, if you could select only two colors to represent an image that contains several different shades of red and one blue dot, you would have to decide whether to pick two reds in an attempt to represent the majority of colors in the image, or one red and one blue to represent the range of colors the image contains. The popular method would do the former, while the median method would do the latter.

In general, the best method to use for an image that has a fairly well dispersed set of colors is QuickDraw's default palette. The popular method is useful when the source image contains only a few more colors than are available on the display device. For example, if you want to display a 32-bit image that uses only 200 distinct colors on an 8-bit device, the popular method is the best choice for speed and accuracy. While this case is trivial, using the popular method does guarantee that the needed colors will be

KONSTANTIN OTHMER has been known to frequent Garden City (a local poker club) and is said to be planning an early retirement from Apple.•

The RGB cube is pictured in color on the first page of *Inside Macintosh* Volume V and is described on page 43 of that volume.•

made available, a claim that can't be made for the default palette. The median and octree methods generally give the best results for images where small patches of a distinctly different color must be preserved at the cost of blending together large patches of similar colors.

Experience will give you a better feel for the strengths and weaknesses of each of these methods. Meanwhile, for purposes of comparison, Figure 1 shows screen snapshots of a 32-bit image as it originally appeared, using QuickDraw's default 16-color palette, using 16 popular colors, using 16 median colors, and using 16 octree colors. The original image has 77 different colors (to a resolution of five bits per color component). A test program on the *Developer CD Series* disc enables you to experiment with this image (or others) and to take a look at the code used to generate the various versions.

Notice in the original image that the colors marking the minutes follow a smooth progression from cyan on the far left to dark blue at the top to magenta on the right to purple on the bottom and then to dark red just before the cyan. Also notice the subtle color blending where the translucent minute and second hands intersect the underlying clock. When the standard 16-color palette is used, the soft colors of the minute markers change into much brighter, harder colors, and the smooth transitions are replaced by sudden transitions. The colors of the background and the face of the clock have changed. Furthermore, the subtle difference in color between the background and the background of the date (January 24) is lost.

The popular method preserves the colors of the largest color areas: the background, the clock face, the background of the date, the color of the minute and hour hands, and the lettering. The colors of the minute markers remain soft, but lose their shading resolution; for example, the cyan is replaced by a darker blue. Because it preserves the range of colors, the median method performs somewhat better on the minute markers than the previous two methods, but the clock face turns to black and the green hand becomes washed out. Although the image's appearance may not be ideal because many of the large areas are wrong, areas of the image that depend on the color ranges (which in this image just happen to occupy small areas) are reproduced more accurately. When the octree method is used, the result is similar to that of the median method, except the green hour hand is completely lost. This is due to the simple tree reduction algorithm we use; if the tree reduction improvements that we suggest at the end of the article were implemented, the hour hand most likely would be preserved, although its shade of green might change (much as happened with the median method). The octree performed better than the median in preserving the color of the text and the background of the date.

One conclusion from these images is that there is not a single best color-picking algorithm, even for a particular picture. For this image, we would be inclined to use the popular method, since we don't care too much about the subtle shading effects on the minute markers. However, an artist might be much more concerned with the

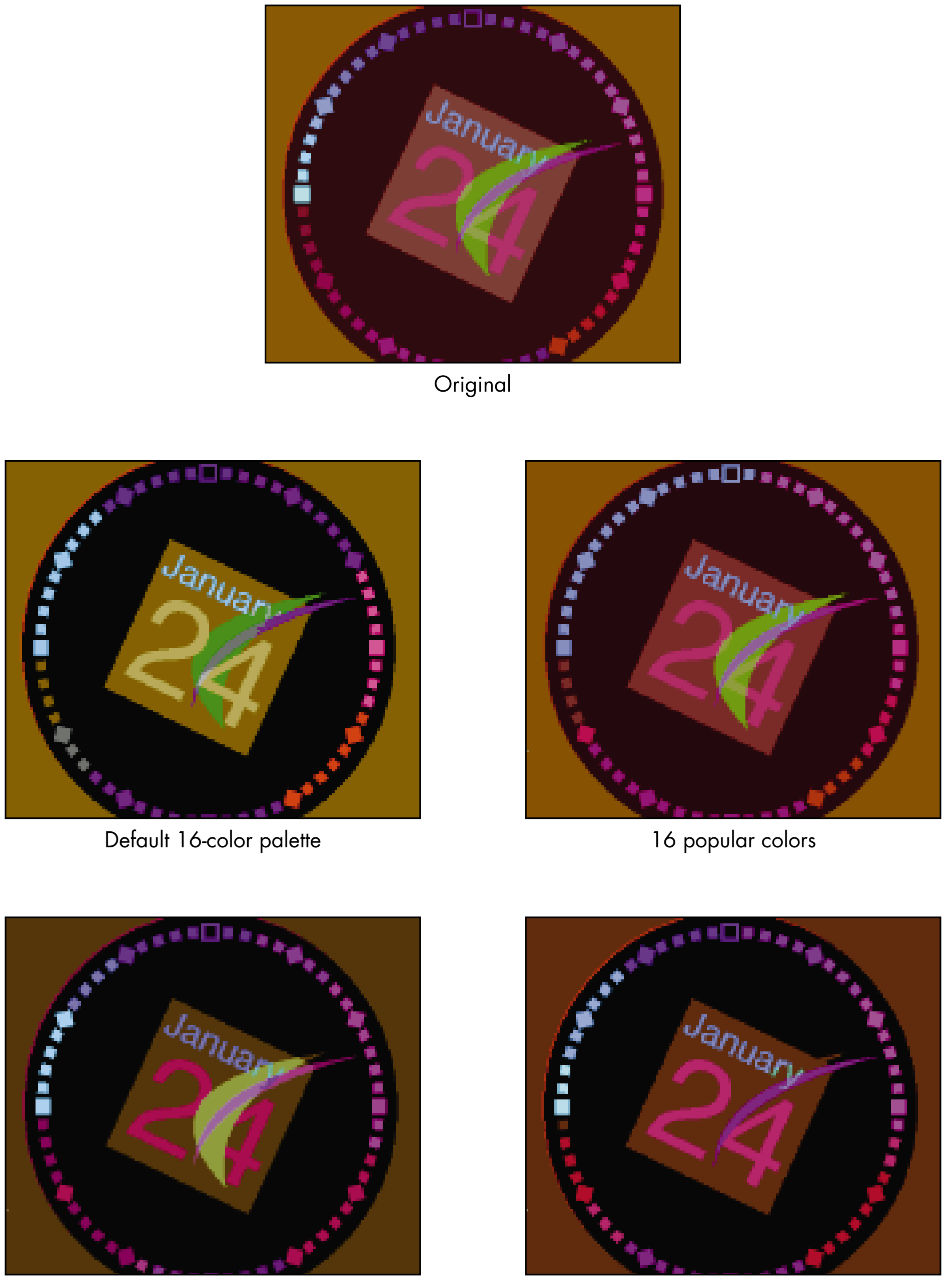


Figure 1
An Image Displayed Using Four Different Color-Picking Methods

subtle shading effects and actually not care if the face of the clock went to a completely different but still solid shade. In this case, the artist would probably pick the median or octree method. This is why applications that provide color optimization should allow the user to choose between the various available methods.

There is also a "system" method built into the Picture Utilities Package that tries to select the best general method available. Currently, the popular method is selected if the number of requested colors is 75% or greater of the total number of distinct colors in the image (to a resolution of five bits per color component); otherwise the median method is selected. The operation of this system method is almost certain to change in the future.

Now that you have some idea of how the available color-picking methods compare, we turn to details of how the popular, median, and octree methods work.

HOW THE POPULAR METHOD WORKS

The popular method of color selection is the easiest to understand and in general produces the least satisfactory results. This method chooses colors based on how frequently they're used in the image. The operation is performed by creating a histogram (a frequency count of each color) and then returning the colors that occur most often (as shown in Figure 2), up to the specified number of colors. If the image contains more than 256 different colors or if any of the source items are 32-bit pixMaps, Picture Utilities only maintains color information to a resolution of five bits per color component. Thus, colors must differ in the highest five bits of any of the three color components (red, green, and blue) to be considered distinct.

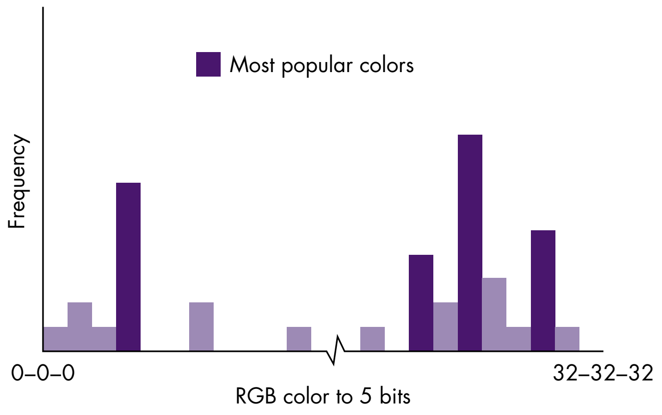


Figure 2
Picking Four Colors Using the Popular Method

In Figure 2, the x-axis represents each possible RGB color to a resolution of five bits per component. These colors range from 0-0-0 (0 red, 0 green, 0 blue) to 32-32-32 (32 red, 32 green, 32 blue) in the high five bits of the red-green-blue color components, for a total of 2^{15} or around 32,000 entries. The y-axis measures the frequency of each of the colors, up to a maximum of 32,767 occurrences. Thus, this table contains 2^{15} entries of one word each and occupies 64K of memory.

For custom methods, Picture Utilities can generate this histogram of color usage for you. If your custom method wants this, its initialization subroutine must return the value ColorBankIs555 for the color bank parameter. The octree method described later in this article does not use a histogram. Instead, it uses its own custom color bank and thus returns ColorBankIsCustom as its color bank parameter.

HOW THE MEDIAN METHOD WORKS

The median method is an iterative algorithm that views the colors in an image as if they were arranged in a cube with axes representing the red, green, and blue components. It starts by generating a histogram of the source colors, just as the popular method does. However, while the popular method is pretty much done after this, the median method's real work has only just begun.

The first real step is to determine the smallest RGB cube that will hold all the colors in the image. After finding the median color along the longest color axis, it then puts all the colors on one side of that color into one box and the remaining colors into another box. It continues this measuring and splitting process until the colors have been assigned to as many boxes as colors requested. Then the weighted average color of each of the boxes is returned as the color set to use.

Since the algorithm is much easier to visualize in two dimensions, we'll illustrate how it works in the red-green plane only. Extending the algorithm to three dimensions is straightforward. In Figure 3, eight colors are present in the red-green color plane (the blue component is taken as 0 for all colors). In this simplified example, each color occurs only once. In the general case, if a color occurs more than once, that color is weighted accordingly in the final step when the colors in each box are averaged.

The first division is along the green axis since the difference between the most green and the least green color is slightly larger than the difference between the most red and the least red color. This division results in the two boxes shown in step 2 of Figure 3. The largest difference in colors along one axis is now along the red axis of the top box. Thus, this box is divided into two along the red axis, leaving us with three boxes as shown in step 3. This time the largest difference is in the red axis of the bottom box, so this box is divided along the median, producing the result shown in step 4.

How direct pixel values are stored in memory is explained in Chapter 17 of *Inside Macintosh* Volume VI.•

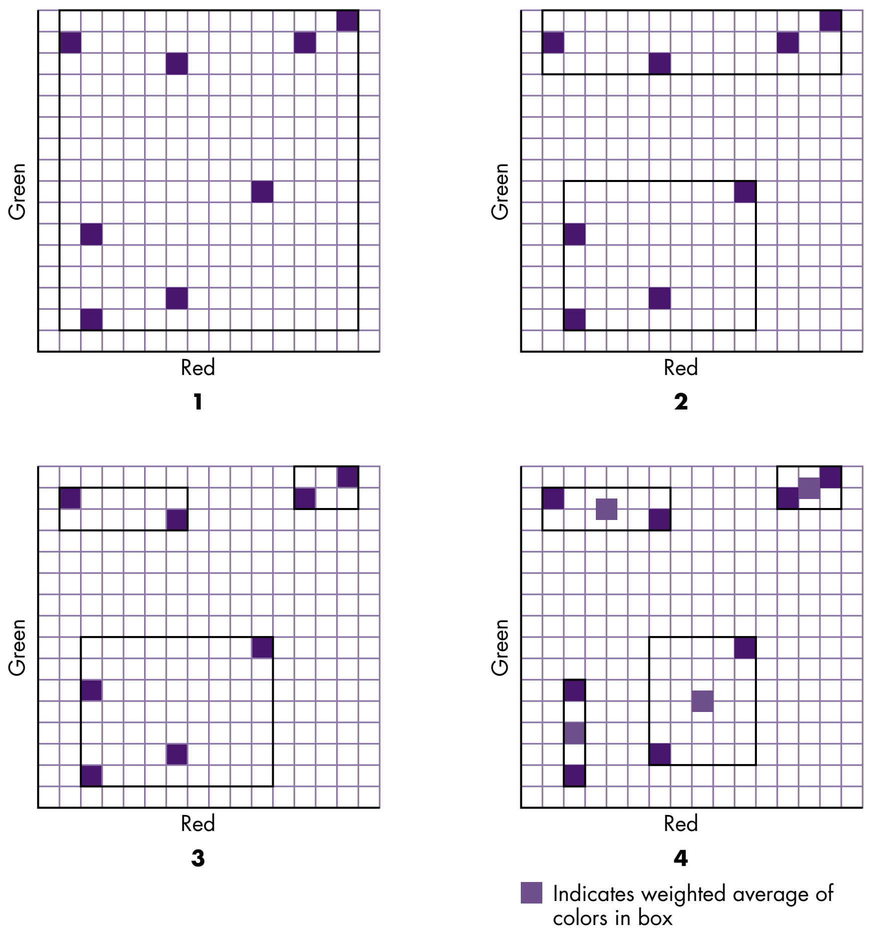


Figure 3
Picking Four Colors Using the Median Method

We now have four boxes, and since the best four colors were requested in this example, we're done. The colors returned are the weighted averages of the colors in each of the boxes, as shown.

HOW THE OCTREE METHOD WORKS

The octree method, like the median method, is an iterative algorithm that describes how the colors in an image are arranged in an RGB cube. Like the median method, the octree method groups the colors in an image together into the same number of boxes as colors requested and then returns weighted averages from these boxes, but the way these boxes are constructed differs substantially between the two methods.

While the popular and median methods process data that's stored in a histogram, the octree method does its color accounting via a tree. This means that rather than truncating colors to 5-5-5 from the get-go, the method maintains the full 8-8-8 color throughout the process.

An octree is similar to a binary tree, except that instead of having two branches at each node, it has eight. An octree corresponds to a cube with axes representing the red, green, and blue components of an image. At each level of the tree, the colors are placed in the branch of the tree indicated by the corresponding bits of the red, green, and blue components. For instance, say a color consists of one bit of red, one bit of blue, and one bit of green. The 0-0-0 (red-green-blue) color will be the first entry in the node, the 0-0-1 color will be the second entry in the node, and the 1-1-1 color will be the eighth entry in the node. Astute readers will notice that to determine which entry a color should be, we simply use the color value itself as a zero-based index into the node.

Our octree must deal with eight bits of red, green, and blue for each color, but this is an easy extension of the previous case. To handle multiple bits of color information, the code extracts the highest bit of each of the red, green, and blue components of the color and uses this as an index into the level-0 node to find the level-1 node. Then the next highest bit of each component is extracted and used as an index into this level-1 node to find the level-2 node. This process continues down to the lowest bit, which is used as an index into the level-7 node to find the color record (a level-8 leaf).

The actual color selection details of the octree method are much easier to understand in two dimensions, just as the core of the median method was. In two dimensions we're working with a quadtree instead of an octree. A quadtree has four branches at each node, as illustrated in Figure 4. (Note that below level 1 only one branch per node is followed to a deeper level, for the sake of simplifying the drawing. In reality, each node sprouts four branches, each of which in turn sprouts four more, and so on to the deepest level.) Colors are inserted into the quadtree much as they would be in an octree, except that the blue component is missing. For instance, the 0-0 (red-green) color is the first entry in the node, the 0-1 color is the second entry, and the 1-1 color is the fourth entry. Thus, the two-dimensional color value can still be used as a zero-based index into the node.

We think of this quadtree as corresponding to a coordinate plane formed by two color axes; each branch then corresponds to a quadrant of the space covered by the parent node. For instance, if a quadtree is being used to represent the red-green plane of an RGB cube with each axis ranging from 0 to 8, the four level-1 nodes represent the four quadrants of this plane defined by (red 0-4, green 0-4), (red 4-8, green 0-4), (red 0-4, green 4-8), and (red 4-8, green 4-8). The level-2 nodes of the (red 0-4, green 0-4) quadrant represent the four quarters of this particular quadrant, and so on down to the deepest level. Of course, in the case of the octree (the structure that our

How an octree works is discussed in more detail on pages 550–555 of *Computer Graphics: Principles and Practice,* 2nd ed., by Foley, Van Dam, Feiner, and Hughes (Addison-Wesley, 1990).•

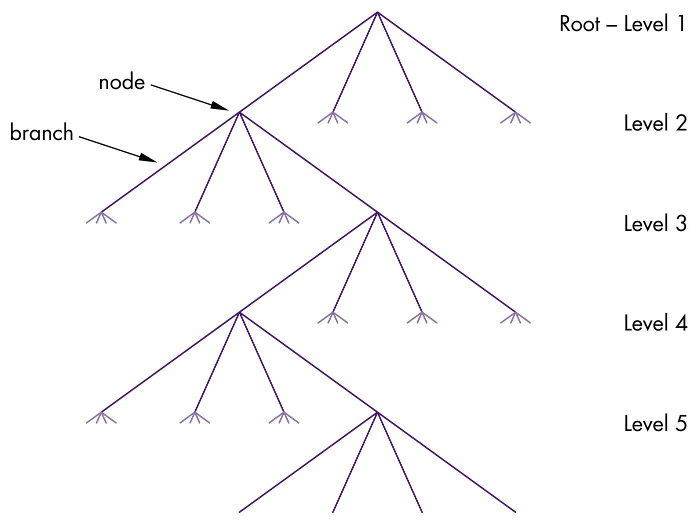


Figure 4
A Five-Level Quadtree

algorithm actually uses), the eight branches contained by a node correspond to the eight subcubes of the space represented by the entire node.

The octree algorithm first adds colors (leaves) to the tree until there is one more color (leaf) than requested. Then the tree is reduced so that there are no more leaves than colors requested. The reduction process starts on the deepest level and attempts to find two or more leaves that have the same parent. If this condition is not met on the deepest level, the search continues up to the next level until multiple leaves with the same parent are found. These leaves are then reduced to the parent node. The process continues until the tree contains no more than the number of colors (leaves) requested and no more colors remain to be added. The final color for each leaf in the tree is determined by calculating a weighted average of all the colors reduced to that leaf.

One major difference that distinguishes our octree implementation from the median or popular method is the way that colors are recorded. The octree method calculates the set of best colors (reduces nodes) as the source colors are being added to the tree, instead of counting and storing all the colors before beginning to pare them down. This is not an inherent limitation of octrees in general; we simply chose this implementation to reduce our memory requirements and to make these requirements independent of the complexity of the image. The disadvantage of this choice is that the octree method must make decisions on which colors to throw out based on incomplete information.

Another interesting difference is that the octree algorithm can actually return fewer colors than requested, instead of always returning the exact number. Theoretically, this can reduce the accuracy of the returned color set, since there are fewer colors to represent the full range and detail of the picture. However, we have found that most images do return either the full number of colors requested or only one or two fewer.

For purposes of comparison, we'll discuss how the octree method picks the best four colors in a color plane when given the same eight colors as used in the example for the median method. The process is illustrated in Figure 5. Note that the quadtree is represented there by a 16 x 16 grid; each position in the grid corresponds to one of the leaves on the fifth level of the quadtree shown in Figure 4.

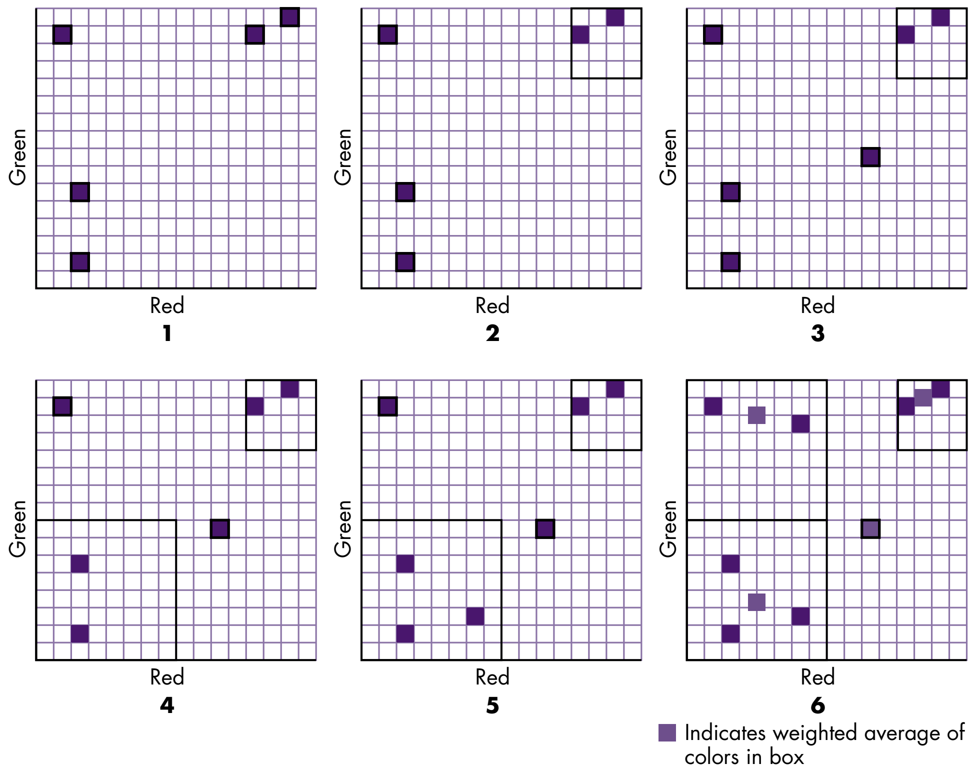


Figure 5
Picking Four Colors Using the Octree Method

In the grid, each color initially occupies its own tiny box. The color selection process translates into collecting the grid squares into bigger and bigger boxes until more than one color falls into a single box. We start with five colors in step 1, one more color than requested. The reduction from step 1 to step 2 in the figure involves reducing a node at the fourth level of the quadtree, creating a 4 x 4 box containing two colors. Had a fifth-level node containing two colors been found, the new box would be 2 x 2 rather than 4 x 4.

In step 3, another color is added and since it doesn't fall into an existing subdivision, the tree is reduced again. This time an 8 x 8 box, as shown in step 4, is formed. Step 5 adds another color, but since it falls in the 8 x 8 box, no further reduction is necessary. Step 6 adds the eighth and last color. It doesn't fall into an existing node, so a reduction is done and another 8 x 8 box is formed. As there are no more colors in the image, the weighted average of the colors in each of the boxes is returned and we're done.

Extending this algorithm to work in three dimensions rather than two is easy. The details of the implementation are in the source code, which can be found on the *Developer CD Series* disc. Once you've studied this source code you may want to write your own completely different color selection algorithm; Chapter 18 of *Inside Macintosh* Volume VI fully describes how to do this.

ADDITIONAL APPLICATION CODE

In addition to the code for the octree algorithm itself, we've supplied code on the CD for a crude test application that demonstrates how to apply both the popular and median methods as well as the octree method to a picture contained in a PICT file. The test application has a very simple user interface that allows you to open any PICT file and see what it looks like using the set of colors returned by the popular, median, and octree algorithms, as well as the standard system color table.

The source code for this application shows how to associate the color information that Picture Utilities returns with windows, and it also contains several useful routines for managing pictures stored on disk. These routines allow you to treat "disk pictures" as objects that can easily be passed around to various routines, drawn, profiled (using Picture Utilities), and even cached in memory if there's space to do so. Take a look at the actual source for the details on all this.

Our test application does not attempt to demonstrate how to use Picture Utilities to profile multiple pictures and pixMaps; see *Inside Macintosh* Volume VI for a complete explanation of how to do this.

ROOM FOR IMPROVEMENT

From the algorithms, it seems as though the median and octree methods should produce an excellent set of colors with which to display any image. Unfortunately, both algorithms fall somewhat short of this goal, especially with images that have a fairly well dispersed set of colors.

We can think of several areas where both could be improved. First, both algorithms weigh the red, green, and blue axes equally. As it turns out, the eye does the best job of perceiving green and a relatively poorer job of perceiving blue. Perhaps different weights could be assigned to the axes to compensate for this quirk of human perception. An advanced color selection method could take into account the fact that even within one color axis, the eye does not perceive the color (intensity and saturation) uniformly, and could compensate for variation in the gamma function of each monitor as well.

Second, in both methods the colors in the final boxes are averaged. While this produces the color that most accurately represents each box, it necessarily leaves some colors unreachable, even by dithering. A better choice of color might be the corner of the final box that's farthest away from the center of the RGB cube. This would assure that all the colors in the image would be within the cube of colors defined by the palette used to represent the image.

The median method isn't a realistic candidate for improvement since it's built into the Picture Utilities Package. On the other hand, we're providing you with the full source code for the octree method, so you're free to modify it. Since that's the case, we'll suggest a few other things that could be improved.

The most important part of the octree method is determining which nodes in the tree to reduce when the maximum number of colors has been reached. The algorithm we use in our sample code is much too simple; we simply scan through all the nodes at the deepest level looking for one that has two or more colors hanging off it. Then if we don't find any such nodes at the deepest level, we move up a level and look for ones that have two or more nodes hanging off them.

One better approach would be to reduce nodes that have the most discrete colors hanging off them; this would tend to blend similar shades into a single color, while preserving "fringe" colors. Another approach would be to reduce the nodes that have the most occurrences of colors hanging off them so that small splashes of color would not lose their distinctness.

Of course, if it's important to preserve subtle shades in the main areas of the picture and it's OK for fringe colors to be drastically modified, the previously mentioned improvements could be reversed to always reduce the nodes with the fewest discrete colors or the nodes with the fewest occurrences. A good general algorithm would probably try to incorporate both of these effects by having an occurrence threshold; colors that occur fewer than a certain number of times would be considered noise and either thrown out completely or reduced before more significant colors. In the case of colors that occur more than the threshold number of times, the algorithm could look for nodes with the most colors, since if a patch of fringe color is large enough to be noticeable, it should probably be preserved.

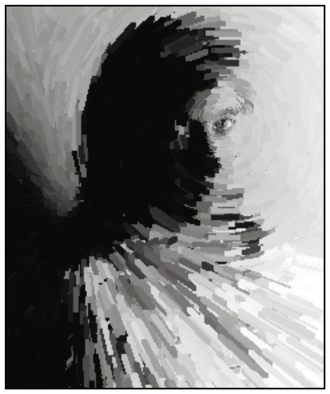
To properly implement these improvements, the octree method would probably need to be modified to collect all the colors before reducing any of them, to ensure accurate occurrence counts. Also, as we've mentioned before, currently our method can return fewer colors than were actually requested. If the code were modified to collect all the colors first, it would be relatively easy to ensure that the octree is never reduced below the requested number of colors. While overall this effect would probably be small compared to the benefit of a smarter reduction algorithm, it would substantially improve the quality of the rare images that degenerate under the current implementation and return many fewer colors than requested.

Other general improvements that could be made to the octree sample provided on the CD are dynamic memory management and overflow checking. We presently allocate all the memory that we'll need for the maximum size octree up front. While this is easy to do, it wastes space and prevents the code from being robust enough to support storing all the colors in the image before reducing any nodes. We also don't check for overflows when accumulating the average color represented by the node. Enhancements like this are very important if a piece of software is going to be more than just a sample. For example, both the built-in Picture Utilities methods do check for overflows and do some dynamic memory allocation.

HACK ON

This article has discussed techniques for picking the best colors for displaying an image that contains more color information than the display device is capable of rendering. While there is no one best algorithm for selecting colors in every case, experience will show you which of the three simple methods we've discussed is most appropriate in each case. With your new understanding of the popular and median methods offered by the Picture Utilities Package and with the source code sample for the octree method, you should find it easy to generate your own custom color selection algorithm.

THANKS TO OUR TECHNICAL REVIEWERS
Sean Callahan, Edgar Lee, Greg Marriot, Mike Reed, Steve Swen•



THE VETERAN NEOPHYTE

YEAH, BUT IS IT ART?

DAVE JOHNSON

Digital image processing has come a very long way. Remember when MacPaint® was a revolutionary concept? Now we've got a plethora of sophisticated graphics programs available on the Macintosh for regular folks: 32-bit painting programs, image-processing programs, CAD programs, photo-realistic rendering programs, solid modeling programs, animation programs—you name it. The power to be your best, or the power to run off into the weeds? I guess it depends on who's at the keyboard. One thing is sure: art will never be the same.

I've been messing around lately with digital filtering of scanned images: taking an existing image and applying some sort of mathematical transformation to it. The results are sometimes funny, sometimes beautiful, often ugly, but always fun.

One of the programs I've been spending time with lets you interactively type in mathematical expressions and apply them to an image. This application, called Pico, is a Macintosh implementation of an image-processing language developed by Gerard J. Holzmann at AT&T Bell Laboratories, and described in his book *Beyond Photography: The Digital Darkroom.* (The Macintosh version I've been using was written by John W. Peterson here at Apple, and I've included it on the *Developer CD Series* disc so that you can play with it, too.) If you're the least bit interested in image processing, you should read Holzmann's slim, friendly book. It describes the language in detail and gives lots of examples of its use. The book is full of fascinating photographs that have been tweaked and transformed using the language, ranging from the hilarious (in particular, see the Einstein caricature on page 35), to the sublimely beautiful (make up your own mind). The overall feel is one of whimsy and fun, with a strong dose of the joy of discovery. The book also includes a very instructive and in-depth discussion of the software that implements the language (a lexical analyzer, a recursive-descent parser, and an interpreter) and source code in C.

Holzmann's language allows you to invent, implement, and try out digital filters on the fly. It uses a C-like syntax and is decidedly mathematical, but that's where a lot of the fun comes in: seeing a photographic image quickly transformed by a simple mathematical formula is really fascinating. The language makes it easy to mess around and discover unusual things about math and filters: you can just type in an expression, hit the Enter key, and see the results immediately. The program operates only on 8-bit gray-scale images that are 256 by 256 pixels, but the power of the language far outshines this limitation. Try it, you'll like it.

There's another kind of digital filtering that I first learned about a little over a year ago in an article by Paul Haeberli at Silicon Graphics: *Paint By Numbers: Abstract Image Representations*, in the SIGGRAPH '90 Conference Proceedings. This is an interactive kind of filtering, which makes it a lot of fun. The concept is simple: Start with a given image, any image (call it the source). Create a new, blank one (the destination) that's the same size. Then you "paint" on the destination with the mouse, and at each point you touch, the color of the source image is determined at that same location. A "brush stroke" is then drawn in the destination at that position, with the source's color. If the brush just drew single pixels, you would be copying the source image exactly, which would be a pretty tedious way to copy it. Ah, but the brush can do anything it wants to, and that's where the fun begins. If the brush draws, say, a circle a few pixels in diameter, you get a sort of "blot" effect, with the blots overlapping each other

DAVE JOHNSON would like to share with you a quote from *The Three-Pound Universe* by Judith Hooper and Dick Teresi, a wonderful book about the human brain: "Research at Yale University's Center for Behavioral Medicine turned up the surprising fact that regular 'reading therapy'—that is, reading a book for half an hour a day—is neurophysiologically equivalent to the practice of TM. (Whether this means the Maharishi's mantras are comparable to *Heidi* or merely reflects the crudeness of our measuring devices, we'll let you decide.)" This finding vindicates Dave's habits in a way no heartfelt argument ever could.•

haphazardly. Or you could draw a line in some random direction from the source pixel's location, or add some noise to the color so that it varies a little from the source color, or draw a clump of dots centered at the source pixel, or draw a silhouette of a wiener dog in the appropriate color, or . . . the possibilities are endless. In a way you're tracing the source, but the brush you use to trace with isn't exact, and the results can be striking.

The finished images tend to look very "painterly" and are often evocative of impressionist paintings like those of Monet or Renoir, or of the pointillist "divisionist" technique of Seurat. (Can you tell I've been spending some time with my handy-dandy *Random House Encyclopedia*? Thanks, Mom.) This is a refreshing move away from the trend toward photo-realistic rendering that you see so much of in computer graphics.

I wrote a Macintosh application that implements a simplified version of what Haeberli did, so I could play around with it. (The application and all the source code are on this month's CD, for you to mess around with. If you find any problems, please let me know.) The most fun part turned out to be writing the brush routines, and I was curious to see just how hard it is to incorporate plug-ins into an application, so I made up a simple plug-in interface for the brushes (plug-ins are code resources separate from the application that are loaded and run as needed). Surprisingly, it turned out to be pretty trivial to implement plug-ins. I figured I was going to be forced to descend to the level of A5 worlds and code resource headers, but with the exception of one subtle gotcha it was easy. Basically you just get a handle to the code resource with GetResource, lock it, dereference it, and call it. I had to do some ugly casting to convince the C compiler to let me make the call, but other than that there were almost no problems.

One thing turned up that I couldn't figure out, and I was forced to seek help. I was writing a filter routine (the application supports both brushes and filters as plug-ins) that was a modified version of the RedGreenInvert routine from the article "Drawing in GWorlds for Speed and Versatility" in this issue. I first tried it linked into my application, so I could use the source debugger on it. Once it was working, I converted it to a plug-in and BOOM, it crashed with a bus error. Some investigative work pointed to SwapMMUMode as the culprit, but I couldn't figure out why. (Trumpet fanfare) Bo3b Johnson to the rescue once again! It turns out that since I was calling the plug-in in 24-bit mode, when it came time to call SwapMMUMode the PC contained an address that had some of its high bits set (in this case the "locked" bit since I had locked the handle and the "resource" bit since it was a handle to a code resource). This is a bad thing. The solution, of course, is to call StripAddress on the pointer to the plug-in before calling it. That way the address in the PC is clean and SwapMMUMode is happy.

There are several commercially available applications that do Haeberli-like image manipulation. Monet, by Delta Tao Software, is a much more complete and sophisticated implementation of Haeberli's concepts, incorporating some of the cooler features like opacity control, getting the direction of the brush strokes from the movement of the mouse, and so on. (You gotta love Delta Tao Software: when a customer asks for an IBM version of one of their products, they gleefully answer "Buy a Macintosh.") There's also Painter, a truly unique and remarkable paint program from Fractal Design that simulates very naturally the media artists use—chalk, pencil, charcoal, and so on. It has a "cloning" function that's similar to Haeberli's in concept: you can manually or automatically draw over the "source" image with any of the brushes. Aldus Gallery Effects by Silicon Beach Software is a product that basically consists of canned filters that can be applied to images to transform them in interesting ways, many of which are similar to the effects you get with Haeberli's technique. And then of course there's Adobe Photoshop, the brilliant, precocious teenager of image-processing programs.

Photoshop seems to have become the de facto industry standard image-processing program. Its versatility is, so far, unmatched by any other program I've seen. And its plug-in interface has also become something of a standard: many Macintosh graphics programs (Painter, for one) now support Photoshop plug-ins, and I know of at least one software company that does nothing *but* write Photoshop plug-ins. I think plug-ins are a very cool thing: they allow extension and customization of an application on the fly, bringing us a tiny step closer to the dream of "erector set" applications that can be taken apart, rearranged, and rebuilt by users to suit their needs. If you want to write some Photoshop plug-ins, you can find the documentation for the plug-in interface (along with examples in MPW C, MPW Pascal, and THINK C) on the CD.

Here's the big, deep question about these digitally transformed images: Are They Art? An image produced by any of these applications can indeed be "arty"; of that there is no doubt. But is it really art? Many graphic artists would immediately answer with a resounding "NO!" They'd say that it just looks like art, that it imitates art (kinda like life), but isn't really art because it's *automatic*. But many painters in the surrealist and abstract impressionist movements took great interest in what they termed "automatic" painting, painting without the intervention of conscious control. Jackson Pollock was a notable practitioner of this technique. Is the creation of a painting by automatic means any different, in principle, from what these computer-based tools do?

"Wait a minute!" these artists might cry, "Pollock began with a blank canvas! What he did was truly original! You (smug smirk) are just taking an existing image and transforming it with a computer. That's not art, that's just (expression of extreme distaste) *filtering*."

"Wipe that smug smirk off your face," I might smirk smugly, "what about the dadaists? They claimed that art was anything that anyone decided to *call* art, and I'm calling these images art." (Fun dadaist tales: Marcel Duchamp, a dadaist in New York, bought a urinal, signed it "R. Mutt," and called it art. He claimed that the signature alone made any manufactured item into a work of art. These "readymades," as he called them, sold quite well. And then there's Kurt Schwitters, a Hanover dadaist, who made collages from rubbish. Now the dadaist movement, admittedly, was intended to upset the status quo, rip apart the definition of art, and shock people out of their bourgeois sensibilities, but their influence is still strongly felt in modern art, and has forever muddied the definition of what art is. For that I heartily thank them.)

There's another eminently pragmatic definition of art: it's art if someone is willing to buy it. This one is distasteful in its materialistic slant, but I must admit that it's a useful one, at least to people who make a living making art. Then there's the Marshall McLuhan stance that "art is anything you can get away with." I personally love this one for its nebulousness, and I'm willing to leave it at that.

Whatever definition we pick, we still can't conclusively say whether these computer-transformed images are art. Art is just too slippery a thing to pin down, like trying to put a cloud in a chair. I think art is primarily a dialog between the creator and the experiencer: if something is communicated, I'll call it art. But in the final analysis, does it really matter? These tools are just another kind of computer fun, and *everyone*, artist or not, can play.

RECOMMENDED READING

- *Beyond Photography: The Digital Darkroom* by Gerard J. Holzmann (Prentice-Hall, 1988).
- *Paint By Numbers: Abstract Image Representations* by Paul Haeberli (in the SIGGRAPH '90 Conference Proceedings).
- *Elbert's Bad Word* by Audrey Wood (Harcourt Brace Jovanovich, 1988).

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

Q *Our application uses the movie poster as a still frame in a cell, similar to using a PICT. If a user sizes the cell width so that it's narrower than the poster, even though we clip the drawing to the cell size, QuickTime posters draw their full width, writing over whatever is in the way. Pictures clip through DrawPicture; why doesn't ShowMoviePoster stay within the clipping region?*

A ShowMoviePoster, as well as the movie and preview showing calls, uses the movie clipping characteristics rather than the destination port's clipping region. You must set the movie's clipping region to obtain the results you want. An easier way to do this is to get the picture for the poster by calling GetMoviePosterPict, and then simply use DrawPicture to display the poster. Because this is just a picture, the clipping region of the port is honored. This way you don't need different code for movies and pictures.

Q *Our QuickTime application gets a Sound Manager error -201 after playing movies in succession, apparently because sound channels used in the previous movies have not been reclaimed. How does QuickTime decide to deallocate sound channels? It doesn't seem to happen in my "while (!IsMovieDone(theMovie) && !Button())" play loop.*

A Sound channels are released by active movies when they notice that some other movie needs them. This is currently done only at MoviesTask time. Before entering your loop to play a single movie, you can do one or both of the following:

- Preroll the movie you're about to play and check the error. If preroll returns -201, call MoviesTask(0,0) to give the other active movies a chance to give up their sound channels. A subsequent preroll of theMovie should return noErr.
- Call SetMovieActive(otherMovies, FALSE). Deactivate the movies that you aren't playing to force them to give up their resources.

Q *When I select all frames in QuickTime and then do an MCCut or MCClear, the standard controller gets larger and redraws itself at the top of the movie. Is this a situation I should be prepared to handle or a bug? Does the controller behave strangely when the selectionTime of a movie is -1 or when the duration of the movie is 0?*

A The behavior you're observing is to be expected if the controller is attached to the movie. In this case, the controller goes to wherever the bottom left corner of the movie box takes it. If the movie loses all its "visible" parts, the movie controller will jump to the top of the window. The only way to get around this is to detach the controller when the movie box is empty; this is also something to keep in mind for the cases when the movie contains only sound, since pure sound movies have no dimensions. You can find sample code showing how to do

Kudos to our readers who care enough to ask us terrific and well thought-out questions. The answers are supplied by our teams of technical gurus; our thanks to all. Special thanks to Pete ("Luke") Alexander, Tim Dierks, Steve Falkenburg, Bill Guschwan, C. K. Haun, Dave Hersey, Dennis Hescox, Rich Kubota, Edgar Lee, Jim Luther, Joseph Maurer, Kevin Mellander, Jim Mensch, Guillermo Ortiz, Craig Prouse, Dave Radcliffe, Greg Robbins, Kent Sandvik, Gordon Sheridan, Bryan ("Stearno") Stearns, Brigham Stevens, Sriram Subramaniam, Forrest Tanaka, John Wang, and Scott ("Zz") Zimmerman for the material in this Q & A column. Thanks also to developer Bruce Ballard for his graphics sample.•

this on the *Developer CD Series* disc, in the SimpleInMovies example that accompanies the QuickTime article in *develop* Issue 7.

Q *Stepping through QuickTime movie video frames in the order they appear in the movie is simple using GetMovieNextInterestingTime, except for getting the first frame. If I set the time to 0 and rate to 1, I get the second frame, not the first. In addition, the video may start later than at 0. How do you suggest finding this first frame of video?*

A To get the first frame under the conditions you describe, you have to pass the flag nextTimeEdgeOK = $2000 to GetMovieNextInterestingTime. What this flag does is make the call return the current interesting time instead of the next, if the current time is an interesting time. You need to do this because there's no way to go negative and then ask for the next interesting time.

Q *I save PICTs to my document's data fork by writing the contents of the PicHandle. To save movies, do I convert the movie to a handle, and then save that as I would with PICTs? I just want the file references, not the data itself.*

A To save movies that are suitable for storage in a file, use PutMovieIntoHandle. The result of this call can be saved in the data fork of your files, and then you can call NewMovieFromHandle to reconstruct the movie for playback or editing.

You should also read the documentation regarding the Movie Toolbox FlattenMovie procedure, which creates a file that contains the 'moov' resource and the data all in the data fork. The advantage here is that the movie file you create using FlattenMovie can be read by any other QuickTime-capable application.

Q *How can I identify the sender of an Apple event?*

A If your application is just sending a reply, it should not be creating an Apple event or calling AESend. Instead, the Apple event handler should stuff the response information into the reply event, as shown on page 6-50 of *Inside Macintosh* Volume VI. The Apple Event Manager takes care of addressing and sending the event.

To find the target ID or process serial number of the sender of an Apple event, use AEGetAttributePtr to extract the address attribute, as follows:

```
retCode := AEGetAttributePtr(myAppleEvent, keyAddressAttr,
                             typeWildCard, senderType, @senderBuffer,
                             sizeof(senderBuffer), senderSize)
```

Looking for the Apple II Q & A section?
It's gone. See the Editorial for details.•

The senderBuffer can later be used with AECreateDesc to create an address to be passed to AESend. The buffer should be at least as large as data type TargetID. See *Inside Macintosh* Volume VI, page 5-22, for a description of TargetID.

Q *When I resize my real-time animation window in System 6, I call UpdateGWorld with the new size, and after that any drawing into the GWorld has no effect. This same code works perfectly in System 7. What could cause this?*

A You probably can't draw anything into your GWorld after using UpdateGWorld to resize it because of the clipping region of your GWorld. In system software versions before 7.0, UpdateGWorld always resizes the GWorld's clipping region proportional to the amount that the GWorld itself is resized. Unfortunately, NewGWorld initializes the clipping region of the GWorld to the entire QuickDraw coordinate plane, [T:-32767 L:-32767 B:32767 R:32767]. If UpdateGWorld resizes any of these coordinates so that they fall outside this range, the coordinates wrap around to the other end of the signed integer space, and that makes the clipping region empty. Empty clipping regions stop any drawing from happening.

The change in System 7 is that UpdateGWorld explicitly checks for the clipping region [T:-32767 L:-32767 B:32767 R:32767]. If it finds this, it doesn't resize the clipping region. Otherwise, UpdateGWorld acts the same way that it did before System 7.

One of our mottos is, "Never give QuickDraw a chance to do the wrong thing." In keeping with that, we always explicitly set the clipping region of a GWorld whenever we change the size of the GWorld. So after calling NewGWorld, set its clipping region to be coincident with its portRect. After calling UpdateGWorld to resize the GWorld, set its clipping region to be coincident with its new portRect. That way, you'll always have a known environment and you won't have to worry about the change that was made in System 7—and you'll be less susceptible to bugs in this area in the future.

Q *UpdateGWorld doesn't seem to respond to the ditherPix flag unless color depth changes. The return flag after changing my color table is 0x10000, indicating that color mapping happened but not dithering. Is this a bug?*

A Yes, this is a bug. UpdateGWorld ignores dithering if no depth change is made. It probably won't be changed in the near future. The workaround is as follows:

1. Create a new pixMap with the new color table.
2. Call CopyBits to transfer your image to the newly created pixMap with dithering from the original GWorld's pixMap.

3. Update the GWorld with the new color table without using ditherPix.
4. Use CopyBits from the newly created pixMap without dithering back to the GWorld.

This will give you the same effect as UpdateGWorld with ditherPix.

Q *Can I create, open, write, and close a file completely at interrupt time? I need to be compatible with both System 6 and System 7.*

A All these operations (and more) can be done completely at interrupt time. Any call that can be made asynchronously can be safely made at interrupt time, provided it's made asynchronously. Glancing through *Inside Macintosh* Volume IV, we can see that this includes just about all of the File Manager, except for the calls to mount and unmount volumes, which must be made at a time when it is safe to move or purge memory.

One caveat: Making a call asynchronously here means *really* making it asynchronously; making the call and then sitting in a little loop waiting for the ioResult field to change does not qualify. Either you must use completion routines to determine when a call has completed, or you must check the ioResult from time to time, never waiting for it at interrupt time (and in this case, a deferred task does qualify as being at interrupt time).

Q *How can I tell whether a window is a Balloon Help window?*

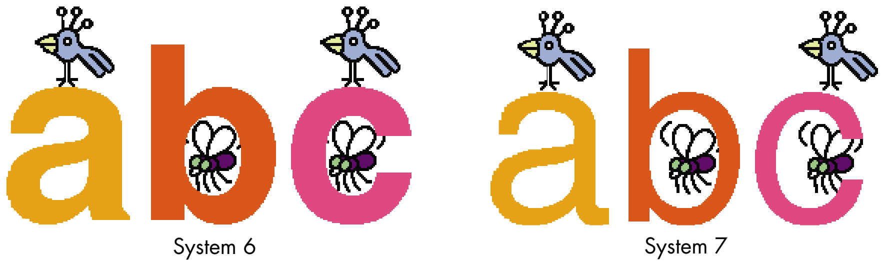
A First, call the Help Manager procedure HMIsBalloon to determine whether a balloon is being displayed at all. Then call HMGetBalloonWindow to get the help window's window pointer, and compare that to the window pointer of the window you've got.

Note that if HMIsBalloon returns TRUE and HMGetBalloonWindow returns a window pointer of NIL, it means that the balloon "window" that's displayed really isn't a window at all; this will happen, for instance, if the balloon is being displayed on top of a pulled-down menu (we call this "to boldly go where no window has gone before").

Q *How can I tell whether a font is monospaced or proportional? The FontRec record's fontType field doesn't correctly tell me whether the font is fixed width as Inside Macintosh Volume V says it should. All system fonts appear to have the same fontType regardless of whether they're fixed or proportional. Currently I test whether the width of the characters "m" and "i" are equal and if they are, I consider the font to be fixed width. Is there an easier (and faster!) way?*

A The Font Manager documentation is not explicit enough about the fact that bit 13 (0x2000) of the fontType field is useless. The Font Manager doesn't check the setting of this bit, nor does QuickDraw (or any printer driver). As you observed, monospaced fonts like Monaco or Courier don't have the bit set; the bit is meaningless. In addition, the fontType field is available only for 'FONT' and 'NFNT' resources; it does not exist in 'sfnt' resources, and you would have to check separately for the resource type of the font. Your idea of comparing the widths of "m" and "i" (or any other characters that are extremely unlikely to have the same widths in a proportionally spaced font) is indeed the only reasonable way of figuring out whether a font is monospaced.

Q *The TrueType system extension (INIT) apparently renders glyphs differently with System 6 than with System 7. For example, our "abc" string in 160-point Helvetica® is almost half as many pixels under System 7, so the styled text no longer lines up with the bitmapped graphics underneath. Any way to avoid this?*



A Your System 6 configuration probably has the specific Helvetica Bold TrueType outlines available, while this Helvetica Bold TrueType version is missing in your System 7. When the Font Manager gets a request for Helvetica, txSize 160, txFace bold, it looks in the font association table of the Helvetica FOND (font family record; see page 37 of *Inside Macintosh* Volume IV). First, it looks for the right size (yes, there's a TrueType outline font: size requirement fulfilled), then it looks for the style (oops, no Bold variant of the font available; must ask QuickDraw to apply its algorithmic "smearing" to produce a bold version of it).

Unfortunately, the QuickDraw emboldening always works the same way, regardless of the size of the character: it just smears the character horizontally by one pixel—which is rather ineffective for big point sizes and, of course, quite different from the typographically truly bold outline of the Helvetica Bold font.

By the way, if you choose the stylistic variants outline or shadow, the result is equally disappointing, because there are no specific TrueType versions available for Helvetica Outline or Helvetica Shadow.

In conclusion, the only way to avoid this problem is to make sure your users have the required font versions in their system. You may want to include this as a recommendation in the manual, or even to come up with an alert in your application if there's no Helvetica Bold in the system. Unfortunately, there's no easy, built-in way to check for this; IsOutline returns TRUE even when there's no Helvetica Bold, because the Helvetica TrueType font is used to render the character in the first place; the QuickDraw smearing is applied in a second step, and is not considered for the result of IsOutline. You would have to take the Helvetica FOND and walk its font association table "by hand."

Q *My application calls SetOutlinePreferred so that TrueType fonts are used if both bitmapped and TrueType fonts are in the system. It was reported to me, however, that some international TrueType fonts look really bad at small point sizes on the screen. Should I avoid calling this function?*

A SetOutlinePreferred is best used as a user-selectable option. Along the same lines, you might want to include the SetPreserveGlyph call (*Inside Macintosh* Volume VI, page 12-21)—again, as a user-selectable option.

Currently, the default for outlinePreferred is FALSE for compatibility reasons (existing documents don't get reflowed if the bitmapped fonts are still around) and for aesthetic and performance reasons (users are free to maintain bitmapped fonts in the smaller point sizes if the TrueType version isn't satisfying for small sizes or is too slow). On the other hand, as soon as a bitmapped font is *unavailable* for a requested point size, and a TrueType font is present, the TrueType font is used even with outlinePreferred = FALSE. Setting outlinePreferred = TRUE makes a difference only for point sizes where a bitmapped font strike is present along with an 'sfnt' in the same family.

TrueType fonts might be preferable even for small point sizes if linearly scaled character widths are more important than screen rendering: if the main purpose of a program is preprint processing for a high-resolution output device, outlinePreferred = TRUE may give better line layout results on the printer, at the price of "not so great" type rendering on a 72 dpi screen. (An example of the conflict between linearly scaled TrueType and nonlinearly scaled bitmapped fonts is Helvetica: StringWidth('Lilli') returns 19 for the 12-point bitmapped font, and 15 for the 13-point size from TrueType!)

All this boils down to the recommendation stated initially: the user should be given the flexibility to decide whether to use the existing bitmaps (using TrueType only for bigger point sizes and high-resolution printers), or to go with TrueType even if the result on the screen is not optimal. (By the way, it's likely that TrueType development will substantially reduce this conflict in the future.)

Q *When you bring up the Finder windows under System 7 on a color system and click a control panel item icon, it paints itself that fancy gray. How can I get that effect?*

A To get the fancy System 7 icon dimming to work in your program, read Macintosh Technical Note #306, "Drawing Icons the System 7 Way," and use the icon-drawing routines contained in it. The routines show how to use the Icon Toolkit, which is what the Finder uses. If you want the same effect under System 6, you'll have to emulate the dimming of the icons via QuickDraw; the IconDimming sample code in the Snippets folder on the *Developer CD Series* disc shows how to do this.

Q *When the OK button is disabled in the System 7 Standard File dialog box, it's drawn in gray. I was looking for sample code on how to do this in a way that's appropriate for multiple screens at various color depths. For example, how should you draw the outline if you have an OK button in a movable modal dialog box with half the OK button on an 8-bit color screen and the other half on a 1-bit monochrome screen?*

A There are two ways to draw the gray (dimmed) outline across several screens in different depths: one uses MakeRGBPat *(Inside Macintosh* Volume V, page 73), the other uses DeviceLoop (*Inside Macintosh* Volume VI, page 21-23). Look for GrayishOutline.p in the Snippets folder on the *Developer CD Series* disc for a code sample that demonstrates both ways.

Q *If the Epcot Center building "Spaceship Earth" were a golf ball and you were proportionally tall enough to hit it, where would it land?*

A Zimbabwe.

Q *How do you determine whether the Picture Utilities Package function GetPictInfo is available? Gestalt doesn't seem to have the right stuff!*

A To determine whether the GetPictInfo routine is available, check the system version number with the Gestalt function. GetPictInfo is available in system software version 7.0 and later. Use the Gestalt selector gestaltSystemVersion to determine the version of the system currently running. Usually it's best not to rely on the system version to determine whether features are available, but in this case, it's the only way to determine whether the Picture Utilities Package is available.

For example, the following C function will determine whether the GetPictInfo call is available:

```
#include <GestaltEQU.h>
Boolean IsGetPictInfoAvail()
{
  OSErr err;
  long  feature;
  err = Gestalt(gestaltSystemVersion,&feature);
  /* Check for System 7 and later */
  return (feature >= 0x00000700);
}
```

In *Inside Macintosh* Volume VI, see page 3-42 for information on using Gestalt to check the system version number, and see page 18-3 for information on the Picture Utilities Package.

Q *How can I directly access the alpha channel (the unused 8 bits in a 32-bit direct pixel using QuickDraw) under System 7? Under System 6 it was easy, but under System 7's CopyBits the alpha channel works with srcXor but not with srcCopy.*

A With the System 7 QuickDraw rewrite, all "accidental" support for the unused byte was removed, because QuickDraw isn't supposed to operate on the unused byte of each pixel. QuickDraw has never officially supported use of the extra byte for such purposes as an alpha channel. As stated in *Inside Macintosh* Volume VI, page 17-5, "8 bits in the pixel are not part of any component. These bits are unused: Color QuickDraw sets them to 0 in any image it creates. If presented with a 32-bit image—for example, in the CopyBits procedure—it passes whatever bits are there."

Therefore, you cannot rely on any QuickDraw procedure to preserve the contents of the unused byte, which in your case is the alpha channel. In fact, even CopyBits may alter the byte, if stretching or dithering is involved in the CopyBits, by setting it to 0. Your alternatives are not to use the unused byte for alpha channel storage since the integrity of the data cannot be guaranteed, or not to use QuickDraw drawing routines that can alter the unused byte.

Q *When used from MPW C++,* ***pragma unused, pragma force_active,*** *and* ***pragma once*** *don't appear to work. In fact,* ***pragma unused*** *actually causes a C compile-time error. Why does this occur in spite of assurances in release notes that all pragmas are passed on to the C compiler?*

A The problem with pragmas and C++ is that the CFront compiler generates C code, and during this phase it also shuffles around the source code lines, so the pragma doesn't end up in the same place as originally intended. Also, CFront moves any pragmas inside the function body outside, because it can't do much with the pragmas, and the best bet is to move them just outside the body for the

C compiler. This means that any pragmas stated inside the function body are unusable in real life.

Here's a summary of how pragmas work with C++:

- **pragma segment**, **pragma parameter**, and **pragma processor** should work OK.
- **pragma force_active** may or may not work, depending on the code case.
- **pragma warnings** and **pragma pop/push** should work in most cases, depending on the code movement.
- **pragma trace** should also work, especially if it's defined just before a function or member function.
- **pragma unused** and **pragma once** won't work, alas.

For more information about pragmas and C++, please consult the MPW 3.2 C++ documentation.

Q *Inside Macintosh Volume II, page 33, states that _GetHandleSize returns D0.L >= 0 if the trap is successful or D0.W < 0 if the trap is unsuccessful. What happens if the handle size is 0xFFFF, for instance? A TST.W will indicate an error when in fact there is none. How should I check for this condition?*

A *Inside Macintosh* is correct (although confusing) regarding the determination of an error condition. The way to do it is first test the long to see if it's valid (D0 >= 0). If the long is valid, you can continue with confidence that no error occurred. If, however, the long in D0 is negative, the low word contains the error (and currently the high word contains $FFFF, the sign extension). The reason the manual highlights the fact that only the low word contains the error is to allow you to save the error in standard fashion since all other errors are word sized, and also to caution you against using the processor status on exit from GetHandleSize since it will be based on the low word only. In other words, if the long is negative, simply ignore the high word. Here's some assembly code that will work:

```
        move.L   theHandle(a6),A0
        _GetHandleSize
        tst.L    D0
        bpl.s    @valueOK
        move.W   D0,theError(A5)
        moveQ    #0,D0
@valueOK
```

Q *What are recommended values for retry interval and retry count when using the AppleTalk NBP call PLookupName on a complicated internet?*

A You might want to start with the NBP retry interval and retry count values Apple uses for its Chooser PRER and RDEV device resource files. The Chooser grabs these values from the PRER's or RDEV's GNRL resource -4096:

Device	Interval	Count
LaserWriter	$0B	$05
AppleTalk ImageWriter	$07	$02
AppleShare	$07	$05
If no GNRL resource	$0F	$03

Apple's engineering teams found these values to work well in most situations.

The count value should be based on how likely it is for the device to miss NBP lookup requests. For example, the AppleTalk ImageWriter has a dedicated processor on the LocalTalk option card just to handle AppleTalk, so its count value is low; most Macintosh models and LaserWriter printers depend on their 680x0 processor to handle AppleTalk along with everything else in the system (the Macintosh IIfx and Macintosh Quadra models are exceptions to this), so their count value is higher.

The interval value should be based on the speed of the network and how many devices of this type you expect there to be on the network. On a network with very slow connections (for example, one using a modem bridge), or in cases where there are so many devices of a particular type that lots of collisions occur during lookups, the interval value should be increased.

Apple puts these values in a resource because not all networks and devices are alike. You should do the same (put your interval and count in a resource so that it can be configured).

Q *I'd like to use the same names that the system uses to identify itself on the AppleTalk network in my program. Where can I find those names?*

A The names used by the system for network services are stored in two 'STR ' resources in the System file. Your program can retrieve those names with the Resource Manager's GetString function.

Only one of the names is available in systems before System 7: the name set by the Chooser desk accessory. That name is stored in 'STR ' resource ID -16096. With System 7, the Sharing Setup control panel lets the user assign two names for network services: the Owner name and the Computer name.

The Owner name is the name stored in 'STR ' resource ID -16096; it identifies the user of the Macintosh. The Owner name is used by System 7 for two primary purposes: to identify the owner of the system when accessing the system remotely through System 7 file sharing or through the user identity dialog used by the PPC Toolbox (and Apple Event Manager), and to serve as the default user name when logging on to other file servers with the Chooser.

The Computer name (also known as the Flagship name) is the name stored in 'STR ' resource ID -16413; it identifies the Macintosh. The Computer name is the name used by system network services to identify themselves on the AppleTalk network. For example, if your system's Computer name is "PizzaBox," the PPC Toolbox registers the name "PizzaBox:PPCToolBox@*" when you start program linking, and file sharing registers the name "PizzaBox:AFPServer@*" when you start file sharing.

Q *What's the recommended technique for telling whether the user has turned off AppleTalk?*

A The best way to determine whether AppleTalk has been turned off is to use the AppleTalk Transition Queue to alert you to .MPP closures. (This is one of the reasons why the AppleTalk Transition Queue was implemented.) The AppleTalk Transition Queue is available only in AppleTalk version 53 or later, and is documented in the AppleTalk chapter of *Inside Macintosh* Volume VI, starting on page 32-17. There's also a code snippet, Transition Queue, in the Snippets folder on the *Developer CD Series* disc.

Q *Sometimes when my system extension (INIT) starts executing, the current zone is the system zone rather than the application zone. Should I call SetZone(ApplicZone) before allocating memory in the system extension?*

A The system does not set the zone to the application zone before loading each system extension, so if a previous extension left the zone set to the system zone, it's possible that an extension could unintentionally be loaded into the system heap and have the current zone be the system zone.

To ensure that nonpermanent memory requested by a system extension is allocated in the application heap, do a SetZone(ApplicZone) before calling NewHandle or NewPtr. Any system extension that calls SetZone should restore the current zone to what it was upon entry.

Any permanent memory allocation by a system extension should be made in the system heap with NewPtrSys or NewHandleSys. Use a 'sysz' resource if the system heap allocations will exceed 16K.

Q *Are there any new rules regarding SCSI driving with virtual memory? My System 6 driver doesn't work with System 7.*

A It's important to remember that VM usually uses a SCSI device for its backing store. As such, if VM needs to use your driver it can't tolerate a driver's page swap in the middle of a page swap. This means if your driver's code is not in the system heap, it needs to be held when called, and your buffers also need to be held if your driver is entered by a Control or Status call. Buffers are automatically held by the system if your driver is entered by a Read or Write call. The following documents provide a good overview of what you need to do to revise a SCSI driver for VM compatibility.

- *Inside Macintosh* Volume VI, which contains new information specific to virtual memory as it relates to drivers and especially SCSI
- Macintosh Technical Note #285, "Coping with VM and Memory Mappings"
- "VM Paper" from the System 7 CD in the VM Goodies folder

Q *I discovered an interesting bug in the Macintosh LaserWriter driver. If the word "timeout" is in the name of a document, the LaserWriter driver will give a timeout error -8132. Are there similar magic words?*

A PostScript error messages are sent from the LaserWriter to the driver as text streams. The driver must check these strings to see if they contain an error message. If a document is named something that contains the same string as a PostScript error message, the driver may think there's an error when the printer sends the "status: printing document XXXXX" message. Other strings cause similar problems; one of them is "printer out of paper." If you want to see the rest of the strings, take a look at the LaserWriter printer driver resource type 'PREC' ID = 109.

Q *What do the terms "maney" and "fakey" mean?*

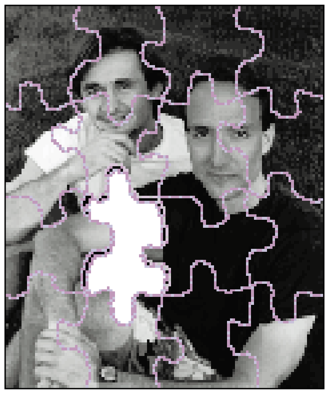
A These are slang words commonly used in California. "Fakey" means you're riding your snowboard backwards. "Maney," often applied to snowboarding, is derived from "maniac"; it means intense, high-energy, absorbing maximum consciousness. It also has allusions to the mane of a lion, as in the pride of the lion. Nietzsche might have equated "maney" with "will to power."

Have more questions? Need more answers? Take a look at the Dev Tech Answers library on AppleLink (updated weekly) or at the Q & A stack on the *Developer CD Series* disc.•

KON & BAL'S PUZZLE PAGE

SLEEPING BEAUTY

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. These problems are supposed to be tough. If you don't get a high score, at least you'll learn interesting Macintosh trivia.



KONSTANTIN OTHMER AND BRUCE LEAK

During the development of QuickTime, a number of interesting bugs reared their ugly heads. Try to figure out this one before KON does.

120 BAL Here's the problem: When you play a movie on a Macintosh IIfx the machine hangs after about six hours. If you turn off the sound, or try it on any Macintosh other than an fx, it doesn't hang.

KON How does it hang? Is the Macintosh locked up?

115 BAL Well, the movies aren't playing. They just all freeze about halfway through. Menus still work, though. You can even switch to the Finder and click between windows and move icons around, but when you launch an application or open a folder, the Finder draws one of the zoom rectangles and then hangs.

KON Like a time bomb. Can you get into MacsBug?

110 BAL Yeah.

KON Whew. For a second I thought this was going to be really tough. You've got MacsBug, so what's the problem?

BAL I don't know, you tell me.

KON How do movies get time? SystemTask or something?

105 BAL No, you have to call MoviesTask.

KON Figures. So I set a break on MoviesTask; is it getting called?

KONSTANTIN OTHMER AND BRUCE LEAK have been puzzling about reality, life, the universe, and even computers for a long time. Since the great success of their Graphics '90 World Tour, which included peace-keeping, hostage-freeing, and wall-smashing, they settled down, shipped a few QuickDraw packages, and cleaned out their closets. Then came the coup: da division of da Union. Konstantin got QuickDraw, 200 rubles, and a guaranteed spot at the front of the bread line. Official party line on Bruce: "vacationing in the Crimea." Bruce was actually working on the first QuickDraw spinoff. To provide a seamless upgrade path, and to leverage off of brand awareness, he decided to call this project QuickTime.•

100 BAL Nope.

KON The application is supposed to call it?

95 BAL Yep.

KON So is WaitNextEvent getting called?

90 BAL Nope.

KON But menus work???

85 BAL Yep.

KON WaitNextEvent is being called with a bogus sleep time. I set a breakpoint on it, pull down a menu, and see what the sleep time parameter is.

80 BAL Now you're thinking, Kon! But no. The sleep time is 1, just as it's supposed to be.

KON Hmmm. So I trace 50 times and see where I'm spinning.

75 BAL There's a bunch of stuff going on; it's not just some simple loop.

KON So I record A-traps.

70 BAL There are three traps getting called: ABF7 from inside MF, A0DD (PPCToolbox) from a big block in the system heap that's not QuickTime, and A030 (EventAvail) from a 'scod' resource that's different from the block calling ABF7.

KON There are no null events coming through to the application, so MoviesTask never gets called. The sleep time must never be expiring. Are ticks running?

BAL How do you figure that out?

KON I DM ticks, continue executing, and then DM ticks again.

65 BAL Ticks doesn't change.

KON It's updated by some hardware mechanism at interrupt time, right?

BAL Sure, a hardware *mechanism*. Is that what they teach you at Caltech?

KON OK, OK. There's a heartbeat task that generates a level-1 interrupt that updates ticks, right?

60 BAL Yeah.

KON So is the level-1 interrupt happening?

BAL How do you check that?

KON I know from reading this cool *develop* column that the level-1 interrupt vector is at location $64, so I set a break there and see if it's firing.

55 BAL Wait a second. That's the way ticks are updated on every Macintosh except the fx. This problem happens only on an fx.

KON Why is the fx different?

50 BAL The guy that designed the hardware assumed that the heartbeat task—and thus ticks—was supposed to happen every 60th of a second. Unfortunately it's supposed to be every 60.14th of a second or something, so Gary Davidian fixed it by installing a Time Manager task that updates ticks. The extended Time Manager, found in System 6.0.4 and later, adds a drift-free mode for Time Manager callbacks. This allows for accurate scheduling of periodic events without long-term drift. With the extended Time Manager, the next callback is scheduled with respect to when the current callback should have fired, rather than the current time. So as of System 6.0.4 there are two types of Time Manager tasks: the regular ones and the drift-free ones. Ticks are updated via a drift-free task, so they advance accurately over time.

KON Hmmm. So is the Time Manager task that updates ticks getting called?

45 BAL Obviously not; ticks aren't changing.

KON How are the tasks in the queue organized?

40 BAL They're kept in the order that they fire in.

KON So where is the ticks task in the queue?

35 BAL Well, it's not the first one, and the first one is scheduled for sometime tomorrow.

KON Fine. I leave and come back tomorrow. Does my movie start playing?

30 BAL It could be. But we're shipping before then.

KON So the first element is getting messed up and never completes. Then none of the other items in the queue get executed because they're all deltas off the first element and the movies hang.

25 BAL Now we're getting somewhere.

KON So how is that first element getting confused?

BAL Whose problem is this?

KON OK. Who owns the first item?

BAL How do you figure that out?

KON I do an IL to see what traps it's calling. I see where it is in the heap. I set a breakpoint on it, force it to fire, and see what it does.

20 BAL You're knee deep in spaghetti. You could probably figure it out this way but it's pretty nasty.

SCORING

100–120	Members of the QuickTime team and their immediate family aren't eligible.
75–95	Scores count only on the first reading!
50–70	Not bad—buy yourself an ice cream.
25–45	The next one will be easier.
5–20	Stick to word searches.•

KON OK. I break on InsTime to see who installs it. I break on PrimeTime to see who starts it up. I figure out whether it's using InsTime or InsXTime.

15 BAL The PrimeTime comes from an 'snth' resource. It was installed with InsXTime.

KON Aha! The Sound Manager. That's why it works when the sound is turned off.

BAL They don't pay you enough, Kon.

KON So how does this element get updated in the queue? Don't Time Manager tasks call PrimeTime to reschedule themselves?

10 BAL Yeah.

KON So is someone calling PrimeTime on the Sound Manager task with a bogus value?

5 BAL Not really. It always calls PrimeTime with a value of 0, indicating that it wants to be called right away. The Sound Manager does this since it's not reentrant. By scheduling a task, it knows it won't be called until it's finished servicing the current interrupt, avoiding reentrancy problems.

KON I'm sure everyone that's reading this has figured it out by now. I know I have.

BAL You're bluffing again, Kon.

KON It's easy. Since the drift-free Time Manager schedules tasks based on when they should occur, and since this sound task is using a count of 0 when calling PrimeTime, its backlog gets bigger and bigger. So each PrimeTime call schedules an event that should have occurred further and further in the past. The task is executed immediately, of course, but the backlog builds. It always puts this element at the head of the queue, but eventually it overflows the long that contains the backlog, and the scheduling time becomes incredibly large. There are about 4 billion microseconds in a long, which is about an hour. The Time Manager counts in units that are about 20 microseconds, so it would take about 20 hours to overflow. Take away a bit for signed math and another because it's calculating scheduling times, and I would guess it should hang about once every 5 hours. It must have taken forever to find that one.

BAL Yeah, it was a drag. We fixed it before we shipped QuickTime, of course. But it's just the kind of thing that makes scheduling software projects so hard.

KON Nasty.

BAL Yeah.

Thanks to Gary Davidian and Jean-Charles Mourey for reviewing this column.•

INDEX

For a cumulative index to all issues of *develop* and a complete source code listing, see the *Developer CD Series* disc.•