develop

The Apple Technical Journal

Issue 9 Winter 1992

Apple Computer, Inc.

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Technical Buckstopper *Dave Johnson*

Review Board *Pete "Luke" Alexander*, *Chris Derossi*, *C. K. Haun*, *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*, *Leslie Steere*, *Carol Westberg*

Indexer *Ira Kleinberg*

Manager, Developer Support Systems and Communications *David Krathwohl*

ART & PRODUCTION

Production Manager *Hartley Lesser*

Art Director *Diane Wilcox*

Technical Illustration *Geoff McCormack*, *John Ryan*

Formatting *Forbes Mill Press*

Printing *Wolfer Printing Company, Inc.*

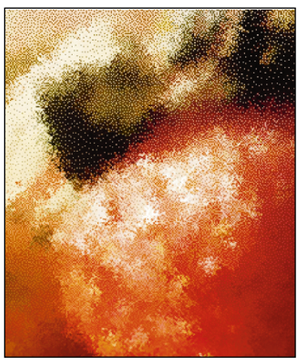
Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

Photography *Sharon Beals*, *Dennis Hescox*, *Kathleen Siemont*

Circulation Management *David Wilson*

Online Production *Cassi Carpenter*



To create this cover, Hal Rucker and Cleo Huggins bought the nicest-looking fruit they could find, photographed it and scanned in a slide, manipulated the scan with Adobe Photoshop, and blended in a dithered version of it. Delicious!

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

The *Developer CD Series* disc for February 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.

CAROLINE ROSE

Dear Readers,

Let's talk about *develop*: what it is, what it might be, what it can do for you, and what you can do for it. This journal exists to meet your needs, so I hope you'll help us out by reading on and giving us your two cents (if not your articles).

Originally, *develop* was thought of as "heavily commented code": along with the accompanying CD, it was meant as a vehicle for providing well-explained code that you, the developer, could plug into your application with the confidence that it would be compatible with future system software. To ensure compatibility, articles and code were written primarily by Apple engineers and heavily reviewed by other engineers at Apple.

But other types of articles have been submitted, and some have made it into print. Most notable was the ground-breaking Threads article in Issue 6, the first article for which source code was not provided. This lack of source code did not go unnoticed by our readers, yet the overall response to the Threads package was extremely favorable. So we've moved from *always* providing source code to providing it *if at all possible*. We still make every effort, however, to give you something that won't break in future systems.

Recently we've had some requests to publish articles that describe algorithms or ideas, not code. Our current feeling is that as long as an article can help you create good Apple products, we'll consider publishing it. Please let us know what you'd like to see. There are some Apple engineers who are willing to contribute to *develop* but would like to know just what developers want to see. We get a lot of input from Developer Technical Support about what you seem to need the most help with—but let us at *develop* know directly, and we can try to make it happen faster.

Regarding who writes the articles: we feel that as long as the code is reviewed by Apple engineers, there's no need to rely solely on people at Apple for contributions. We'd like to encourage all of you to think about what you'd like to share with your fellow developers—something that would help them and also give you a way to showcase and release your code in a way that wouldn't otherwise be possible. We offer something those *other* journals don't: not only review by Apple engineers and the assurance of future compatibility, but also an editorial process that will make your prose shine so brilliantly you'll need to wear shades. We'll assign an editor who will

CAROLINE ROSE (AppleLink: CROSE) has been writing computer documentation ever since Steve Jobs was barely a teen. When his company moved in down the block from where she worked as a writer and then a programmer, Caroline took no notice—until they asked if she wanted to write what even then was known as *Inside Macintosh*. Around the time she completed that three-volume tome, Steve left Apple to form NeXT, and Caroline signed on to launch NeXT's Publications group. A year ago she returned to Apple to take on the fun-filled job of being *develop*'s editor in chief. For fun outside of work, Caroline dances up a storm, listens to music, plays with her cat and other friends, treks through the wilderness (in boots or on skis), swims like a maniac, reads fiction (not sci-fi!), studies Italian, does Tai Chi, and never stops exploring new ways to have fun.

help turn your raw material into a polished piece—or tread lightly on it if that's all you need. We'll give your article that professional look and feel without killing the humor. So, if you're willing, please send me your ideas or outlines, and we'll take it from there.

Back to the subject of your opinions about *develop*: Many of you who are Apple Associates and Partners have by now been formally surveyed on how you rate various support-related materials, of which *develop* is only one shining example. We'd also like to hear from the rest of you, however informally. I can't overemphasize how important your opinions are and how much they'll affect *develop*'s future. So please, express yourself! Tell us what's good or bad about this journal's content, format, delivery, or anything else. We're all ears.

Issue 8 ended with this trivia question: What word was used instead of "click" to describe the action of pressing a button on that first mouse? The answer, which none of you have gotten as of this writing, is "bug." Maybe you'll do better on this next one: The original hardcover *Inside Macintosh* Volumes I-III had a running pattern of Macintosh computers across its endpapers (those heavy sheets at the very beginning and end of hardcover books). What broke this pattern, and why?

Caroline Rose
Editor

SUBSCRIPTION INFORMATION

Use the order form on the last page of this issue to subscribe to *develop*. 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 reverse of the order form on the last page of this issue. Back issues are also on the *Developer CD Series* disc.•

LETTERS

CURLING UP WITH DEVELOP

Regarding your editorial in Issue 8: I agree with you on liking to have a "hard copy" to be able to curl up with when trying to understand something for the first time. I can always go to the computer and try examples or ideas. But to lay back and put up my feet or nestle under a quilt in bed is more relaxing to let concepts sink in and develop on their own, to spring forth with clarity later.

I like the idea of sending the disc separately in its own case—though I never had a mangled disc problem. It is the magazine itself that has a rougher trip. I received Issue 8 without the disc and hope that the disc is not far behind. The mailing label on the back cover was half off, not torn, but detached. Flapping in the breeze, so to speak.

Keep up the good work on the magazine. I look forward to it each time.

—Robert Redmond

Thanks for your letter. It's not only heartening to hear from developers who agree with me on this, but it makes a difference. Your opinions do count.

The disc is now in a separate case, but it's not mailed under a separate label. They should have arrived wrapped cozily together. We'll send you the disc right away. Sorry about that.

—Caroline Rose

TEXT FORMATS GALORE

There's a problem with the Macintosh Technical Notes which I'm finding with increasing frequency in Apple's electronic publications. The only word processor I use is Nisus, with which I can read MS Word 3.0 and 4.0 files without buying a Microsoft product. This may be an unreasonable prejudice, but I bet it isn't uncommon.

But Nisus can't decipher fast-saved Word files. This means, I suspect, that the entire set of new Macintosh technical publications is unavailable to me. Worse, I fear that the next Developer CD is going to have lots of files with new, valuable, and (for me) hidden information.

I know Apple is serious about electronic distribution of technical documents. I'm sure fast-saving in Word is a great convenience to the authors, but surely using a format not widely readable defeats the purpose of the exercise. I don't object to standardizing on Word 3.0 or 4.0, so long as that format—and not Microsoft's convenience variant—is actually used.

Could you *please* ask your authors, when providing documents for publication, to use an accessible format?

—Fritz Anderson

Thank you for alerting us to this problem. It was a snafu on our part. None of the files should have been fast-saved in Word.

We know that having text documents in Word and MacWrite® represents a bias toward these products. Unfortunately our alternatives are limited and we'll probably have to continue using these products until the spring.

The good news is that we're working on a new text formatting tool. This tool will be available on the CD and will be able to open, search, and print text documents available on the CD. The dilemma of how

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). •

to make every document available to every developer has been a topic of discussion for some time. We're hoping this will solve the problem.

Again, thank you for your input. Developer feedback is the fuel of change around here. Keep it coming.

—Sharon Flowers

NEW AND IMPROVED CD

I just received Issue 8 of *develop*, and was pleased to find that the developer's CD has improved. Is this new?

—Mike Caputo

Yes, starting with Issue 8 the CD is not just Developer Essentials, but the entire Developer CD Series disc (of which Developer Essentials is just a subset).

—Caroline Rose

SUBMITTING TO DEVELOP

First of all, I'd like to say that I'm a big fan of *develop*. The combination of excellent technical articles (with required humor) and a CD-ROM of other developer materials is unmatched. At least the flak surrounding the CD-ROM has finally calmed down in the Letters section. I've always liked the idea from the start even though I purchased a CD-ROM drive only last week.

I'm writing to find out if *develop* accepts articles from non-Apple employees. I haven't looked through the back issues to see if there were any, but none come to mind. If so, do you have a style guide for writing articles?

Keep up the great work!

—Paul-Marcel St-Onge

Thank you for your kind words about develop*; it's always a pleasure to receive mail from a big fan. Yours is the type of letter that editors in chief dream of.*

We do indeed accept articles from non-Apple employees (see this issue's Editorial). We have a vast array of materials ready for prospective authors, including an introductory document, a short submission form, a set of detailed author's guidelines, and even a Microsoft Word template for entering your article in a develop*-like format. We'll start with the intro and then send the rest as you need it.*

On the subject of the CD-ROM controversy, it may experience a revival as a result of Apple's dropping printed develop *and Tech Notes from the monthly mailing to Associates and Partners. We'd like to hear what developers think about that.*

—Caroline Rose

ART ILLEGIBLE ON-LINE

Figure 1 from Michael Gough's article on Futures (Issue 7) was illegible when printed under System 7 from a Macintosh II on a LaserWriter IINTX. It's just as illegible on the screen. Any ideas?

The HyperCard® format on the CD is convenient for flipping to pages and articles but is terrible for seeing all of a page at one time, since HyperCard's windows are not resizable. Furthermore, HyperCard is slow—especially from a CD. And searching is neither fast nor intuitive.

—Steve Tyler

I looked into the problem and found out that a mistake was made when the electronic version of Issue 7 was created: Art that's in EPS format is normally opened in an application that interprets PostScript® and then saved as a PICT. This process wasn't followed, with the result that the conversion to PICT was only an approximation, and so not very legible. This will be fixed in Issue 7 on the CD.

Regarding the HyperCard format, a lot of people agree with you. We're working on an alternate viewing mechanism—but this mechanism may not apply to develop *for a while yet. Meanwhile, HyperCard's windows are in fact resizable. If you're not able to resize them, your memory partition for HyperCard is probably not large enough; try increasing it.*

—Caroline Rose

FEELING LOST? SEE THE MACINTOSH DEVELOPMENT TOOLS ADVISOR

To a Macintosh developer starting a new project, the range of equipment available can seem daunting. To an experienced engineer suffering the constant barrage of catalogs, technical brochures, and advertisements, it can feel safest to hang on to familiar tools, whatever their shortcomings. But what's available? What systems or tools might help you get your project done? Developer University has released the Macintosh Development Tools Advisor to help answer these questions.

The Tools Advisor offers a broad array of information. A hypertext system, it tailors the data it presents to your particular interests and demands. The Advisor incorporates comprehensive technical data on over 80 programming tools—compilers and languages, debuggers and prototypers, CASE tools, and multimedia packages. It also includes essays on critical topics such as object-oriented programming, Apple events, and System 7. In preparing the Tools Advisor, Developer University collected a considerable body of catalog-style information on products available.

But a catalog is rarely sufficient. It's not enough to read lists of capabilities as recorded by manufacturers. You need to know how the tools get used in actual projects. So the Tools Advisor provides a collection of stories by programmers who use the tools it describes. These stories provide a real feel for the product. They're sometimes critical, warning of potential hazards and shortcomings of particular tools. They're also often inspiring in explaining how particular achievements were made. To help you find stories most appropriate to you, the Advisor lets you match a loose profile of your needs and wants to stories by developers with similar backgrounds and tasks.

To augment its profiles of programming tools, critical essays, and developers' stories, the Tools Advisor includes a glossary that describes exactly what technical and trade terms mean and what they imply to a development effort. Glossary entries and cross-references let you navigate the intricate terrain of technical information without losing sight of your particular interests.

Two versions of the Tools Advisor are available. The disk-based edition includes screen shots and comprehensive data on programming tools in a range of categories as well as technical details on the Macintosh and on Macintosh programming in general. The CD-ROM edition of the Tools Advisor adds demonstration versions of dozens of tools; for instance, you can take a multimedia tool for a test drive as you learn about animations and about other developers' experiences with that product.

We hope that with the Tools Advisor guiding you, you won't feel lost any more.

You can obtain a copy of the Tools Advisor through APDA. The disk-based version can also be found on the *Developer CD Series* disc. To use the Tools Advisor, you'll need a Macintosh with System 6.0.5 or later, HyperCard 2.0 or later, and a hard disk. To use the CD-ROM version, you'll of course also need a CD-ROM drive. •

MAKING THE MOST OF COLOR ON 1-BIT DEVICES

Macintosh developers faced with the dilemma of which platform to develop software for—machines with the original QuickDraw or those with Color QuickDraw—can always choose to write code that runs adequately on the lower-end machines and gives additional functionality when running on the higher-end machines. While this sounds like a simple and elegant solution, it generally requires a great deal of development and testing effort. To make this effort easier and the outcome more satisfying, we offer techniques to save color images and process them for display on 1-bit (black-and-white) devices.

KONSTANTIN OTHMER AND DANIEL LIPTON

Suppose you're writing a program that controls a 24-bit color scanner and you'd like it to work on all Macintosh computers. The problem you'll run into is that machines with the original QuickDraw (those based on the 68000 microprocessor) only have support for bitmaps, thus severely crippling the potential of your scanner. But don't despair. In our continuing quest to add Color QuickDraw functionality to machines with original QuickDraw, we've worked out techniques to save color images and process them for display, albeit in black and white, on the latter machines. We've also come up with a technique to address the problem of a laser printer's inability to resolve single pixels, which results in distorted image output. This article and the accompanying sample code (on the *Developer CD Series* disc) share these techniques with you.

SAVING COLOR IMAGES

The key to saving color images is using pictures. Recall that a picture (or PICT) in QuickDraw is a transcript of calls to routines that draw something—anything. A PICT created on one Macintosh can be displayed on any other Macintosh (provided the version of system software on the machine doing the displaying is the same as or later than the version on the machine that created the picture). For example, on a Macintosh Plus you can draw a PICT containing an 8-bit image that was created on a

KONSTANTIN OTHMER has wanted his photograph to appear in *Sports Illustrated* for as long as he can remember. Unfortunately, his college was in the NCAA's Division III, which is often overlooked by SI's editors, and somehow they've missed his virtuosity on the ski slopes at Tahoe, Vail, and Red Lodge. So Kon's had to scale down his dream, setting his sights on making the pages of *develop* instead. Here he's gotten to try on various alter egos. To come up with his latest persona, he spent a few late nights in a secret Apple lab with skilled pixel surgeon Jim Batson.•

Macintosh II. With System 7, you can even display PICTs containing 16-bit and 32-bit pixMaps on machines with original QuickDraw. (Of course, they will only be displayed as 1-bit images there.)

Creating a picture normally requires three steps:

1. Call OpenPicture to begin picture recording.
2. Perform the drawing commands you want to record.
3. Call ClosePicture to end picture recording.

The catch is that the only drawing commands that can be recorded into a picture are those available on the Macintosh on which your application is running. Thus, using this procedure on a machine with original QuickDraw provides no way to save color pixMaps into a picture, since there's no call to draw a pixMap. In other words, you can't create an 8-bit PICT on a Macintosh Plus and see it in color on a Macintosh II. But that's exactly what would make a developer's life easier—the ability to create a PICT containing deep pixMap information on a machine without Color QuickDraw. With this ability, you could capture a color image in its full glory for someone with a Color QuickDraw machine to see, while still being able to display a 1-bit version on a machine with original QuickDraw.

To get around the limitations of the normal procedure, we came up with a routine called CreatePICT2 to manually create a PICT containing color information. Your application can display the picture using DrawPicture. Now, you may be wondering whether creating your own pictures is advisable. After all, Apple frowns on developers who directly modify private data structures, and isn't that what's going on here? To ease your mind, see "But Don't I Need a License to Do This?"

The parameters to CreatePICT2 are similar to those for the QuickDraw bottleneck procedure stdBits. The difference is that CreatePICT2 returns a PicHandle and does not use a maskRgn.

The first thing the routine does is calculate a worst-case memory scenario and allocate that amount of storage. If the memory isn't available, the routine aborts, returning a NIL PicHandle. You could easily extend this routine to spool the picture to disk if the memory is not available, but that's left as an exercise for you. (*Hint*: Rather than writing out the data inline as is done here, call a function that saves a specified number of bytes in the picture. Have that routine write the data to disk. Essentially, you need an equivalent to the putPicData bottleneck.)

At this point the size of the picture is not known (since there's no way to know how well the pixMap will compress) so we simply skip the picSize field and put out the picture frame. Next is the picHeader. CreatePICT2 creates version $02FF pictures, with a header that has version $FFFF. This version of the header tells QuickDraw to ignore the header data. (OpenCPicture, available originally in 32-Bit QuickDraw

DANIEL LIPTON (a.k.a. "The PostScript Kid") is a two-and-a-half-year veteran of Apple's System Software Imaging Group, where he's working on the next generation of printing software for the Macintosh. When he's not thinking backward, he enjoys taking in a good flick, spending time with his iguana, "Iggy" (who's never quite forgiven Dan for the time she nearly froze to death in the cargo compartment of a 747), and writing zany new lyrics to classic tunes (his "Working in the Print Shop Blues" is well known to his coworkers). Most of all, Dan enjoys building and flying model airplanes, and he's recently joined the competition circuit. In fact, when asked what he'd really like to do with his life, Dan replies:

sunny { { { { hours 8 { flying } for } rather_be } dayforall } } if •

BUT DON'T I NEED A LICENSE TO DO THIS?

The reason Apple doesn't want developers modifying data structures is that it makes it hard to change them in the future. For example, early Macintosh programs locked handles by manually setting the high bit of the handle rather than calling HLock. This caused numerous compatibility problems when the 32-bit-clean Memory Manager was introduced.

So what gives? What if Apple changes OpenPicture so that it creates a totally different data format—won't the manually created pictures break?

Calm down, because the answer is no. The difference between creating your own pictures and directly modifying other data structures is that Apple can't make the current picture data format obsolete without invalidating users' data that exists on disk. Just as you can still call DrawPicture on version 1 pictures and everything works, you will always be able to call DrawPicture on existing version 2 pictures, regardless of the format of pictures created in the future.

One possible pitfall is that you might create a picture with subtle compatibility risks that draws on the existing system software but breaks at some future date. To minimize the chances of such an occurrence, you should compare the pictures you generate with those that QuickDraw generates in identical circumstances. You must be able to account for any and all differences.

Creating your own pixMaps (as our example code does) is definitely in the gray area between risky and outright disastrous behavior, and you shouldn't do it. Then why would an article written by two upstanding citizens do such a thing? The answer is that the pixMaps used by this code are kept private; they're never passed as arguments to a trap. We could just as easily have called them something else, but pixMaps work for what we're doing, so we used them. If you want to pass a pixMap to a trap, you can generate it using the NewPixMap call (not available on machines with original QuickDraw) or let other parts of Color QuickDraw, like OpenCPort, generate it.

version 1.2 and in Color QuickDraw in System 7, still creates version $02FF pictures, but the header version is now $FFFE and contains picture resolution information.)

In addition, the bounds of the clipping region of the current port are put in the picture. Without this, the default clipping region is wide open, and some versions of QuickDraw have trouble drawing pictures with wide-open clipping regions.

Next we put out an opcode—either $98 (PackBitsRect) or $9A (DirectBitsRect), depending on whether the pixMap is indexed or direct. Then the pixMap, srcRect, dstRect, and mode are put in the picture using the (are you ready for this?) PutOutPixMapSrcRectDstRectAndMode routine. Finally, either PutOutPackedDirectPixData or PutOutPackedIndexedPixData is called to put out the pixel data.

There's an important difference between indexed and direct pixMaps here. The baseAddr field is skipped when putting out indexed pixMaps and is set to $000000FF for direct pixMaps. This is done because machines without support for direct pixMaps (opcode $9A) read a word from the picture, skip that many bytes, and

continue picture parsing. When such a machine encounters the $000000FF baseAddr, the number of bytes skipped is $0000 and the next opcode is $00FF, which ends the picture playback. A graceful exit from a tough situation.

An interesting fact buried in the PutOutPixMapSrcRectDstRectAndMode routine is the value of packType. All in-memory pixMaps (that aren't in a picture) are assumed to be unpacked. Thus, you can set the packType field to specify the type of packing the pixMap should get when put in a picture. "The Low-Down on Image Compression" (*develop* Issue 6, page 43) gives details of the different pixMap compression schemes used by QuickDraw. Note that all of QuickDraw's existing packing schemes lose no image quality. QuickTime (the new INIT described in detail in the lead article in *develop* Issue 7) adds many new packing methods, most of which sacrifice some image quality to achieve much higher compression.

Anyway, these routines support only the default packing formats: 1 (or unpacked) for any pixMap with rowBytes less than 8, 0 for all other indexed pixMaps, and 4 for 32-bit direct pixMaps with rowBytes greater than 8. Note that these routines do not support 16-bit pixMaps.

Finally, the end-of-picture opcode is put out and the handle is resized to the amount actually used.

```
PicHandle CreatePICT2(PixMap *srcBits, Rect *srcRect, Rect *dstRect,
   short mode)
{
PicHandle    myPic;
short        myRowBytes;
short        *picPtr;
short        iii;
long         handleSize;

#define CLIPSIZE 12
#define PIXMAPRECSIZE 50
#define HEADERSIZE 40
#define MAXCOLORTABLESIZE 256*8+8
#define OPCODEMISCSIZE 2+8+8+2  /* opcode+srcRect+dstRect+mode */
#define ENDOFPICTSIZE 2
#define PICSIZE PIXMAPRECSIZE + HEADERSIZE + MAXCOLORTABLESIZE + \
   ENDOFPICTSIZE + OPCODEMISCSIZE + CLIPSIZE

   myRowBytes = srcBits->rowBytes & 0x3fff;
/* Allocate worst-case memory scenario using PackBits packing. */
   myPic = (PicHandle) NewHandle(PICSIZE + (long)
      ((myRowBytes/127)+2+myRowBytes)*(long)(srcBits->bounds.bottom
      - srcBits->bounds.top));
```

```
   if(!myPic)
      return(0);

/* Skip picSize and put out picFrame (10 bytes). */
   picPtr = (short *) (((long)*myPic) + 2);
   *picPtr++ = dstRect->top;
   *picPtr++ = dstRect->left;
   *picPtr++ = dstRect->bottom;
   *picPtr++ = dstRect->right;

/* Put out header (30 bytes). This could be done from a resource or
   taken from an existing picture. */
   *picPtr++ = 0x11;      /* Version opcode. */
   *picPtr++ = 0x2ff;     /* Version number. */
   *picPtr++ = 0xC00;     /* Header opcode. */
   *picPtr++ = 0xFFFF;    /* Put out PICT header version. */
   *picPtr++ = 0xFFFF;
/* The rest of the header is ignored--0 it out. */
   for(iii = 10; iii > 0; iii--)
      *picPtr++ = 0;      /* Write out 20 bytes of 0. */

/* Put out current port's clipping region. */
   *picPtr++ = 0x01;      /* Clipping opcode. */
   *picPtr++ = 0x0A;      /* Clipping region only has bounds rectangle. */
   *picPtr++ = (**thePort->clipRgn).rgnBBox.top;
   *picPtr++ = (**thePort->clipRgn).rgnBBox.left;
   *picPtr++ = (**thePort->clipRgn).rgnBBox.bottom;
   *picPtr++ = (**thePort->clipRgn).rgnBBox.right;

   HLock(myPic);
   if(srcBits->pixelType == RGBDirect)
   {         /* Must be 32-bits/pixel */
   /* Put out opcode $9A, DirectBitsRect. */
      *picPtr++ = 0x9A;
      *picPtr++ = 0;  /* BaseAddr for direct pixMaps is 0x000000FF. */
      *picPtr++ = 0xFF;
      PutOutPixMapSrcRectDstRectAndMode(srcBits, &picPtr, srcRect,
         dstRect, mode);
      if(PutOutPackedDirectPixData(srcBits, &picPtr))
         goto errorExit;    /* Nonzero indicates an error. */
   }
   else
   {
   /* Put out opcode $98, PackBitsRect. */
      *picPtr++ = 0x98;
```

```
            PutOutPixMapSrcRectDstRectAndMode(srcBits, &picPtr, srcRect,
               dstRect, mode);
            if(PutOutPackedIndexedPixData(srcBits, &picPtr))
               /* Nonzero indicates an error. */
               goto errorExit;

         }
         HUnlock(myPic);

      /* All done! Put out end-of-picture opcode, $00FF. */
         *picPtr++ = 0x00FF;

      /* Size handle down to the amount actually used. */
         handleSize = (long) picPtr - (long) *myPic;
         SetHandleSize(myPic, handleSize);
         /* Write out picture size. */
         *((short *) *myPic) = (short) handleSize;
         return(myPic);

      errorExit:
         DisposHandle(myPic);
         return(0);
      }
```

Just remember that it's not advisable to pass a pixMap you create yourself to a trap. The reason is that although it's unlikely, the format of a pixMap could change (since it's not a persistent data structure, as a picture is); this would then break your application.

The subroutines the CreatePICT2 routine calls as well as some sample code that uses CreatePICT2 are on the *Developer CD Series* disc.

PROCESSING COLOR IMAGES FOR DISPLAY

The remainder of this article focuses on processing color images for display on 1-bit (black-and-white) devices, both monitors and laser printers.

There are many techniques for representing a full-color image on a monitor when color resources are limited. The Picture Utilities Package (new in System 7) offers routines for determining optimal colors to use when displaying a pixMap in a limited color space. For example, if you want to display a 32-bit image on an 8-bit monitor, Picture Utilities can tell you the 256 best colors to use to display the image. The CreatePICT2 routine just described creates a picture that you can legally analyze using the Picture Utilities.

You can also use the techniques of thresholding and of dithering, of which there are three varieties: error diffusion, ordered, and random. Ordered dithering, also known as halftoning, is particularly useful for producing images to be printed on a laser printer. We'll examine each of these techniques in turn.

USING A 50% THRESHOLD

The first technique that leaps to mind when one is faced with displaying a color picture on a 1-bit screen is to convert each color to a luminance and then use a threshold value to determine whether or not to set the corresponding pixel. It turns out that green contributes the most to the luminance and blue contributes the least. Red, green, and blue contribute approximately 30%, 59%, and 11%, respectively, to the luminance. Thus, our formula to convert an RGB value to a luminance becomes

Luminance = (30*RED + 59*GREEN + 11*BLUE)/100

If the resulting luminance is 128 (50% of 256) or greater, the pixel is set to white; otherwise it's set to black. This technique produces the results shown in Figure 1 for gray gradations and a lovely picture of one of the authors. Note that thresholding occurs at the source pixel resolution. Thus, even though the output device used to produce Konenna is 300 dpi, the thresholded picture appears to be 72 dpi. In contrast, the techniques of error-diffusion dithering and halftoning discussed on the following pages occur at the destination device resolution.

The results shown in Figure 1 are far from ideal. The gray gradations end up as a black rectangle beside a white rectangle, and the picture of Konenna, while still cute, is completely devoid of detail.

USING ERROR-DIFFUSION DITHERING

The major problem with the threshold algorithm is that a great deal of information is thrown away. The luminance is calculated as a value between 0 and 255, but the only information we use is whether it's 128 or greater.

An easy fix is to preserve the overall image lightness by maintaining an error term and then passing the error onto neighboring pixels. Both original and Color QuickDraw have dithering algorithms built in for precisely this purpose. (Yes, it's true—while a dither flag cannot be passed explicitly to any original QuickDraw trap, a picture containing a color bit image created using dither mode on a Color QuickDraw machine will dither when drawn with original QuickDraw.) The error is calculated as

Error = Requested Intensity - Closest Available Intensity

For a black-and-white destination, the closest available intensity is either 0 (black) or 255 (white). The requested intensity is the luminance of the current pixel plus some

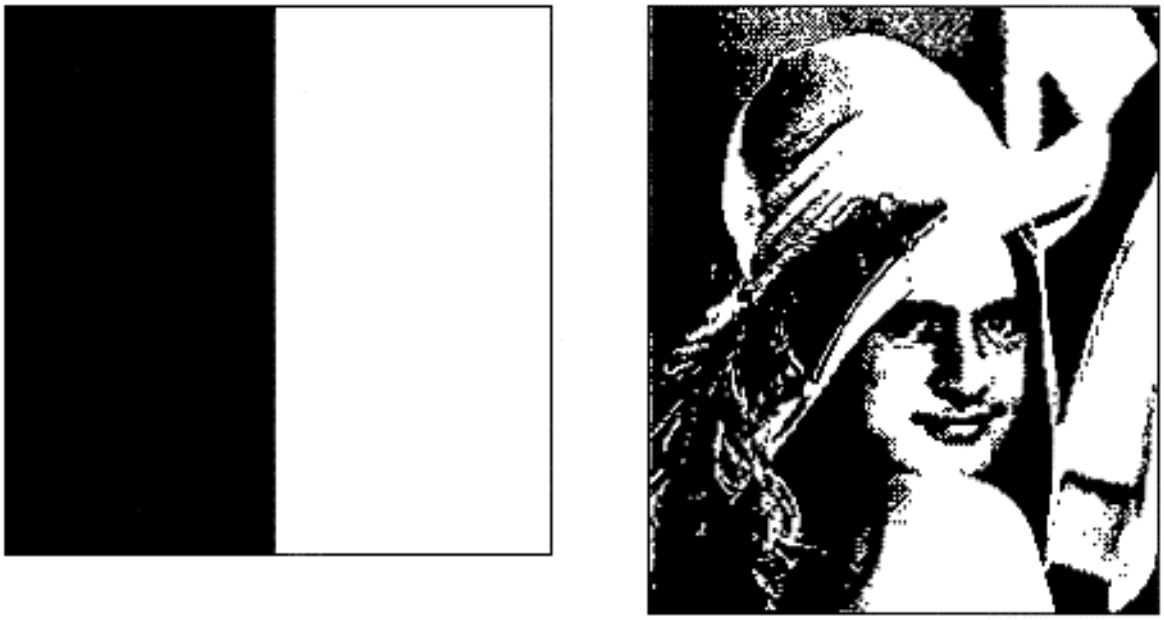


Figure 1
Gray Gradations and Konenna Pictured Using 50% Threshold

part of the error term of surrounding pixels. Ideally, the error term is spread evenly among all surrounding pixels. But to maintain acceptable performance, QuickDraw uses a shortcut. In original QuickDraw, the error term is pushed to the right on even scan lines and to the left on odd scan lines. Color QuickDraw uses the same technique, except it pushes only half the error to the left or right, and the other half to the pixel immediately below. The result of using this technique in Color QuickDraw at monitor resolution for the two test images is shown in Figure 2.

This form of dithering is normally referred to as error diffusion. That is to say that each pixel is thresholded at 50%, but the error incurred in that process is distributed across the image in some manner, thus minimizing information loss. Error diffusion produces very pleasing results when the device being drawn onto is capable of accurately rendering a single dot at the image resolution. Monitors are quite good at this; laser printers are not. If you want your application's output to look good on a laser printer, a different technique is called for.

USING ORDERED DITHERING (HALFTONING)

There are two kinds of laser printers: write-white and write-black. A write-white printer (such as some of the high-end Linotronic printers that use a photographic process) starts the image out black and uses the laser to turn off pixels. A write-black

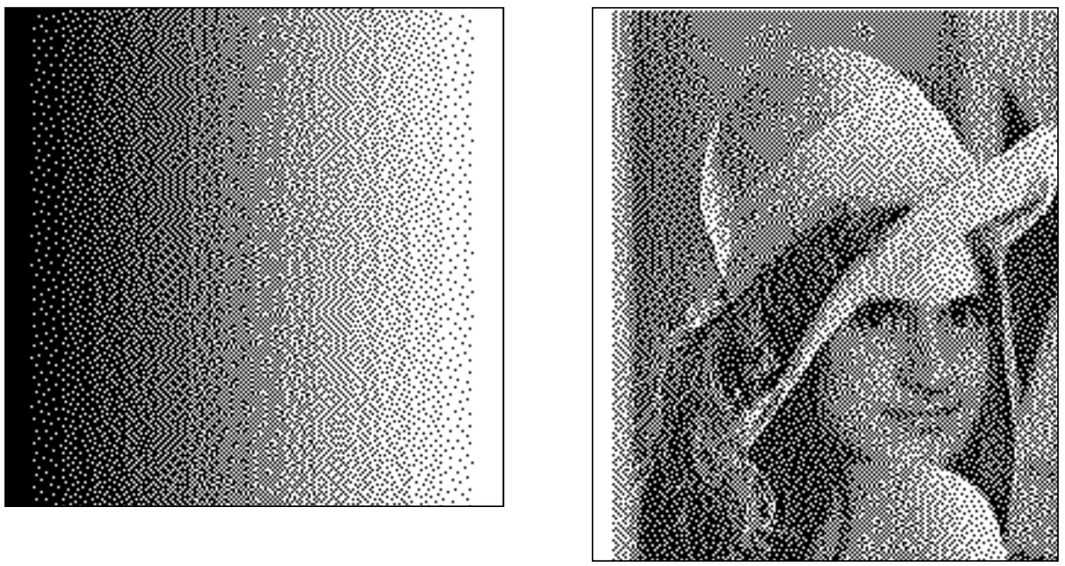


Figure 2
Gray Gradations and Konenna Dithered at Monitor Resolution

printer (such as Apple's LaserWriter) starts the image out white and turns on pixels with the laser. Since the pixels are thought of as being square and the laser beam is round, neither process can accurately turn on or off single pixels.

Generally, the circle generated by the laser beam is slightly bigger than the pixel as the computer "sees" it, to guarantee that all space is covered (see Figure 3). The effect of this with a write-black printer is that the black dots tend to be bigger than the individual pixels, causing any 1-bit image drawn at device resolution to appear too dark. The effect with a write-white printer is that the black dots tend to be smaller than the individual pixels, causing any 1-bit image drawn at device resolution to appear too light. If the area of the circle is 20% greater than the individual pixel, the percentage of unwanted toner, or error, for a single pixel is 20%.

Figure 3
A Laser's Idea of a Square Pixel

Because the error is introduced only at the black/white boundaries, it's reduced when two or more pixels are drawn next to each other. Then the percentage of error is reduced to the perimeter of the pixel group. So in the case where the error for a single pixel is 20%, two pixels drawn next to each other would have only a 15.5% error, and four pixels in a square would have only a 10.25% error in the area covered.

Ordered dithering, or halftoning, minimizes the dot-to-pixel error just described by clumping pixels. Pixels are turned on and off in a specific order in relation to each other and the luminance of the source image. The order can be specified in such a way that clumps of pixels next to each other are turned on as the luminance decreases. This allows us to minimize the effects of the laser printer's dot-to-pixel error. The order is determined by what's known as a dither matrix. (*Warning:* From here on out, things get deep, so put on your waders. You don't really need to understand all the following to use the sample code we provide.)

About the dither matrix. With a dither matrix, to render intermediate shades of gray or primary colors, we sacrifice spatial resolution for shading—that is, we effectively lower the device's dots-per-inch rating while increasing the number of shades that we can print. For example, if we use a 2x2 cell of 300-dpi dots for every pixel on the page, we've lowered the spatial resolution of the device to 150 dpi but we now have 2^4 or 16 different patterns to choose from for each one of the pixels. Each pattern has anywhere from 0 to 4 of the 300-dpi dots blackened, or a density between 0 and 100%. In fact, for the 16 possible patterns there are only five possible densities: 0%, 25%, 50%, 75%, and 100%, corresponding to 0, 1, 2, 3, and 4 dots blackened in the cell. The dither matrix determines which five of the possible patterns to use to represent the five possible densities. It's left to you as an exercise to generate these matrixes using the algorithm we provide below. (The sample code on the *Developer CD Series* disc has a commonly useful example.)

If we construct a matrix with the same dimensions as the dot cell that we're going to use (2x2 for the described case) so that the matrix contains the values 25, 50, 75, and 100, we can use this matrix to determine each of the five possible patterns. Each dot in the pattern corresponds to a position in the matrix. To generate a pattern for 50% gray, we turn on all the dots in the pattern with corresponding matrix values less than or equal to 50. The position of the values in the matrix determines the shape of the pattern, as shown in Figure 4.

The dither matrix is used to render an image in much the same way as the 50% threshold described earlier. In fact, that process uses a 1x1 dither matrix whose single element has a value of 50%. The dither matrix is sampled with (x mod m, y mod n), where (x, y) is the device pixel location and (m, n) is the width and height of the dither matrix.

It turns out that the spatial resolution of the device isn't really reduced by the size of the dither matrix. For regions that are all black, for example, the resolution remains

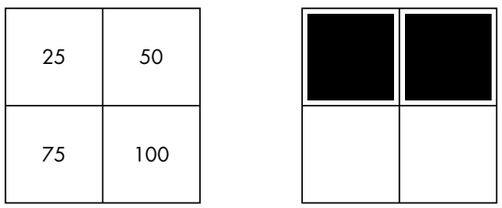


Figure 4
A 2x2 Dither Matrix

the device resolution. Each pixel in the device is still sampled back to a pixel in the source image.

The basic algorithm for doing an ordered dither of an image onto a page becomes the following:

For all device pixels *x*, *y:*

- s_x, s_y = transform(*x*, *y*) where transform maps device pixel coordinates to source pixel coordinates
- If sourceLuminance(s_x, s_y) > ditherMatrix[*x* mod *m*, *y* mod *n*], device-dot(*x*, *y*) = black

The code on the *Developer CD Series* disc is an elaboration on this basic algorithm.

As stated before, the position of the various values in the dither matrix determines the patterns that various luminances generate. A general way to specify this order is to use a spot function, as the PostScript interpreter does. If the rectangle of the dither matrix is thought to be a continuous space whose domain is 0–1 in the *x* and *y* directions, *spot-function*(*x*, *y*) will return some value that ultimately can be converted into a luminance threshold in the matrix. If the desired pattern is a dot that grows from the center as the luminance decreases (known as a clustered-dot halftone), *spot-function*(*x*, *y*) is simply the distance from (*x*, *y*) to the center of the cell (0.5, 0.5). The dither matrix would be generated from the spot function as follows:

```
for i = 1 to m
    x = i/m
    for j = 1 to n
        y = j/n
        matrix[i, j] = spot-function(x, y)
```

The result of this process is that the matrix contains the spot function's results. What we really want in the matrix are threshold values for the luminance. The spot

function result is converted as follows: Treating the dither matrix as a one-dimensional array *A*, generate a sort vector *V* such that *A*[*V*[*i*]] is sorted as *i* goes from 1 to *m***n*. Then, replacing all of the values in *A* with *V*[*i*] * 100/(*m***n*) will yield the desired threshold matrix, with each value being a percentage of luminance. (The code uses numbers that are more computer-friendly than percentages.) These percentages assume that the device is capable of accurately rendering a single pixel. The values can be modified by a gamma function to more accurately produce a linear relationship between image luminance and pixel density.

Ordered dithering is generally done at a specific angle and frequency. The frequency is the number of cells (or dither matrixes) per inch and the angle refers to how the produced patterns are oriented with respect to the device grid. In the preceding example, the frequency (if printing on a 300-dpi device) is 150 cells per inch and the angle is 0°.

Because of the way our brains work (our eyes tend to pick up patterns at 90° angles but not at 45° angles), it's desirable to orient these patterns at arbitrary angles. Since the dither matrix itself is never rotated with respect to the device, we must generate the dither matrix in such a way that it contains enough repetitions of the rotated cell to achieve the effect of being rotated itself. In other words, because a square device requires us to "tile" an area with 0° rectangles, we need to find a 0° rectangle enclosing a part of the rotated pattern that forms a repeatable tile. For some angles of rotation, this rectangle may be much larger than the pattern itself.

Suppose we want to halftone to a 300-dpi device at a frequency of 60 cells per inch and an angle of 45°. At 0°, the dither matrix would be 5x5 (300/60), yielding 26 possible shades of gray. However, as Figure 5 illustrates, we need an 8x8 matrix to approximate the desired angle. These dimensions are found by rotating the vectors (0, 5) and (5, 0) by 45° and pinning them to integers, yielding the vectors (4, 4) and (-4, 4). Since the magnitude of the vector (4, 4) is 4*sqrt(2), the actual halftone frequency achieved will be 300/(4*sqrt(2)), around 53. The error in frequency and angle is due to the need to pin the vectors to integer space.

Here's the basic algorithm for computing the dither matrix:

1. The halftone cell is specified by the parallelogram composed of the vectors (x_1, y_1) and (x_2, y_2) and based at (0, 0).
2. *A*, the area of the modified halftone cell, is (x_1*y_2) - (x_2*y_1). For the required dither matrix, the horizontal dimension is *A*/*P* and the vertical dimension is *A*/*Q*, where $P = GCD(y_2, y_1)$ and $Q = GCD(x_2, x_1)$.
3. For every point in the matrix, which is in (*x*, *y*) orthogonal space, we want to find its relative position in the space of one of the repeated halftone cells, defined by the vectors (x_1, y_1) and (x_2, y_2). (See Figure 6.) Call this point (*u*, *v*). The transformation is

The source of step 2 in the above algorithm is "An Optimum Algorithm for Halftone Generation for Displays and Hard Copies" by Thomas M. Holladay, from the *Proceedings of the Society for Information Display*, Vol. 21, No. 2, 1980.•

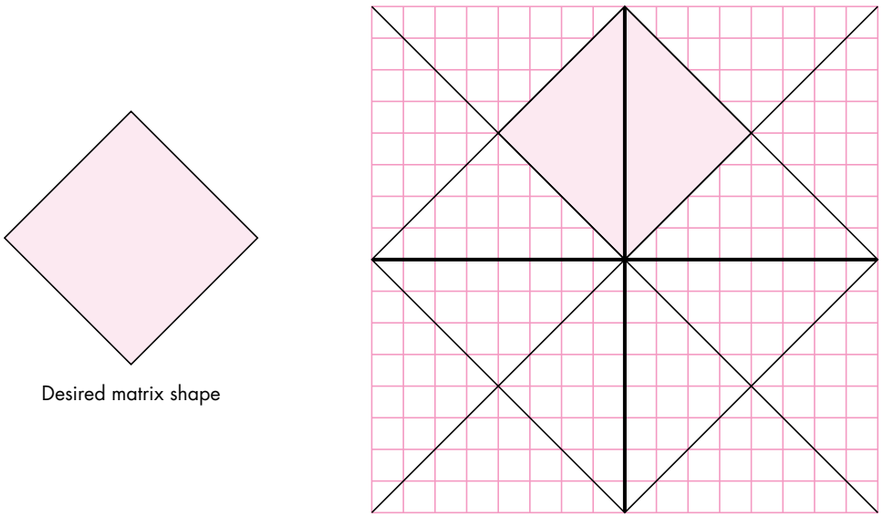


Figure 5
Approximating the Desired Angle

$u = A*x + B*y$, $v = C*x + D*y$. Since the point (x_2, y_2) in (x, y) space is the point (1, 0) in halftone cell space and the point (x_1, y_1) is the point (0, 1) in halftone cell space, the coefficients A, B, C, and D are found by solving the following simultaneous linear equations:

$$\begin{aligned} A*x_1 + B*y_1 &= 0 \\ C*x_1 + D*y_1 &= 1 \\ A*x_2 + B*y_2 &= 1 \\ C*x_2 + D*y_2 &= 0 \end{aligned}$$

We compute the dither matrix in the rotated case as follows:

For each position in the matrix (i, j):

- Get (x, y) the center of the matrix point (i, j)

  $x = i + 0.5$
  $y = j + 0.5$

- Transform (x, y) to a point in halftone cell space (u, v)

  $u = A*x + B*y$
  $v = C*x + D*y$

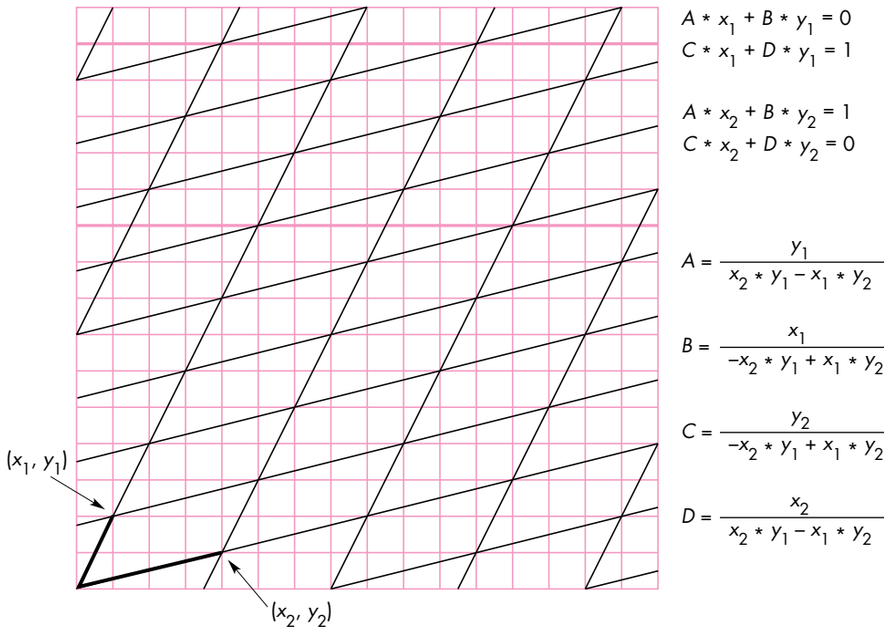


Figure 6
Transforming a Halftone Cell

u and *v* now express the point (*x*, *y*) as multiples of the two cell vectors. Therefore, the fractional parts of *u* and *v* represent the position as if the particular halftone cell at the point (*x*, *y*) were the (0, 0) cell.

- *Z* = *spot-function* (*u* - floor(*u*), *v* - floor(*v*))
- Find the index of the record (containing fields *x*, *y*, and *Z*) such that *u* = *x*, *v* = *y*. If the record doesn't exist, enter *u*, *v*, *Z* into the table. (Note that the equality between [*u*, *v*] and [*x*, *y*] requires an allowable epsilon difference to account for fixed-point round-off error.)
- matrix[*i*, *j*] = index

Find the order of records sorted by values of *Z;* store order in sort vector (described earlier in connection with converting the spot function result). Reassign values of matrix based upon sort vector.

Figure 7 shows our example matrix with values from 0 through 255, representing luminances, filled in. A luminance from an image with this range could be sampled

directly against the matrix. The values in this matrix are those that would actually be used for a 300-dpi, 60-line-per-inch, 45° halftone. As in Figure 5, the matrix is repeated four times for the sake of clarity, with the 45° halftone cells overlaid. The position of any particular number in the matrix relative to the 45° cell it falls in corresponds exactly to the relative position of that same number in any of the other 45° cells. Thus, the effect of having a rotated halftone cell is created with an unrotated dither matrix.

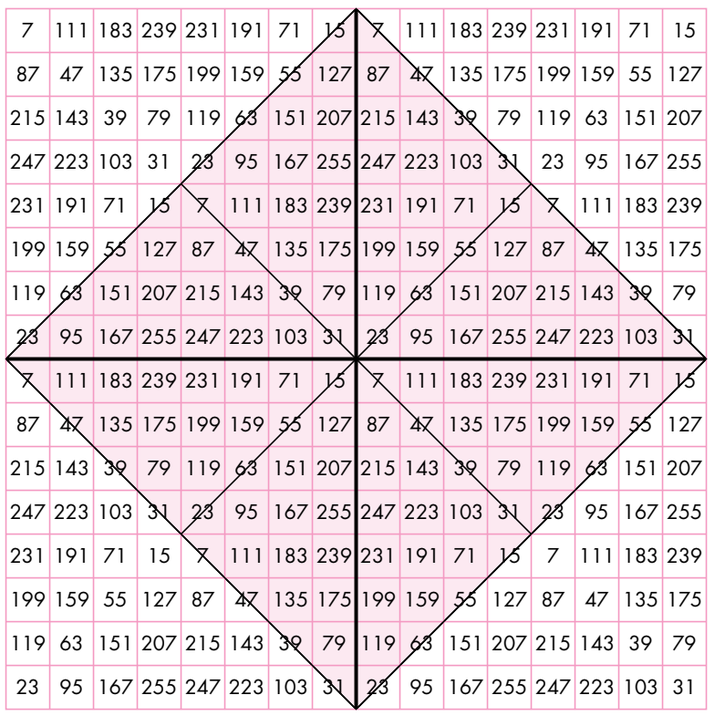


Figure 7
Our Example Matrix With Luminance Values Filled In

This particular example leads us to some other interesting possibilities. It turns out that QuickDraw patterns are 8x8 matrixes, just like our example. This means that we can halftone other QuickDraw primitives besides pixMaps when drawing to a 300-dpi non-PostScript device (provided that pattern stretching is disabled, by setting the bPatScale field in the print record to 0) and achieve a look similar to what a PostScript device would give us.

Here's how. Suppose we want to paint a region with a luminance of 150 on the scale from 0 to 255. We simply create a QuickDraw pattern in which all of the 1 bits correspond to the cells in the 8x8 matrix that are greater than or equal to 150. This pattern (shown in Figure 8) can then be used to paint any region or other QuickDraw primitive to get the halftone effect. Furthermore, because QuickDraw patterns are aligned to the origin of the grafPort, separate objects drawn touching one another will not generate undesirable seams, even when drawn with different shades. The nature of the clustered dot pattern is such that gradations appear continuous to the extent possible at the resolution of the device.

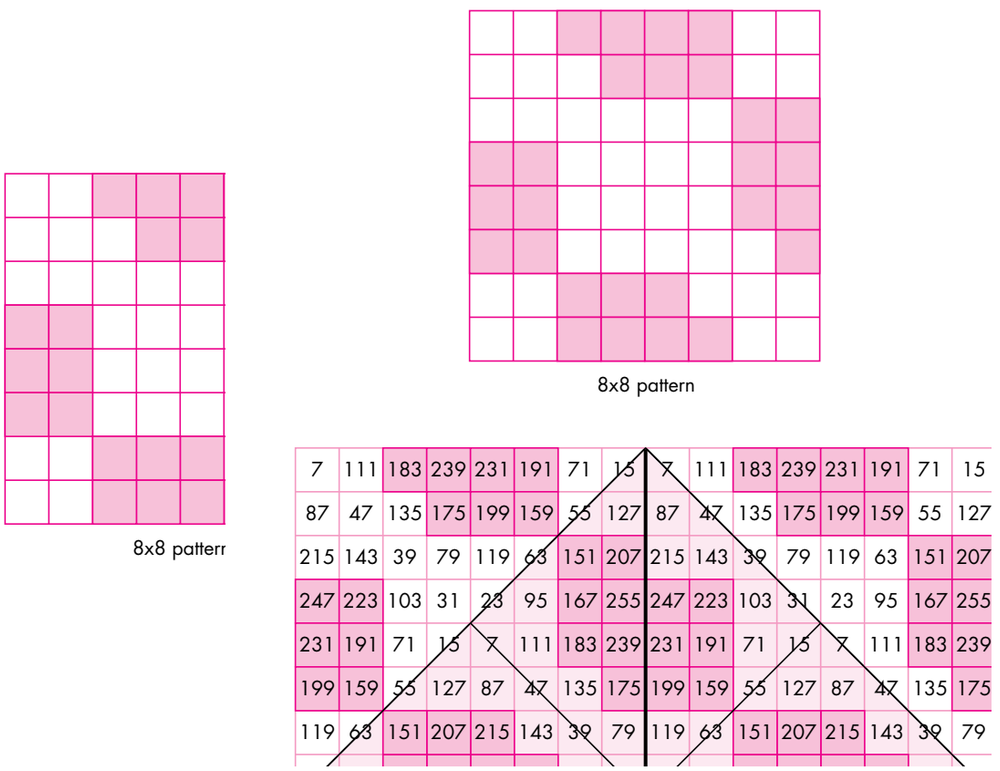
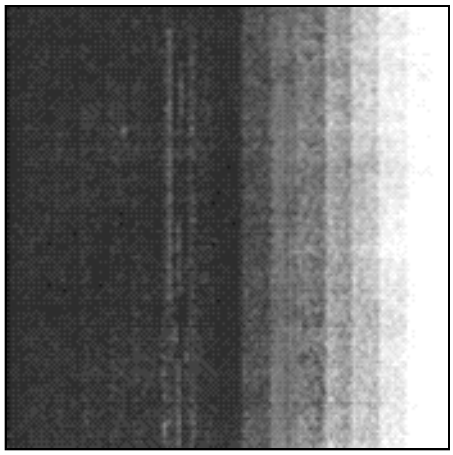
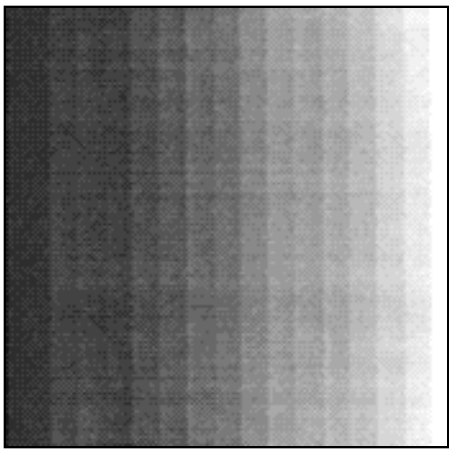


Figure 8
Pattern for an Image With a Luminance of 150

Figure 9 shows the gray gradations and Konenna printed on a laser printer using error-diffusion dithering compared with halftoning using the 8x8 matrix. The difference in print quality is radical. For more commentary on this difference, see "Printing: Ideal Versus Real."



Gray gradations dither



Gray gradations halftone

Konenna dither

Konenna halftone

Figure 9
Gray Gradations and Konenna Dithered and Halftoned at Laser Printer Resolution

PRINTING: IDEAL VERSUS REAL

We've already talked about the error introduced in printing by the fact that the laser beam is round while the pixel is square. Many other factors also can make the transfer of toner to paper deviate from the ideal. Sources of error include differences in inks, papers, printer drums, and even humidity. Additionally, a printer's behavior changes over time as the drum wears. Compensating for all these factors to achieve ideal images would require constant calibration and recalibration of the printer.

An error appears most pronounced in the final print when imaging directly at device resolution, as Figure 9 shows. Halftoning hides much of this error and produces reasonably uniform results among printers with varying degrees of error.

The tonal reproduction curves (known as TRC or gamma curves) shown in Figure 10 indicate the gray levels produced by the Apple LaserWriter when dithering and halftoning. Note that with dithering, the measured luminance of an image remains dark much longer than with halftoning as requested luminance increases, due to the error when each pixel is printed. Of particular interest is the point on the dither curve right at 50% luminance. The measured luminance is actually darker than when 44% luminance is requested. The reason is that with a 50% dither, every other pixel is drawn, maximizing the effect of the laser error.

While the TRC curve for the halftone print doesn't match the ideal curve, it's much closer to the ideal than is the dither curve. To get the halftone even closer to ideal, you could adjust the luminance calculation by the amount indicated by the halftone TRC to compensate. Indeed, most image-processing applications perform this TRC adjustment to compensate for the nonlinearities of the output device. See *Designing Cards and Drivers for the Macintosh Family,* Second Edition (Addison-Wesley, 1990) for more information about how gamma correction works on the Macintosh II family for monitors.

About the code. And now, about the code. To illustrate the principle of dithering, our sample code is pixel-based—that is, the calculations are done on a pixel basis. Thus, the perfomance is sluggish. A real-world commercial application would use an optimized version of this code. One way to do this is to make the routines work on a scan-line rather than a pixel basis. Also note that the routine that does the halftoning only supports input pixMaps of 8 or 32 bits. It would be easy to extend the routine to accept pixMaps of other depths.

The first routine we need is one that calculates the luminance given a pointer to the current pixel. The LUMVAL routine returns a long luminance in the range of 0 to 255 using the 30%-59%-11% formula described previously.

```
long LUMVAL(Ptr pPixel, PixMapPtr pMap)
   {
      long      red, green, blue;

      if (pMap->pixelSize == 32) {
         red = (long)(unsigned char)*(++pPixel);      /* Skip alpha,
                                                         get red. */
         green = (long)(unsigned char)*(++pPixel);    /* Get green. */
```

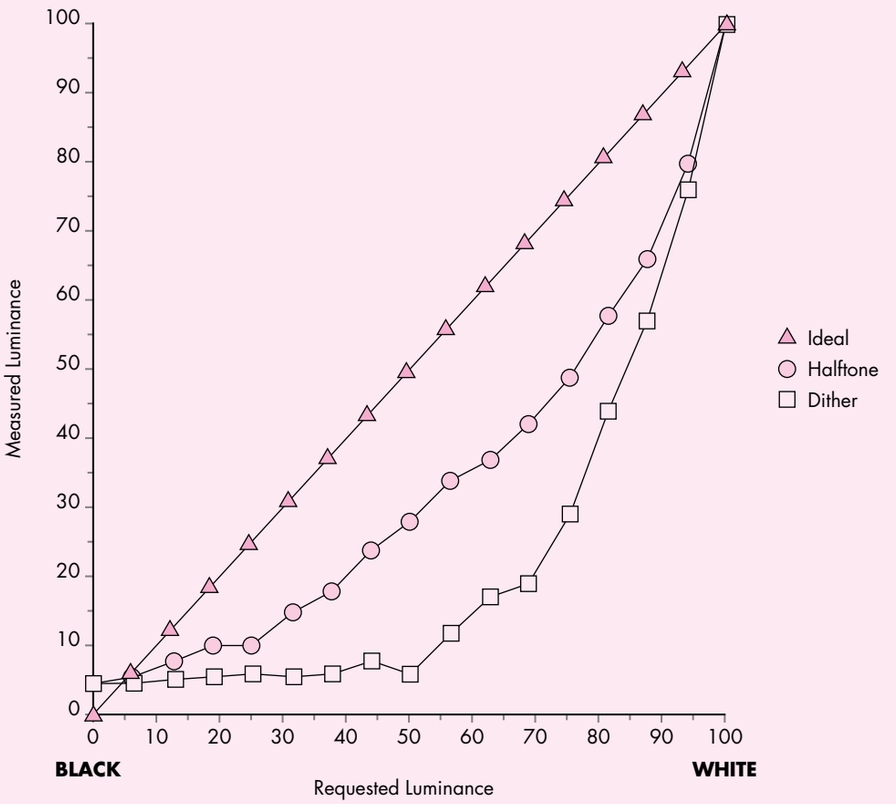


Figure 10
TRC Curves for the LaserWriter

```
        blue = (long)(unsigned char)*(++pPixel);     /* Get blue. */
        return((30 * red + 59 * green + 11 * blue)/100);
    } else if (pMap->pixelSize == 8) {
        RGBColor*       theColor;
        theColor = &((*(pMap->pmTable))->ctTable[ (unsigned
            char)*pPixel ].rgb);
        return( (30 * (theColor->red >> 8) + 59 * (theColor->green >>
            8) + 11 * (theColor->blue >> 8))/100);
    }  /* End if */
}  /* LUMVAL */
```

The routine that actually does the halftoning is the HalftonePixMap routine. Rather than taking a PixMapPtr as the CreatePICT2 routine did, this routine takes a PixMapHandle. This enables us to pass in either a pixMap we create manually (as we did when we called CreatePICT2) or a PixMapHandle that QuickDraw creates (for example, from a GWorld). We must distinguish which one we pass in so that the routine knows whether it can access the fields of the pixMap directly (which it can if we created it) or if it must use QuickDraw to access the fields. This is relevant only for the LockPixels and GetPixBaseAddr routines.

Furthermore, the HalftonePixMap routine assumes the resolution of the source pixMap is 72 dpi (screen resolution) and only supports devices with square pixels (same hRes and vRes). You can pass in the resolution of the destination device in the Resolution parameter, but it must be greater than or equal to 72 dpi.

Like the CreatePICT2 routine, HalftonePixMap returns a PicHandle. In this case, the picture contains a 1-bit/pixel pixMap. You can display it using DrawPicture.

The prototype for the HalftonePixMap routine is

```
PicHandle HalftonePixMap(PixMapHandle hSource, Boolean qdPixMap,
   short Resolution);
```

The source code for the complete routine can be found on the *Developer CD Series* disc.

USING RANDOM DITHERING

Random dithering is yet another kind of dither useful for drawing images. It's discussed last, however, because of its inherent limitations.

The method is simple. It's much the same as the 50% threshold method described earlier. The only difference is that instead of being compared to 50%, the luminance values are compared to a random number between 0 and 100%. The effect of this is that the probability of any dot in the device image being turned on is directly proportional to the luminance of the pixel in the source image at the corresponding point.

This method has three limitations. First, calculating a random number is an expensive operation that we would not want to do for every device pixel. Second, except at very high resolutions, images dithered in this manner appear very noisy, like bad reception on a black-and-white TV. And third, this method requires a random number generator that's very good at producing a uniform distribution.

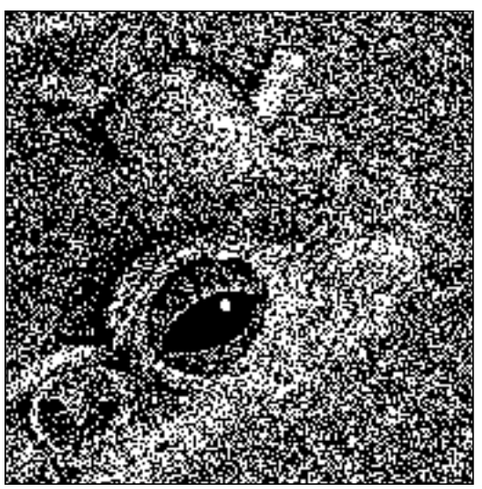
Ironically, this least frequently used method of dithering most accurately models the physical process of photography. Photographic film is like laser printing in that it's composed of pixels. However, the pixels are grains of silver rather than toner.

Additionally, there are tens of thousands of grains per inch rather than the 300 dots per inch we're used to with laser printers. The lower the ASA rating of the film, the higher the grain density.

The place on a film where a photon strikes one of these silver grains turns black when the film is developed (which is why you get negatives). Since photons are really, really, really small, the likelihood of a single photon striking one of the grains of silver is very low. However, the brighter the light, the more photons there are; so the probability of striking one of those silver grains increases in proportion to the luminance. Thus, we see how random dithering simulates photography.

Figure 11 shows the image of a frog's head produced using halftoning with an 8x8 matrix as compared with using a 72-dpi random dither. You can see that the randomly dithered image looks like a really grainy photograph.

Halftoned with 8x8 at 72 dpi



Randomly dithered at 72 dpi

Figure 11
Frog's Head, Halftoned and Randomly Dithered

HASTA LA VISTA, BABY

This article has addressed several issues. First, the problem of saving deep pixMaps on machines with original QuickDraw was overcome by showing you how to manually create a PICT, which can then be rendered by calling DrawPicture. Such a

PICT can be exported by an application so that it can be viewed in color on a Color QuickDraw machine.

Second, several solutions to the problem of displaying and printing color images on black-and-white devices were discussed. Images can be displayed on screen using a 50% threshold or error-diffusion dithering. Ordered dithering (halftoning) provides a way to get around the problem of the laser printer's inability to resolve single pixels. Random dithering has practical limitations but represents yet another alternative for producing color images on black-and-white devices.

Thanks to these techniques, the market for applications that deal with color images need not be limited to Color QuickDraw machines and PostScript printers. The necessary code is small (and already written for you) and the gain in functionality is very high. Now get to work on those applications!

WANT TO READ MORE?

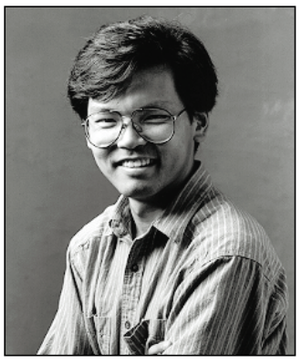
If you'd like to delve more deeply into the mysteries of processing color images for display, check out the following:

- "An Optimum Algorithm for Halftone Generation for Displays and Hard Copies" by Thomas M. Holladay, in the *Proceedings of the Society for Information Display,* Vol. 21, No. 2, 1980.
- *Digital Halftoning* by Robert Ulichney (MIT Press, 1987). This book, based on a Ph.D. thesis done at MIT, is devoted entirely to discussing halftoning algorithms; it's extremely thorough and includes many example images halftoned in different ways.
- *Fundamentals of Interactive Computer Graphics* by J. D. Foley and A. Van Dam (Addison-Wesley, 1982). The standard text on computer graphics. Not nearly as thorough as Ulichney, but has a solid discussion of the basics.

And then, of course, the two books all Macintosh programmers should own:

- *Programming with QuickDraw* by Dave Surovell, Frederick Hall, and Konstantin Othmer (Addison-Wesley, 1992). Everything you need to know about graphics on the Macintosh.
- *Debugging Macintosh Software with MacsBug* by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991). Everything you need for debugging Macintosh software, including in-depth discussions of a number of the Macintosh managers.

THANKS TO OUR TECHNICAL REVIEWERS
Sean Parent, Forrest Tanaka, Dave Williams•



GRAPHICS HINTS FROM FORREST

USING THE PALETTE MANAGER OFF-SCREEN

FORREST TANAKA

Most people who've done any graphics programming on the Macintosh are aware of the Palette Manager, because it's the documented way to control the on-screen color environment, and perhaps because my cohorts and I in Developer Technical Support keep going on about how right the world would be if everyone used it. In an effort to follow the rules as best they can, some people have taken the Palette Manager so much to heart that they use it not only with windows, but with off-screen cGrafPorts as well—something that isn't heard about very much. Some of these people have concluded that all the features of the Palette Manager apply just as well to off-screen cGrafPorts as they do to windows. Logical enough, right?

Well, that's the kick; whether this is logical or not, the truth is that only a small part of the Palette Manager works with off-screen cGrafPorts. Specifically, the pmCourteous usage mode and the pmWhite and pmBlack usage-mode modifiers work fine when they're used in a palette that's attached to an off-screen cGrafPort, but the pmTolerant, pmAnimated, and pmExplicit usage modes do not. In this column, I'll describe how you can take advantage of the Palette Manager features that work off-screen and how you can simulate the features that don't work.

The pmCourteous usage mode seems pretty useless to a lot of people because it has no effect on the current color environment. But in general, making a palette full of pmCourteous colors is a lot better than hard-coding RGBColors into your code. Instead of hard-coding colors, make a palette of courteous colors—as many entries as you need colors—and save it as a 'pltt' resource. When your application runs, call SetPalette to attach this palette to your off-screen cGrafPort. When you need to use a color while drawing into this cGrafPort, pass the desired color's palette index to PmForeColor or PmBackColor, and then draw. This is better than hard-coding colors because you or a software localizer can easily change the colors by changing the 'pltt' resource—no code changes are necessary.

The pmWhite and pmBlack usage-mode modifiers are new with System 7; they let you specify whether you want a particular palette entry to map to white or black in a black-and-white graphics environment. By default, colors whose average color-component value is larger than 32767 are mapped to white and other colors are mapped to black. (If you use RGBForeColor, Color QuickDraw also checks to see whether your specified color is different from your background color but maps to your background color; if so, Color QuickDraw uses the complement of the color you specified so that your drawing is visible over the background.) By specifying that a palette entry is pmCourteous + pmBlack or pmCourteous + pmWhite, you can control which colors map to black and to white when there aren't enough colors available. This applies to palettes attached to off-screen cGrafPorts as well as to palettes attached to windows.

Those are the Palette Manager features that do work off-screen. Now I'll talk about the features that don't and what you can do to get the same effect.

The pmExplicit usage mode is handy when you want to draw using a pixel value without knowing or caring what color that pixel value represents. With this mode you can easily show the colors in a screen's color table, and you can also draw into a pixel image with a specific value even though you specify the color for that value elsewhere.

FORREST TANAKA has been playing Developer Technical Support as one of the graphics support people for slightly more than two years. "It amazes me still," he says, "that the more you learn about the Macintosh graphics tools, the farther off total understanding seems to be." Outside of DTS, he likes to ride his bike, and uses it to commute the three blocks to his office ("Hey, it's faster than driving the three blocks!"), and he likes to try getting his radio-controlled car to act as if it's actually controlled.•

PRINT HINTS FROM LUKE & ZZ is in hibernation.•

When you have a palette that's attached to an off-screen cGrafPort, pmExplicit colors are interpreted as pmCourteous colors. Instead of using a palette, you should convert your pixel value to an RGBColor and use this as the foreground or background color. Set the current GDevice to your off-screen GDevice so that the color environment is set; then pass your pixel value to Index2Color, which is documented on page 141 of *Inside Macintosh* Volume V. Index2Color converts your pixel value to the corresponding RGBColor, which you can pass to RGBForeColor or RGBBackColor, and then you can draw. The result is that your pixel value is drawn into the destination pixel image.

Both the pmAnimated and pmTolerant usage modes are used to modify the color environment, and both are interpreted as pmCourteous when they're in a palette that's attached to an off-screen cGrafPort. The most important difference between the two usage modes is in the style of color-table arbitration that they do—pmTolerant gives the front window the colors it needs, while pmAnimated additionally makes sure that nothing outside the front window is drawn in its colors. Color-table arbitration doesn't apply off screen, so the pmAnimated and pmTolerant usage modes can be unified into "I want to change my off-screen colors."

Changing the colors in an off-screen color environment means changing its color table; the most straightforward way to do this is to modify the contents of the color table directly. That is, get your off-screen color table's handle and then directly assign new values to the rgb fields in its CSpecArray. You could also assign a whole new color table to the off-screen environment by assigning the new one to the pmTable field of the off-screen pixMap. Either way, you have to tell Color QuickDraw what you've done by updating the changed color table's ctSeed field. The next time you draw into your off-screen graphics environment, Color QuickDraw detects your change by comparing the ctSeed of your changed color table against the iTabSeed of the current GDevice's inverse table, and it rebuilds the inverse table according to the changed color table. You can update the ctSeed field by assigning to it the return value of GetCTSeed, which is documented on page 143 of *Inside Macintosh* Volume V. If the 32-Bit QuickDraw extensions are available, you can update a color table's ctSeed simply by passing the color table to CTabChanged, documented on page 17-26 of *Inside Macintosh* Volume VI.

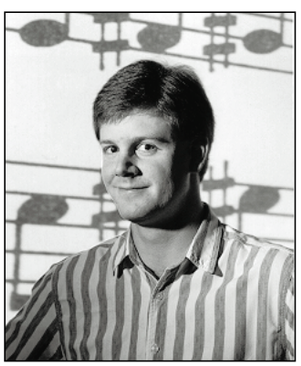
If you have a GWorld and you want to replace its color table, you should call UpdateGWorld, passing it a new color table. UpdateGWorld makes sure that all the cached parts of a GWorld are properly updated, which is tough to do any other way. If you don't pass any flags to UpdateGWorld, it's within its rights to destroy your existing GWorld's image. But if you pass the clipPix or stretchPix flag, UpdateGWorld is obligated to keep your existing image, and it tries to reproduce the existing image in the new colors as best it can.

To wrap up, you can use the Palette Manager with off-screen graphics environments, but you'll only be able to use the pmCourteous usage mode and the pmWhite and pmBlack usage-mode modifiers. But that's not to cast aspersions on these features, because they can be very handy for both on-screen and off-screen drawing. The pmExplicit, pmTolerant, and pmAnimated usage modes don't work for off-screen drawing, but there are easy ways to simulate those features without the Palette Manager and without risking future compatibility.

For more details about changing or replacing off-screen color tables, see the October 1991 version of Macintosh Technical Note #120, "Principia Off-Screen Graphics Environments."•

THE TEXTBOX YOU'VE ALWAYS WANTED

NeoTextBox is an alternative to the TextEdit utility routine TextBox. NeoTextBox provides full-justification capability and the option to use TrueType features while retaining all the advantages of TextBox. The three routines that comprise NeoTextBox compile to fewer than 900 bytes yet offer a 40% performance increase over TextBox in common cases.



BRYAN K. ("BEAKER") RESSLER

In the deepest, darkest corner of the TextEdit chapter in *Inside Macintosh* Volume I, there's an extremely useful routine called TextBox.

```
pascal void TextBox(void *text, long length, Rect *box, short just)
```

Given a rectangle and some text, TextBox word wraps the text inside the rectangle, drawing in the font, style, and size specified in the current grafPort.

Anyone who's tried to word wrap text knows that it's not as easy as it first appears. Perhaps that's why TextBox takes the approach it does: to perform its task, TextBox creates a new TERec with TENew, sets up the rectangles in the record, and calls TESetText to create a temporary handle to a copy of the text you provided to TextBox. TextBox then calls TEUpdate to wrap and draw the text, and finally TEDispose to dispose of the TERec. By calling TextEdit to do the text wrapping and drawing, TextBox avoids doing any hard work. Unfortunately, it also incurs quite a bit of overhead.

Despite its pass-the-buck implementation, TextBox's use of TextEdit has several advantages. Perhaps most important, TextBox works correctly with non-Roman script systems like Japanese and Arabic without the need for any extra programming. Another handy side effect is that updates in TextEdit degenerate into calls to DrawText, and can therefore be recorded into QuickDraw pictures. TextBox was designed specifically for drawing static text items in dialog boxes and performs this function well.

BRYAN K. RESSLER, or "Beaker" as he's known at Apple, is one of our twisted software engineers who seems to be convinced that anything is possible on a Macintosh, and if it's already been done, it can be done better. He got his BSCS from the University of California, Irvine, and wrote commercial MIDI applications before coming to Apple. Beaker wrote many of the programs used for testing TrueType fonts. When he's not on a coding frenzy, he writes *noncommercial* MIDI applications, tries to have a life, and keeps a consistent blood-caffeine level so high you need scientific notation to express it.•

So TextBox is great—if you're drawing dialog boxes. But you want more. You want better performance. You want more flexibility. You want to control line height. You want full justification (instead of only left, center, and right alignment). You want to use whizzy TrueType calls when they're available. You want to control the text drawing mode. You can't stand the way TextBox always erases (and therefore isn't too useful when you're drawing to printers—it slows printing way down). Yeah, and you don't like that 32K text limitation either. You want to word wrap *War and Peace* in a single call to TextBox. And you'd like some useful information back, too, like the line height it used, and where the last line of text was drawn, so that you can draw something below the text. And, of course, you want to retain the advantages of TextBox.

Well, this is your lucky day.

ENTER NEOTEXTBOX

NeoTextBox is the TextBox you've always wanted (and didn't even have to ask for). NeoTextBox is on the average 33% faster than an equivalent call to TextBox. Plus, it's considerably more flexible:

- NeoTextBox allows a line height specification. You can ask for the default (same behavior as TextBox); use variable line height, which adjusts for characters that extend beyond the font's standard ascent or descent line; or specify a line height in points.
- NeoTextBox provides left, center, and right alignment and full justification.
- NeoTextBox never erases the rectangle it's drawing into. It lets you erase or, if you wish, draw a colored background.
- NeoTextBox returns the total number of lines in the wrapped text.
- NeoTextBox can return, via VAR parameters, the vertical pen position of the last line of text and the line height that was used to draw the text.

NeoTextBox gives you all this extra functionality, yet retains the advantages of TextBox. It is completely language independent and uses the Script Manager heavily (just like TextEdit). It's easy to call, and if you don't want all the spiffy new features, it's easy to get TextBox-like behavior with a free performance increase.

Let's take a look at the parameters for NeoTextBox.

```
short NeoTextBox(unsigned char *theText, unsigned long textLen,
   Rect *wrapBox, short align, short lhCode, short *endY,
   short *lhUsed)
```

The first two parameters, theText and textLen, are analogous to TextBox's text and length parameters: they specify the text to be wrapped. Note that theText isn't a Pascal string—it's a pointer to the first printable character.

The third and fourth parameters, wrapBox (box in TextBox) and align, also hearken back to NeoTextBox's ancestor. Just as in TextBox, wrapBox specifies the rectangle within which you're wrapping text, and the align parameter specifies the alignment. In addition to the standard TextEdit alignments teFlushLeft, teCenter, and teFlushRight (see "Text Alignment Constants for System 7"), a new alignment is defined—ntbJustFull. It performs full justification in whatever manner is appropriate for the current script.

The fifth parameter, lhCode, specifies how the line height is derived. If lhCode is 0, the default line height is derived via a call to GetFontInfo. This gives the same behavior as TextBox. If lhCode is less than 0, the line height is derived by determining which characters in the text that's being drawn extend the most above and below the baseline (see "SetPreserveGlyph With TrueType Fonts"). Finally, if lhCode is greater than 0, the value of lhCode itself specifies the line height. For instance, you can draw 12-point text in 16-point lines.

The last two parameters, endY and lhUsed, are reference parameters that allow you to retrieve the vertical position of the last line of text and the line height that was used to draw the text, respectively. The endY parameter can be very useful if you

TEXT ALIGNMENT CONSTANTS FOR SYSTEM 7

Before System 7, there was a conflict between the names of the text alignment constants and their actual behavior. To help make applications compatible with non-Roman scripts, teJustLeft was interpreted as the default text alignment appropriate for the current script rather than forcing text to be aligned on the left as specified. For example, on a Hebrew system, a TextBox call with a just parameter of teJustLeft would actually use the default justification for Hebrew, which is teJustRight.

To overcome this conflict, new constants were introduced in System 7, as shown in Table 1.

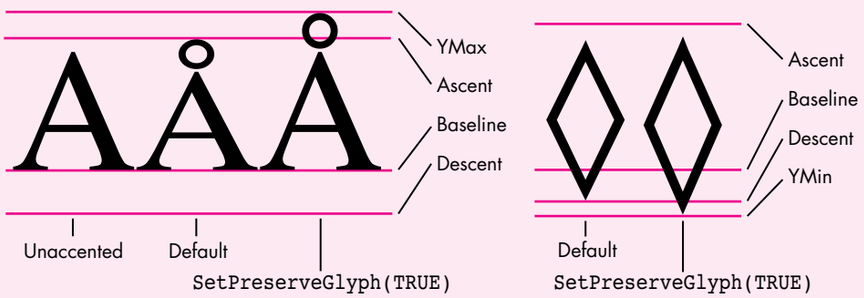
Table 1
Text Alignment Constants

New Constant	Old Constant	Value	Meaning
teFlushLeft	teForceLeft	–2	Align text on the left for all scripts
teFlushRight	teJustRight	–1	Align text on the right for all scripts
teFlushDefault	teJustLeft	0	Use conventional alignment for script
teCenter	teJustCenter	1	Center text for all scripts

SETPRESERVEGLYPH WITH TRUETYPE FONTS

Before TrueType, all characters in all fonts fit beneath the font's ascent line and above the descent line, like the default characters shown in Figure 1. Bitmapped fonts were drawn so that diacriticals, like the angstrom over the *A* in Ångström, would fit beneath the ascent line. To do this, the letterform had to be distorted. With the advent of TrueType, this "feature" can be controlled, because TrueType fonts carry outline data that's true to the original design (hence the name TrueType).

Since most applications expect characters to fit beneath the ascent line and above the descent line, QuickDraw transforms characters in TrueType fonts to force them within those bounds. To override this transformation and preserve the original glyph shape, use the Font Manager call SetPreserveGlyph(TRUE). After this call, TrueType fonts will be drawn as shown to the right in Figure 1. Preserving the glyph shape makes it possible to take advantage of NeoTextBox's variable line height feature.



Example shown: 72-point Times

Example shown: 72-point Symbol

Figure 1
How SetPreserveGlyph Affects Line Height

intend to draw anything below the text, since it tells you exactly where the last line of text was drawn. To find out what the actual derived line height was if you used a negative lhCode, use the lhUsed parameter. Pass nil for either or both of these last two parameters if you don't want this extra information.

NeoTextBox returns the *total* number of lines in the text. That includes lines clipped off because they were below the bottom of wrapBox. You can tell whether the text overflowed wrapBox by whether the value returned in endY is greater than wrapBox.bottom. If you want to know how many lines fit in wrapBox, simply divide the height of wrapBox by the value returned in lhUsed.

REQUIREMENTS

NeoTextBox uses some advanced Script Manager routines that are available only in System 6 or later. NeoTextBox assumes they're available, so make sure your main program checks that it's running on System 6 or later via a Gestalt or SysEnvirons call.

NeoTextBox requires one global variable, a Boolean named gHasTrueType. It should be set to TRUE if the TrueType trap ($A854) is available, or FALSE if not. If your development environment provides glue for Gestalt, you can use the following lines to set up gHasTrueType:

```
#define kTrueTypeTrap 0x54   /* The TrueType trap number */
#define kUnimplTrap   0x9f   /* The "unimplemented" trap number */
long  gResponse;

if (Gestalt(gestaltFontMgrAttr,&gResponse) == noErr)
   gHasTrueType = BitTst(&gResponse,31-gestaltOutlineFonts);
else {
   gHasTrueType = (NGetTrapAddress(kTrueTypeTrap,ToolTrap) !=
      NGetTrapAddress(kUnimplTrap,ToolTrap));
}
```

THE BASIC ALGORITHM

NeoTextBox does a lot. But, in order to appease the programmer's natural desire to avoid work, we allow the Script Manager to do the hard parts. (Do you know how to do full justification in Arabic?) In short, here's how NeoTextBox gets its job done:

1. It saves the current grafPort's clipping region and clips to the box we're drawing into.
2. It calculates the appropriate line height with the function NTBLineHeight.
3. It calls the Script Manager routine StyledLineBreak to find each line-break point in the input text.
4. It draws each line with the function NTBDraw.
5. It advances the pen down one line.
6. When there's no more text, it restores the clipping region and returns the appropriate values.

It sounds simple, doesn't it? That's because StyledLineBreak does all the work. It knows how to find word breaks in whatever script we're using. StyledLineBreak is smart, too. For instance, in English, it knows that it's OK to break a hyphenated word if necessary. It uses rules that are provided by the installed script systems, so it always takes the appropriate actions. Let's take a closer look at the code.

THE NEOTEXTBOX FUNCTION

The source code for NeoTextBox that's shown here is written in MPW C 3.2. We'll start in the NeoTextBox function and break out to a couple of utility functions when we come upon them.

Here's the NeoTextBox declaration and local variables:

```
short NeoTextBox(unsigned char *theText, unsigned long textLen,
   Rect *wrapBox, short align, short lhCode, short *endY,
   short *lhUsed)
{
   RgnHandle          oldClip;        /* Saved clipping region */
   StyledLineBreakCode breakCode;     /* From StyledLineBreak */
   Fixed              fixedMax;       /* boxWidth in fixed point */
   Fixed              wrapWid;        /* Width to wrap within */
   short              boxWidth;       /* Width of box */
   long               lineBytes;      /* Number of bytes in one line */
   unsigned short     lineHeight;     /* Calculated line height */
   short              curY;           /* Current vert pen location */
   unsigned short     lineCount;      /* Number of lines we've drawn */
   long               textRemaining;  /* Number of bytes of text left */
   unsigned char      *lineStart;     /* Pointer to start of a line */
   unsigned char      *textEnd;       /* Pointer to end of input text */
```

Many of these variables are used in the call to StyledLineBreak, which is explained in detail later. The most important variables to know about here are breakCode, which contains the line break code returned by each call to StyledLineBreak; lineStart and lineBytes, which are returned by StyledLineBreak to specify a single line; and curY, the current vertical pen location.

GET READY

NeoTextBox, like TextBox, clips to wrapBox. Since this is a general-purpose routine, it's safest to save the clipping region, then restore it at the end. We calculate the width of wrapBox, because it's used a lot, and convert it to fixed point as fixedMax, which is used in calls to StyledLineBreak as a VAR parameter. Also, we retrieve the appropriate text alignment if the user has requested default alignment.

```
GetClip((oldClip = NewRgn()));
ClipRect(wrapBox);
boxWidth = wrapBox->right - wrapBox->left;
fixedMax = Long2Fix((long)boxWidth);
if (align == teFlushDefault)
   align = GetSysJust();
```

DETERMINE THE LINE HEIGHT

Now we need to determine the appropriate line height. NeoTextBox calls NTBLineHeight to perform this function, passing the text pointer, the text length, the wrap rectangle, the caller-specified line height code, and the address of curY, the current vertical pen location. NTBLineHeight calculates and returns the line height and calculates the correct starting pen location. Here's the NTBLineHeight function:

```
unsigned short NTBLineHeight(unsigned char *theText,
   unsigned long textLen, Rect *wrapBox, short lhCode, short *startY)
{
   short              asc, desc;
   FontInfo           fInfo;
   Point              frac;
   unsigned short     lineHeight;

   GetFontInfo(&fInfo);
   if (lhCode < 0) {
      /* lhCode < 0 means "variable line height", so if it's a */
      /* TrueType font use OutlineMetrics, otherwise use default. */
      frac.h = frac.v = 1;
      if (gHasTrueType && IsOutline(frac, frac)) {
         OutlineMetrics((short)textLen, theText, frac, frac, &asc,
            &desc, nil, nil, nil);
         lineHeight = MAXOF(fInfo.ascent, asc)
            + MAXOF(fInfo.descent, -desc) + fInfo.leading;
         *startY = wrapBox->top + MAXOF(fInfo.ascent, asc)
            + fInfo.leading;
      } else {
         /* Punt to "default" if we can't use TrueType. */
         lineHeight = fInfo.ascent + fInfo.descent + fInfo.leading;
         *startY = wrapBox->top + fInfo.ascent + fInfo.leading;
      }
   } else if (lhCode == 0) {
      /* lhCode == 0 means "default line height." */
      lineHeight = fInfo.ascent + fInfo.descent + fInfo.leading;
      *startY = wrapBox->top + fInfo.ascent + fInfo.leading;
   } else {
      /* lhCode > 0 means "use this line height" so we trust 'em. */
      lineHeight = lhCode;
      *startY = wrapBox->top + lhCode + fInfo.leading;
   }
   return(lineHeight);
}
```

Remember, there are three possible line height codes:

- Variable line height (specified by an lhCode less than 0) is handled first. If the TrueType trap is available and this particular font is a TrueType font, OutlineMetrics is called to determine the line height (see "Descent Into Hell"). OutlineMetrics can return a variety of information, but we really only want the highest ascent and the lowest descent, which are returned in the local variables asc and desc. Then we choose whichever is higher, the default ascent or asc, and whichever is lower, the default descent or desc. If TrueType isn't available or the particular font isn't a TrueType font, we punt to the default line height.
- If lhCode is 0, the default line height is used. This is defined as the sum of the ascent, descent, and line gap (leading) derived by a GetFontInfo call.
- Finally, if lhCode is greater than 0, the caller is providing a specific line height. In this case, NTBLineHeight returns lhCode as the line height.

Each of the three line height calculation methods also figures the correct startY based on the line height and wrapBox->top.

Back in NeoTextBox, we call NTBLineHeight to set up our local variables lineHeight and curY:

```
lineHeight = NTBLineHeight(theText, textLen, wrapBox, lhCode, &curY);
lineCount = 0;
lineStart = theText;
textEnd = theText + textLen;
textRemaining = textLen;
```

DESCENT INTO HELL

Descent is the amount of space that should be allocated for a font below the text baseline. When you call GetFontInfo, the value returned for descent is a positive number of points below the baseline. Although this is convenient, in the typographic industry it's more common to represent descent values as *negative* numbers.

In an attempt to be more typographically useful, TrueType's OutlineMetrics call returns its descent values as negative numbers. So, to avoid a descent into hell, remember to note the sign of descent values when mixing calls to GetFontInfo and OutlineMetrics.

Here we also set up some other local variables. The variable lineCount records the number of lines we've drawn. The pointer lineStart points to the beginning of the current line, which initially is the beginning of the text. The variable textEnd is a pointer to just beyond the end of the input text and is used for testing when the text is all used up. Finally, the variable textRemaining keeps track of how many bytes of input text remain to be processed.

THE BREAK-DRAW LOOP

Now NeoTextBox is ready to break lines and draw the text. This task is performed by the following do-while loop:

```
do {
   lineBytes = 1;
   wrapWid = fixedMax;

   breakCode = StyledLineBreak(lineStart, textRemaining, 0,
      textRemaining, 0, &wrapWid, &lineBytes);

   NTBDraw(breakCode, lineStart, lineBytes, wrapBox, align, curY,
      boxWidth);

   curY += lineHeight;
   lineStart += lineBytes;
   textRemaining -= lineBytes;
   lineCount++;
} while (lineStart < textEnd);
```

If this looks simple, that's because it is. Anyone who's tried to write code to wrap text knows that it's a difficult task. Making the algorithm compatible with different script systems complicates the matter even more. Fortunately, we have the Script Manager, which in this case makes our lives a *lot* easier.

The workhorse: StyledLineBreak. First we set lineBytes to 1, signaling to StyledLineBreak that this is the first "script run" on this line. Since we have only one script run, we always reset lineBytes at the top of the loop. Also, we reset wrapWid to be fixedMax (which was previously initialized to the fixed-point width of the wrap rectangle). WrapWid tells StyledLineBreak the width within which to wrap the text and returns how much of the line is left (if any) after wrapping (that's why we have to reset it at the top of the loop each time).

Now we call StyledLineBreak. We provide a pointer to the beginning of the text for this line, the number of bytes of text remaining, the wrap width, and the address of a variable where StyledLineBreak puts the number of bytes in this line. StyledLineBreak does the hard work of finding word boundaries, adding up character widths, and handling special cases, all in an internationally compatible way.

After StyledLineBreak returns, lineBytes tells us the length of the current line beginning at lineStart, and breakCode has a line break code that tells us the circumstances of the line break, as shown in Figure 2.

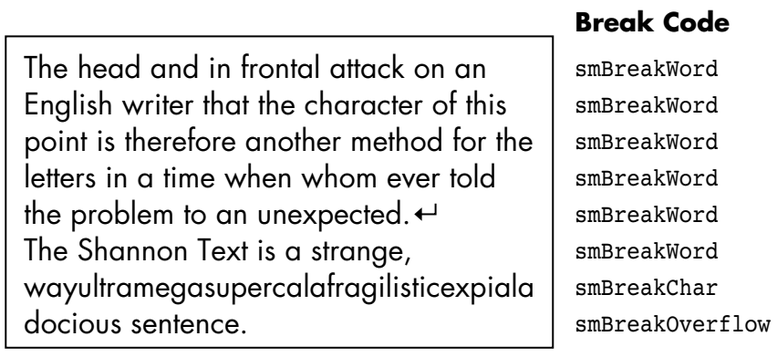


Figure 2
Line Break Codes

Usually, StyledLineBreak returns smBreakWord, indicating that it broke the line on a word boundary. The break code smBreakChar says that it encountered a word that was too long to fit on a single line and was forced to break in the middle of a word. StyledLineBreak returns smBreakOverflow if you run out of text before filling the given width. These line break codes help determine how to draw the text.

Draw the text with NTBDraw. After StyledLineBreak figures the length of the line, NeoTextBox calls NTBDraw to draw the line. NeoTextBox passes a pointer to the line of text, the length of the line in bytes, the wrap rectangle, the alignment, the current vertical pen location, and the width of the wrap rectangle. Let's take a look at NTBDraw:

```
#define kReturnChar    0x0d

void NTBDraw(StyledLineBreakCode breakCode, unsigned char *lineStart,
   long lineBytes, Rect *wrapBox, short align, short curY,
   short boxWidth)
{
   unsigned long   blackLen;    /* Length of non-white characters */
   short           slop;        /* Number of pixels of slop for */
                                /* full justification */

   blackLen = VisibleLength(lineStart, lineBytes);
```

```
   if (align == ntbJustFull) {
      slop = boxWidth - TextWidth(lineStart, 0, blackLen);
      MoveTo(wrapBox->left, curY);
      if (breakCode == smBreakOverflow ||
            *(lineStart + (lineBytes - 1)) == kReturnChar)
         align = GetSysJust();
      else DrawJust(lineStart, blackLen, slop);
   }
   switch(align) {
      case teFlushLeft:
      case teFlushDefault:
         MoveTo(wrapBox->left, curY);
         break;
      case teFlushRight:
         MoveTo(wrapBox->right - TextWidth(lineStart, 0,
            blackLen), curY);
         break;
      case teCenter:
         MoveTo(wrapBox->left + (boxWidth - TextWidth(lineStart, 0,
            blackLen)) / 2, curY);
         break;
   }
   if (align != ntbJustFull)
      DrawText(lineStart, 0, lineBytes);
}
```

NTBDraw's job is to move the pen and draw the text as indicated by the alignment parameter, align, and the line break code, breakCode. NTBDraw first calculates the visible length of the line with a call to the Script Manager routine VisibleLength. This excludes white-space characters at the end of the line. What exactly are white-space characters? Well, that depends on the script. VisibleLength knows which characters are visible and which are not for the current script, and returns an appropriate length in bytes, which is stored in the local variable blackLen.

When align is ntbJustFull, we need to determine whether the current line has a carriage return character ($0D) at the end, because a line with a carriage return (for example, the last line in a paragraph) should always be drawn with the default system alignment, rather than fully justified.

Looking back at the break codes for different types of lines shown in Figure 2, notice that the line that ends with the carriage return (denoted graphically in the illustration) returns a line break code of smBreakWord, where you might expect it to return smBreakOverflow. As you can see, StyledLineBreak expects the caller to know when a line is the last line of a paragraph. Therefore, every line whose break code is smBreakWord must be checked for a carriage return.

NTBDraw looks at the last byte in the line it's drawing to see if it's a carriage return. Since the carriage return character ($0D) falls into the control-code range, it's guaranteed never to occur as the low byte of a two-byte character. This frees us from having to test whether the last character in the line is two-byte and allows us to proceed directly to the last byte.

We now know whether the current line has a carriage return or not. If not, we calculate the amount of white-space slop remaining in the line, then call the Script Manager routine DrawJust to draw the text fully justified—whatever that means for this script. (In Arabic, for instance, full justification is performed completely differently than for Roman text.) If the current line *does* end in a carriage return, we override the align parameter with the default system alignment and fall through.

For the left, right, and center alignments, the switch statement moves the pen appropriately, and a DrawText call is made to draw the text. The visible length (in blackLen) helps correctly calculate the pen position for right and center alignment and full justification.

Update the variables. After NTBDraw returns, we need to update a bunch of local variables and loop around again.

```
   curY += lineHeight;
   lineStart += lineBytes;
   textRemaining -= lineBytes;
   lineCount++;
} while (lineStart < textEnd);
```

First, we add lineHeight to curY, setting us up for the next line. LineStart, the pointer to the beginning of a line, gets updated to the character after the end of the current line. TextRemaining gets reduced by the number of bytes consumed by the current line, and lineCount gets incremented. If lineStart still hasn't run off the end of the text, the whole break-draw process is repeated.

RETURN SOME VALUES

Now that NeoTextBox has done such a fine job wrapping the text, it's time to return some useful values to the caller.

```
if (endY)
   *endY = curY - lineHeight;
if (lhUsed)
   *lhUsed = lineHeight;
```

NeoTextBox returns these values only if the caller wants them. This makes it easy to get TextBox-like behavior from NeoTextBox without having to do any work: if you don't want a return value, just pass nil instead of providing the address of a variable.

CLEAN UP AND WE'RE DONE

The only thing left to do is a little cleanup, and we're outa here.

```
   SetClip(oldClip);
   DisposeRgn(oldClip);

   return(lineCount);
}
```

We restore the clipping region, dispose of our saved region, and return lineCount.

CALLS TO NEOTEXTBOX

One of the best features of NeoTextBox is that you can easily substitute it for calls you're currently making to TextBox. If that's all you want to do, replace every occurrence that looks like this

```
TextBox(textPtr, textLen, &wrapBox, justify);
```

with this

```
{
   EraseRect(&wrapBox);
   NeoTextBox(textPtr, textLen, &wrapBox, justify, 0, nil, nil);
}
```

To use NeoTextBox in place of TextBox, you pass 0 for lhCode (default line height) and nil for endY and lhUsed, and ignore the return value. If you add NeoTextBox to your program and just do the substitution above, every NeoTextBox call will be on the average 33% faster than the old TextBox call. If you use TextBox a lot, that can mean a real performance increase.

You can use NeoTextBox in more ways than just as direct substitution to improve performance. It does, after all, have whizzy new features that TextBox never had. Let's take a look at a more sophisticated call to NeoTextBox that uses some of its unique features:

```
short UseNTB(void)
{
   Rect        wrapBox;
   RGBColor    ltBlue;
   Handle      textHdl;
   long        textLen;
   short       numLines = 0;
   short       endY, lineHt;
```

```
   /* Set up our RGBColor and wrapBox. */
   SetRect(&wrapBox, 10, 10, 110, 110);
   ltBlue.red = 39321;
   ltBlue.green = 52428;
   ltBlue.blue = 65535;

   /* Paint the background, then set up the port text parameters. */
   PenNormal();
   RGBForeColor(&ltBlue);
   PaintRect(&wrapBox);
   ForeColor(blackColor);
   TextFont(helvetica); TextSize(12);
   TextFace(0); TextMode(srcOr);

   /* Retrieve some text for us to draw. */
   textHdl = GetResource('TEXT', 128);
   if (textHdl) {
      textLen = GetHandleSize(textHdl);
      /* Be sure to lock the handle. NeoTextBox can move memory! */
      HLock(textHdl);

      /* Wrap text and set numLines, endY, and lineHt. */
      numLines = NeoTextBox(*textHdl, textLen, &wrapBox, ntbJustFull,
         18, &endY, &lineHt);
      HUnlock(textHdl);

      /* Beep if text overflows wrapBox. */
      if (endY > wrapBox.bottom)
         SysBeep(1);

      /* Prove we know where the text ended by drawing a line. */
      MoveTo(wrapBox.left, endY + lineHt);
      Line(20, 0);
   }
   return(numLines);
}
```

This sample function draws a 100-by-100-pixel box in light blue, then wraps text from a TEXT resource into the rectangle, ORing the text over the blue background. The text is fully justified 12-point Helvetica®, with 18-point line spacing. If the text overflows the box, a beep sounds. A small line is drawn at the baseline where subsequent text might be drawn.

Here's an example using NeoTextBox with variable line height and TrueType fonts:

```
void UseVariableLineHeight(Rect *wrapBox, short align)
{
   Boolean   oldPreferred, oldPreserve;
   Handle    textHdl;
   long      textLen;

   if (gHasTrueType) {
      oldPreferred = GetOutlinePreferred();
      oldPreserve = GetPreserveGlyph();
      SetOutlinePreferred(TRUE);
      SetPreserveGlyph(TRUE);
   }

   textHdl = GetResource('TEXT', 128);
   textLen = GetHandleSize(textHdl);
   HLock(textHdl);
   NeoTextBox(*textHdl, textLen, wrapBox, align, -1, nil, nil);
   HUnlock(textHdl);

   if (gHasTrueType) {
      SetOutlinePreferred(oldPreferred);
      SetPreserveGlyph(oldPreserve);
   }
}
```

Notice that we save the current settings of the Font Manager's OutlinePreferred and PreserveGlyph flags. This allows us to be transparent to the caller. By setting OutlinePreferred to TRUE, we are ensured of using TrueType fonts, even if bitmapped fonts are available. By setting PreserveGlyph to TRUE, we get the accurate glyph shapes and measurements (see "SetPreserveGlyph With TrueType Fonts" on page 34). Calling NeoTextBox with -1 as its lhCode parameter causes it to use variable line height, which results in the difference shown in Figure 3.

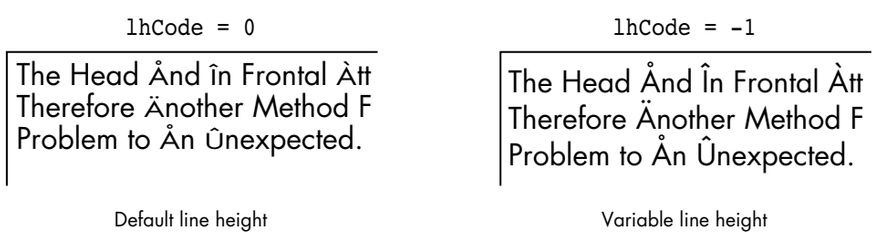


Figure 3
Using Variable Line Height

LIMITATIONS AND POSSIBLE ENHANCEMENTS

NeoTextBox is a nice alternative to TextBox, but it has its limitations and areas that could benefit from improvement. Following are some suggestions for overcoming the limitations and adding useful features.

32K TEXT SIZE LIMIT

All you *War and Peace* fans out there need to do a little work. NeoTextBox shares the 32K text limitation that TextBox has, but not for the same reason. TextBox can wrap only 32K of text in one call because it uses TextEdit. In NeoTextBox, the limitation arises from the OutlineMetrics call, which is used in deriving variable line height and can only handle 32K of text. Heavy-duty Tolstoy types could remove the code that implements variable line height and subsequently word wrap most novels in a single NeoTextBox call (knock yourselves out).

DON'T FORGET TO ERASE

Perhaps this isn't really a limitation, but you can't simply replace a TextBox call with a NeoTextBox call. You need to call EraseRect explicitly if you want TextBox behavior, as shown earlier in the section "Calls to NeoTextBox."

SCREEN-ONLY OPTIMIZATIONS

If you know you'll be using NeoTextBox only for screen applications (that is, you won't be using it to draw into a printer port), you can make a few optimizations. If you don't care about the return values, you can use RectInRgn to check whether the wrap rectangle intersects with the current port's visRgn; if it doesn't, you can simply return.

If you don't need the return value giving the number of total lines, you can make the break-draw loop terminate when curY exceeds wrapBox->bottom + lineHeight.

SPECIAL ONE-LINE CASE

In Macintosh computers with 256K ROMs, TextBox has a feature that might be a worthwhile addition to NeoTextBox. If the TextWidth of the input text is less than boxWidth, simply use DrawText to draw the text and don't bother with any of the wrapping code. TextBox has this feature because it's used for dialog box statText items, which are often one line.

DON'T DRAW OFF THE END OF WRAPBOX

It might make NeoTextBox faster if NTBDraw isn't called when curY is greater than wrapBox->bottom + lineHeight. You'd still have to wrap all the text (to determine the total number of lines), but you wouldn't be drawing text that you know will be clipped.

MAKE SAVING/RESTORING THE CLIPPING REGION OPTIONAL

It might be useful to be able to set up some complex clipping region and have NeoTextBox wrap as usual but clip its text to whatever the clipping region is set to at invocation. You could add a Boolean swapClip parameter to control this.

STYLED NEOTEXTBOX

With considerable effort, NeoTextBox could be extended to handle styled and multiscript text. Since StyledLineBreak, the workhorse of NeoTextBox, is designed to be used with styled text, such an enhancement is possible.

CONCLUSION

Once you start using NeoTextBox, you'll find it ending up in all your applications. In tests on a Macintosh IIfx running System 7, NeoTextBox was between 25% and 50% faster than TextBox, 33% faster on the average. Performance varies depending on font, screen depth, and the ratio of wrapping to drawing. For left-aligned Geneva text on an 8-bit screen, NeoTextBox is 40% faster than TextBox. That alone is a good reason to use it. Plus, it has features you can't get out of TextBox at all.

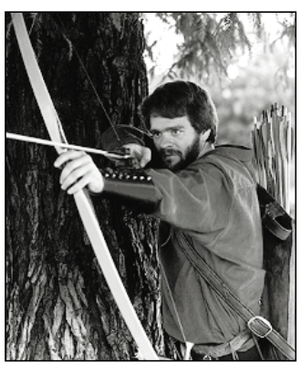
Perhaps the moral of this article is if you don't like some feature of the Toolbox or OS go ahead and write your own. But you'll be doing yourself a favor—and you'll be a lot more compatible in the future—if you can find lower-level system, Toolbox, or OS facilities to aid you in your task, rather than recoding the entire feature yourself.

So go ahead and whip NeoTextBox into your application. Enjoy the improved performance and new features. And if there's something you don't like, go right in there and change it. Make NeoTextBox the TextBox *you've* always wanted!

THANKS TO OUR TECHNICAL REVIEWERS
Sue Bartalo, John Harvey, Joe Ternasky•

MAKING YOUR MACINTOSH SOUND LIKE AN ECHO BOX

Happy notes for sound buffs: As you'll see from the sample code provided on the Developer CD Series disc, you can make your Macintosh play and record sounds at the same time, simply by using double buffering to record into one buffer while playing a second buffer, and then flipping between the buffers. If you want to take things a few steps further, pull out elements of this code and tailor them to suit your own acoustic needs.



RICH COLLYER

We all know that the Macintosh is a sound machine, so to speak, but with a little clever programming you can turn it into an echo box as well. The sample 2BufRecordToBufCmd included on the *Developer CD Series* disc is just a small application (sans interface) that demonstrates one way to record sounds at the same time that you're playing them. There are other ways to achieve the same goal, but my purpose is to educate you about the Sound Manager, not to lead you down the definitive road to becoming your own recording studio.

In addition to the main routine, 2BufRecordToBufCmd includes various setup routines and a completion routine. For easy reading, I've left out any unnecessary code out of this article.

CONSTANT COMMENTS

Before I get into the sample code itself, here are a few of the constants you'll run into in the application.

GETTING A HANDLE ON IT

The kMilliSecondsOfSound constant is used to declare how many milliseconds of sound the application should record before it starts to play back. The smaller the number of milliseconds, the more quickly the sound is played back. This constant is used to calculate the size of the 'snd ' buffer handles (just the data). Depending on the sound effect you're after, kMilliSecondsOfSound can range from 50 milliseconds to 400,000 or so. If you set it below 50, you risk problems: there may not be enough time for the completion routine to finish executing before it's called again. On the high end of the range, only the application's available memory limits the size. The

RICH COLLYER is just your run-of-the-mill three-year Developer Technical Support veteran: He's often heard screaming at his computer to the soothing accompaniment of Blazy and Bob on KOME radio, he's honed his archery skills to a fine point dodging (and casting) the slings and arrows at Apple, and he actually admits to a degree from Cal Poly with a specialty in computational fluid dynamics. We let you in on his outdoor adventures last time he wrote for us and he claims most of his indoor adventures aren't appropriate *develop* material, but we have it on good authority that he lives with carnivorous animals, if that's any clue. He's also a confirmed laserdisc and CD addict; he keeps promising to start a recovery program for those of us with the same affliction just as soon as he finishes writing that next sample . . .•

smaller the value, of course, the faster the buffers fill up and play back, and the faster an echo effect you'll get. A millisecond value of 1000 provides a one-second delay between record and echo, which I've found is good for general use. You'll want to experiment to find the effect you like. (Beware of feedback, both from your machine and from anyone who's in close enough proximity to "enjoy" the experimentation secondhand.)

YOUR HEAD SIZE, AND OTHER #DEFINES

The next three constants (kBaseHeaderSize, kSynthSize, and kCmdSize) are used to parse the sound header buffers in the routine FindHeaderSize. kBaseHeaderSize is the number of bytes at the top of all 'snd ' headers that aren't needed in the application itself. While the number of bytes isn't really of interest here, you need to parse the header in order to find the part of the sound header that you'll pass to the bufferCmd. How much you parse off the top is determined by the format of the header and the type of file; for the purposes of this code, however, all you need to be concerned with are the 'snd ' resources. The second constant, kSynthSize, is the size of one 'snth'. In the calculations of the header, I find out how many 'snth's there are, and multiply that number by kSynthSize. The last constant, kCmdSize, is the size of one command, which is used in the same way as kSynthSize. (These equations are derived from *Inside Macintosh* Volume VI, page 22-20.)

ERROR CHECKING WITH EXITWITHMESSAGE

2BufRecordToBufCmd includes error checking, but only as a placeholder for future commercialization of the product. If the present code detects an error, it calls the ExitWithMessage routine, which displays a dialog box that tells you more or less where the error occurred and what the error was. Closing this dialog box quits the application, at which point you have to start over again. Note that calling ExitWithMessage at interrupt time could be fatal, since it uses routines that might move memory. For errors that could occur at interrupt time, DebugStr is used instead.

USING THE SOUND INPUT DRIVER

Use of the sound input driver is fairly well documented in *Inside Macintosh* Volume VI, Chapter 22 (pages 22-58 through 22-68 and 22-92 through 22-99), but here's a little overview of what 2BufRecordToBufCmd does at this point in the routine, and why. When you use sound input calls at the low level (not using SndRecord or SndRecordToFile), you need to open the sound input driver. This section of the code just opens the driver, which the user selects via the sound cdev.

```
gError = SPBOpenDevice (kDefaultDriver, siWritePermission, &gSoundRefNum);
```

To open the driver, you call SPBOpenDevice and pass in a couple of simple parameters. The first parameter is a driver name. It doesn't really matter what the name of the driver is; it simply needs to be the user-selected driver, so the code passes

GESTALT YOUR MACHINE

You do need to check two rather critical sound attributes for 2BufRecordToBufCmd. First of all, your machine must have a sound input driver. There's very little point in trying to record sounds if the sample is being run on a machine that doesn't have sound input capabilities. Checking bit 5 of the returned feature variable with the Gestalt Manager will give you this handy bit of information.

Second, your hardware needs to support stereo sound, since you need one channel for sound input and one for sound output. Check for this attribute by checking bit 0 of the returned feature variable.

The following code shows how you can test all of the bits returned in the feature variable. (I didn't use this code in my sample.)

```
err = Gestalt (gestaltSoundAttr, &feature);
if (!err) {
   if (feature & (1 << gestaltStereoCapability))
      //This Macintosh Supports Stereo (test bit 0)
   if (feature & (1 << gestaltStereoMixing))
      //This Macintosh Supports Stereo Mixing (test bit 1)
   if (feature & (1 << gestaltSoundIOMgrPresent))
      //This Macintosh Has the New Sound Manager (test bit 3)
   if (feature & (1 << gestaltBuiltInSoundInput))
      //This Macintosh Has Built-in Sound Input (test bit 4)
   if (feature & (1 << gestaltHasSoundInputDevice))
      //This Macintosh Supports Sound Input (test bit 5)
   }
```

in nil (which is what kDefaultDriver translates into). The constant siWritePermission tells the driver you'd like read/write permission to the sound input driver. This will enable the application to actually use the recording calls. The last parameter is the gSoundRefNum. This parameter is needed later in the sample so that you can ask specific questions about the driver that's open. The error checking is just to make sure that nothing went wrong; if something did go wrong, the code goes to ExitWithMessage, and then the sample quits.

```
gError = SPBSetDeviceInfo (gSoundRefNum, siContinuous, (Ptr) &contOnOff);
```

Continuous recording is activated here to avoid a "feature" of the new Macintosh Quadra 700 and 900 that gives you a slowly increasing ramp of the sound input levels to their normal levels each time you call SPBRecord. The result in

2BufRecordToBufCmd is a pause and gradual increase in the sound volume between buffers as the buffers are being played. Continuous recording gives you this ramp only on the first buffer, where it's almost unnoticeable.

BUILDING 'SND ' BUFFERS

Now that the sound input driver is open, the code can get the information it needs to build the 'snd ' buffers. As its name implies, 2BufRecordToBufCmd uses two buffers. The reason is sound (no pun intended): The code basically uses a double-buffer method to record and play the buffers. The code doesn't tell the machine to start to play the sound until the recording completion routine has been called, so you don't have to worry about playing a buffer before it has been filled with recorded data. The code also does not restart the recording until the previous buffer has started to play.

INFORMATION, PLEASE

To build the sound headers, you need to get some information from the sound input driver about how the sound data will be recorded and stored. That's the function of the GetSoundDeviceInfo routine, which looks for information about the SampleRate (the number of samples per second at which the sound is recorded), the SampleSize (the sample size of the sound being recorded—8 bits per sample is normal), the CompressionType (see "Putting on the Squeeze"), the NumberChannels (the number of sound input channels, normally 1), and the DeviceBufferInfo (the size of the internal buffers).

This code (minus the error checking) extracts these values from the sound input driver.

```
gError = SPBGetDeviceInfo (gSoundRefNum, siSampleRate,
   (Ptr) &gSampleRate);

gError = SPBGetDeviceInfo (gSoundRefNum, siSampleSize,
   (Ptr) &gSampleSize);

gError = SPBGetDeviceInfo (gSoundRefNum, siCompressionType,
   (Ptr) &gCompression);

gError = SPBGetDeviceInfo (gSoundRefNum, siNumberChannels,
   (Ptr) &gNumberOfChannels);

gError = SPBGetDeviceInfo (gSoundRefNum, siDeviceBufferInfo,
   (Ptr) &gInternalBuffer);

value = kMilliSecondsOfSound;
gError = SPBMillisecondsToBytes (gSoundRefNum, &value);
gSampleAreaSize = (value / gInternalBuffer) * gInternalBuffer;
```

PUTTING ON THE SQUEEZE

If you want to use compression for 2BufRecordToBufCmd, keep in mind that the Sound Manager basically supports three types of sound compression: none at all, which is what I'm using, and MAC3 and MAC6, which are Mace compression types for 3:1 and 6:1 compression, respectively.

If you set the compression, the sound data is compressed after the interrupt routine is called (if you have one) and before the Sound Manager internal buffers are moved to the application's sound buffers.

You have a couple of options for playing back a compressed sound. Either the bufferCmd or SndPlay will decompress the sounds on the fly. If you need to decompress a sound yourself, you'll want to call the Sound Manager routine Exp1to3 or Exp1to6 (depending on the compression you were using).

Opening the sound input driver gives you the gSoundRefNum. The values siSampleRate, siSampleSize, siCompressionType, siNumberChannels, and siDeviceBufferInfo are constants defined in the SoundInput.h file; these constants tell the SPBGetDeviceInfo call what information you want. The last parameter is a pointer to a global variable. The SPBGetDeviceInfo call uses this parameter to return the requested information.

The last bit of work the code needs to do before it's ready to start building the 'snd ' headers is to convert the constant kMilliSecondsOfSound to the sample size of the buffer. To do this, the routine needs to call SPBMillisecondsToBytes and then round down the resulting value to a multiple of the size of the internal sound buffer. This is to bypass a bug connected with the continuous recording feature of Apple's built-in sound input device, which will collect garbage rather than audio data if the recording buffer is not a multiple of the device's internal buffer. Creating a buffer of the right size not only avoids this problem, but also enables the input device to more efficiently record data into your buffer.

Now the code has the information it needs to build the sound buffers. To save code space, I've made a short routine that builds the buffers and their headers. All the code has to do is call this routine for each of the buffers it needs and pass in the appropriate data.

IT'S A SETUP

The first line of code in the SetupSounds routine is fairly obvious. It simply calls the Memory Manager to allocate the requested handles, based on the known size of the data buffer and an estimated maximum size for the header, and does some error checking (see the code itself). Then, if the handle is good, the routine builds the 'snd ' header. Setting up the sound buffer requires building the header by making a simple call, SetupSndHeader, to the Sound Manager. There's a small problem with calling SetupSndHeader only once, however: When you call it, you don't know how big the

sound header is, so you just give the call the buffer, along with a 0 value for the buffer size. When the call returns with the header built, one of the values in the header—the one that's the number of bytes in the sample—will be wrong. (The header size will be correct, but the data in the header will not be.) To correct this, you simply wait until your recording is complete and then put the correct number of bytes directly into the header, at which time you'll know how much data there is to play back. The misinformation in the header won't affect your recording, only the playback.

Once the header's built, the code resets the size of the handle, moves the handle high (to avoid fragmentation of the heap), and locks it down. It's important to lock down the handles in this way; otherwise the Sound Manager will move the sound buffers it's working with out from under itself.

```
*bufferHandle = NewHandle (gSampleAreaSize + kEstimatedHeaderSize);

gError = SetupSndHeader (*bufferHandle, gNumberOfChannels, gSampleRate,
   gSampleSize, gCompression, kMiddleC, 0, headerSize);

SetHandleSize (*bufferHandle, (Size) *headerSize + gSampleAreaSize);
MoveHHi (*bufferHandle);
HLock (*bufferHandle);
```

TELLING IT WHERE TO GO

The next part of the program allocates and initializes a sound input parameter block, gRecordStruct. This structure tells the sound input call how to do what the code wants it to do.

The first instruction is obvious: it simply creates a new pointer into which the structure can be stored.

```
gRecordStruct = (SPBPtr) NewPtr (sizeof (SPB));
```

The recording call will need to know where it can find the open sound input driver, so next it needs the reference number to the driver (gSoundRefNum). The subsequent three lines of code inform the recording call how much buffer space it has to record into. Here, you could either give the call a count value, tell it how many milliseconds are available for recording, or give it the size of the sound buffer. For this code, it's easiest to just make the bufferLength the same as the count and ignore the milliseconds value. The code then tells the recording call where to put the sound data as it's recorded.

```
gRecordStruct->inRefNum = gSoundRefNum;
gRecordStruct->count = gSampleAreaSize;
gRecordStruct->milliseconds = 0;
```

```
gRecordStruct->bufferLength = gSampleAreaSize;
gRecordStruct->bufferPtr = (Ptr) ((*bufferHandle) + gHeaderLength);
gRecordStruct->completionRoutine = (ProcPtr) MyRecComp;
gRecordStruct->interruptRoutine = nil;
gRecordStruct->userLong = SetCurrentA5();
gRecordStruct->error = 0;
gRecordStruct->unused1 = 0;
```

The recording call also needs to know what to do when it's finished recording. Since the call is done asynchronously, it needs a completion routine. (I'll talk more about this routine later on.) You *could* leave out the completion routine and just poll the driver periodically to see if it's finished recording. To do that, you'd repeatedly call the routine SPBGetRecordStatus, and when the status routine informed you that recording was finished, you'd restart the recording and play the buffer that had just been filled. For this code, however, it's better to know as soon as possible when the recording is done because the more quickly you can restart the recording, the more likely you are to prevent pauses between recordings.

The userLong field is a good place to store 2BufRecordToBufCmd's A5 value, which you'll need in order to have access to the application's global variables from the completion routine. As you can see, the rest of the fields are set to 0. The code doesn't need an interrupt routine. There's also no point in passing an error back or using the unused1 field.

You'd need to use an interrupt routine if you wanted to change the recorded sound before compression, or before the completion routine was called (see "Routine Interruptions").

TIME TO CHANNEL

Just before the code jumps into the main loop, it needs to open a sound channel. This generally is not a big deal, but for 2BufRecordToBufCmd, I initialized the channel to use no interpolation.

ROUTINE INTERRUPTIONS

The interrupt routine gives you a chance to manipulate the sound data before any sound compression is done. For some of the operations that you may want to carry out inside the interrupt routine, you'll need access to the A5 world of the application, which is why I stored 2BufRecordToBufCmd's A5 value in the userLong field of gRecordStruct.

For more information about sound interrupt routines, take a look at *Inside Macintosh* Volume VI, page 22–63.

Warning: Don't try to accomplish too much in an interrupt routine. In general, you'll want interrupts to be minimal, and possibly written in assembly language, to avoid unnecessary compiler-generated code.

```
gError = SndNewChannel (&gChannel, sampledSynth, initNoInterp, nil);
```

Interpolation causes clicks between the sound buffers when they're played back to back, which can be a rather annoying addition to your recording (unless, of course, you're going for that samba beat).

JUST FOR THE RECORD

To start recording, all the code needs to do now is call the low-level recording routine, pass in gRecordStruct, and tell it that it wants the recording to occur asynchronously.

```
gError = SPBRecord (gRecordStruct, true);
```

LOOP THE LOOP

The main loop of this code is a simple while loop that waits until the mouse button is pressed or an error occurs in the recording, at which time the application quits.

```
/* main loop of the app */
while (!Button() || (gRecordStruct->error < noErr));
```

ROUTINE COMPLETION

You don't want a completion routine to do much, generally, since it's run at interrupt time and keeps your system locked up while it's running. There are three parts to this completion routine, one of which has four parts to itself.

The first part of the completion routine sets its A5 value to be the same as the A5 value of the application. This gives you access to the application's global variables from the completion routine.

```
storeA5 = SetA5 (inParamPtr->userLong);
```

If the completion routine weren't broken into two parts here, the MPW C compiler optimization scheme would cause a problem at this point: access to global arrays would be pointed to in an address register as an offset of A5 before you had a chance to set A5 to your application's A5 value, and you'd get garbage information. Therefore, it's necessary to restore your A5 value (part 1 of the completion routine) and then call the secondary completion routine to actually do all the work.

Before the routine does any work, it needs to make sure that there have not been any problems with the recording. If there were errors, the code drops out of the completion routine without doing anything.

```
if (gRecordStruct->error < 0)
   return;
```

Next the routine prepares the header of the buffer, which has just been filled, by correcting the header's length field. This field needs to be set to the count field of gRecordStruct, which now contains the actual number of bytes recorded.

```
header = (SoundHeaderPtr) (*(gBufferHandle[gWhichRecordBuffer]) +
   gHeaderSize);
header->length = gRecordStruct->count;
```

Once the header's been fixed, the code just sends the buffer handle off to the play routine to play the sound. (See "Play Time" for a full explanation of the play routine.)

```
PlayBuffer (gBufferHandle[gWhichRecordBuffer]);
```

The last part of the real completion routine prepares gRecordStruct to start the next recording. To do this, the code needs to select the correct buffer to record to and rebuild gRecordStruct to reflect any changes. The macro NextBuffer performs an XOR on the variable gWhichRecordBuffer to make it either 1 or 0. The changes include setting the correct buffer to record to and checking to see that the bufferLength is correct. Once the structure is reset, the code makes the next call to SPBRecord to restart the recording.

```
#define NextBuffer(x) (x ^= 1)

gWhichRecordBuffer = NextBuffer (gWhichRecordBuffer);
gRecordStruct->bufferPtr = (*(gBufferHandle[gWhichRecordBuffer]) +
   gDataStart);
gRecordStruct->milliseconds = 0;
gRecordStruct->count = gSampleAreaSize;
gRecordStruct->bufferLength = gSampleAreaSize;

err = SPBRecord (gRecordStruct, true);
```

The last piece of the completion routine resets A5 to what its value was when the routine started.

```
storeA5 = SetA5 (storeA5);
```

PLAY TIME

The code in the PlayBuffer routine is very simple Sound Manager code. All it does is set up the command parameters and call SndDoCommand. The routine needs to know what channel to play into and what buffer to play, so the code sets up the local sound structure by telling it which buffer to play, and sends that local structure to SndDoCommand along with the necessary channel information (gChannel). SndDoCommand then plays the sound. The last parameter in the SndDoCommand call, false, basically tells the Sound Manager to always insert the command in the

channel's queue: if the queue is full, SndDoCommand will wait until there's space to insert the command before returning.

```
localSndCmd.cmd = bufferCmd;
localSndCmd.param1 = 0;
localSndCmd.param2 = (long) ((*bufferHandle) + gHeaderSize);
gError = SndDoCommand (gChannel, &localSndCmd, false);
```

If you wanted to send the sounds to a different machine to be played, you could simply replace the code in the the PlayBuffer routine with IPC or Communications Toolbox calls telling a second machine to play the buffers.

CLEANING UP AFTER THE SHOW

Once the code finds the mouse button down or discovers that an error occurred in the recording and exits the main loop, there's only one last thing to do: clean up. The first part of cleaning up is to close the sound input driver. Before you can close the driver, you need to make sure it's not in use; the routine SPBStopRecording stops the recording.

```
gError = SPBStopRecording (gSoundRefNum);
SPBCloseDevice (gSoundRefNum);
```

Next you need to dispose of the handles and pointers you've been using. Before sending them on their way, however, you have to make sure that they have been allocated, so the code checks to see whether or not the handles and pointer are nil.

```
for (index = 0; index < kNumberOfBuffers; ++index)
   DisposeHandle (gBufferHandle[index]);
DisposePtr ((Ptr) gRecordStruct);
```

Last but not least, the code disposes of the sound channel for you. Setting the quitNow flag clears the sound queue before the channel is closed.

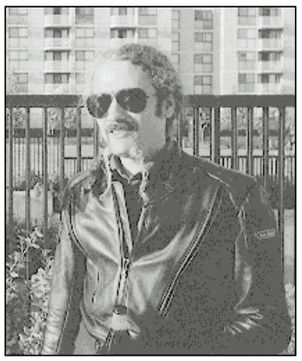
```
gError = SndDisposeChannel (gChannel, true);
```

COMPOSE YOURSELF

So now you know a little bit more about doing basic sound input at a low level. I've fielded many questions about clicks, pauses between buffers, and so on, which I've resolved and built into 2BufRecordToBufCmd. The specific techniques I've outlined here may not apply to what you're interested in doing right now, but if you're using the sound input driver or are interested in continuous recording, parts of this sample may be useful to you in some other application. You've heard the saying "take what you like and leave the rest"? Sound advice (so to speak).

THANKS TO OUR TECHNICAL REVIEWERS

Neil Day, Kip Olson, and Jim Reekes, who burned the midnight oil ripping this code to shreds and putting it back together again.•



BE OUR GUEST

BACKGROUND-ONLY APPLICATIONS IN SYSTEM 7

C. K. HAUN

One of the least heralded new features of System 7, but nonetheless a very important one, is full support for faceless background applications (FBAs). An FBA is a full-fledged application that's invisible to the user. It has its own event loop, and it receives time and some events like any other application, but it doesn't have a menu bar, windows, dialogs, or other graphic components. An FBA is a normal file of type 'APPL'.

FBAs are, by a stretch of the imagination, similar to UNIX® daemons. The purpose of an FBA is to provide services to other applications or to monitor the system. For instance, an application that periodically checks your hard drive for files that haven't been backed up lately is a perfect candidate for FBA status. Thus, an FBA can be a silent partner to your application, INIT, cdev, desk accessory, or driver.

An FBA is the best way to provide certain services. For example, an FBA paired with a desk accessory can enable the DA to send Apple events, something a DA cannot usually do. (See the AECDEV/AEDAEMON sample in the snippets provided with the DTS Sample Code on the *Developer CD Series* disc.) An FBA can replace an INIT that patches traps to get time and provides services, or it can replace a driver that depended on periodic run messages to operate. Converting to an FBA not only frees you from having to patch to get the time you need, but also gives you a fully supported and documented interface and design. You get all the advantages of a full application, without the overhead of a user interface.

An FBA can also be an application manager for a suite of applications. With an FBA, you can control the launching of and communication between applications, using LaunchApplication and Apple events.

Writing an FBA is simple. An FBA is a subset of a standard Macintosh application, consisting of a minimal event loop and the code to handle two types of events, null events and high-level events. No other events are sent to an FBA. This makes a great deal of sense, since every other event (keystroke, mouse click, and such) is designed for foreground applications.

The SmallDaemon backgrounder shell included on the *Developer CD Series* disc shows just how simple the basics of an FBA are:

```
Boolean         gQuit = false;
EventRecord     gERecord;
unsigned long   gMySleep = 50000; /* A long, long
                                     time */
main()
{
/* Routine for installing my Apple event
   handlers. Need to install the four required
   handlers, plus handlers for any other events
   I want to accept. */
   InitAEStuff();
   while (gQuit == false) {
   /* A normal call to WaitNextEvent. I want
      only highLevelEvents, since all other
      events relate to interface actions, and I
      have no interface. */
      if (WaitNextEvent(highLevelEventMask,
            &gERecord, gMySleep, 0)) {
         /* I'll get only null and high-level
            events. */
         switch (gERecord.what) {
            case nullEvent:
            /* No null processing in this
               sample. */
            break;
```


C. K. HAUN has been programming Apple computers since 1979, writing commercial education, utility, and game applications for the Apple II, IIGS, and Macintosh, with some occasional dark forays into the Big Blue world. (It paid the rent.) He currently works in Developer Technical Support and focuses mainly on Apple events and the Edition Manager. Besides working to provide the best possible support to developers, he's been trying to organize the Silicon Valley chapter of Heck's Moofers, a motorcycle club devoted to the precept that computer nerds on bikes can raise heck too, darn it. And yes, that really *is* his mustache.•

```
            case kHighLevelEvent:
            /* Get a high-level event (an Apple
               event) and process it. */
            DoHighLevel(&gERecord);
            break;
        }
    }
  }
}
```

As you can see, there's not much there. The first thing you'll notice is that an FBA doesn't start up any managers. All the managers you normally start are based on user interface actions. Thus, they should *not* be called in an FBA—in fact, calling them will cause your FBA to crash. There's one exception to this rule: you can initialize QuickDraw, but *only* to provide yourself with off-screen grafPorts or to use some QuickDraw functions. Do not do any actual screen drawing from an FBA.

You'll also notice that you don't pass a mouse region to WaitNextEvent. That's obvious, since you're never in the foreground and have no windows or mouse control to track. And the only events you'll be handling are null events and high-level events (Apple events).

The next step is to make sure the system knows that you're a background-only application. You do this with a SIZE resource, by setting the canBackground and onlyBackground bits to true. When the Finder launches your FBA, it checks these bits and finds that you're a background-only application.

Some tips and techniques to keep in mind:

- Always remember that an FBA is invisible to the user. An FBA does *not* show up in the Process menu or in the Finder About window. Also, its heap memory is added to the system size thermometer bar in the Finder About window, even though it has its own application heap. The user often will have no idea that it's running, so be friendly.
- Being friendly means yielding time. Have a very large sleep time when you're not actually doing some processing. If you don't, users will think that their machine has slowed down inexplicably. You can always use the new Process Manager call WakeUpProcess to get yourself back into fast processing when you need to.
- Being friendly also means making sure you're occupying the *smallest* heap size you can possibly run in. Again, users may never know that your FBA exists, so if you eat up a bunch of memory in your FBA, users will not understand why they've lost so much memory. Be conservative.
- Putting your FBA in the Startup Items folder in the System Folder is also a good idea. That will ensure that your FBA is launched right after Finder startup and will put the FBA high in the Process Manager's heap, preventing fragmentation of application memory space.
- Use WakeUpProcess to get your FBA running. Keep your sleep time at maximum normally, and when you need to start doing null processing (in response to an Apple event or PPC completion routine, for example) use the Process Manager to wake yourself up. WakeUpProcess can be called at interrupt time. Then, when you've finished processing, drop back to a maximum sleep time.

For further information and help with writing an FBA, refer to the Process Manager chapter of *Inside Macintosh* Volume VI. Then try it—you'll like it!

We welcome guest columns from readers who have something interesting or useful to say. Send your column idea or draft to Caroline Rose at Apple Computer, Inc., 20525 Mariani Avenue, M/S 75-2B, Cupertino, CA 95014 (AppleLink: CROSE).•

SIMPLE TEXT WINDOWS VIA THE TERMINAL MANAGER

A major feature of System 7 is the fact that the Macintosh Communications Toolbox is built in and available for all to use. This article presents code that uses the Communications Toolbox Terminal Manager to implement a simple text window that can be dropped into any application. The window is useful for displaying debugging or status information such as variable values or memory usage during application development.

CRAIG HOTCHKISS

Many times, when writing code, I've wished I had some sort of text repository, a place to write some quick text. I tried to use TextEdit for this at first, but its structures grow as you add text, so my memory accounting became confused. Sprinkling DebugStr calls throughout the code told me what I needed to know most of the time, but they were interruptive to both the user interface and timing-sensitive functions. Finally, I turned to the Terminal Manager because in addition to a terminal tool it contains the nuts and bolts necessary to display a text window and uses a fixed amount of memory.

TermWindow, included on the *Developer CD Series* disc, consists of a few simple routines that use the Terminal Manager and a terminal tool to display text in a Macintosh window. The package can be used in any application for purposes such as displaying variable values, heap checks, memory usage, and routine paths. I've even used some features of the terminal tool to grab attention, like having text blink when I encounter an OSErr or when memory begins to get tight.

Figure 1 shows a sample TermWindow terminal emulation window. I used the Developer Technical Support (DTS) Sample application (included with MPW) and the Apple VT102 tool to produce the display. The double-high/double-wide text shown in the figure is a feature of the VT102 emulator. I displayed it by writing the escape sequences shown on the next page to the window. (Two lines are needed for double-high text, one to display the top half of the line, and one for the bottom half.)

CRAIG HOTCHKISS works on the Connectivity team in Apple's system software group. When he's not pondering new ways for you to discover the world via your Macintosh, you might catch him practicing maneuvers with his stunt kite, playing chess, or "on his way" to a volleyball game. Before coming to Apple, Craig spent several years (in the great state of the world champion Twins) at the telephone company frustrated with DOS in preparation for database work on PDP and VAX machines.•

```
<ESC>#3HELP! I'm trapped in here!
<ESC>#4HELP! I'm trapped in here!
```

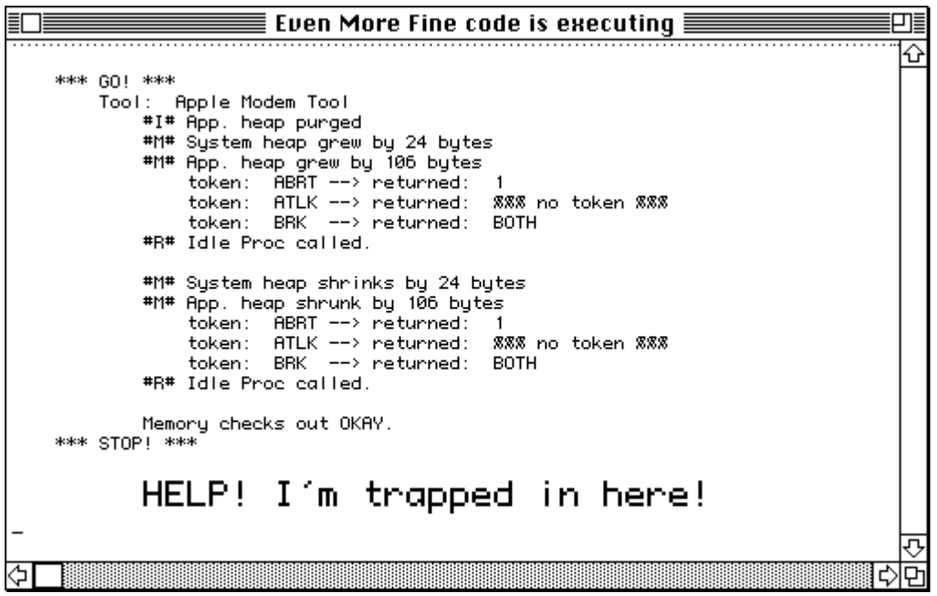


Figure 1
A TermWindow Window

THE HEADER FILE

The file TermWindow.h defines a basic TermWindowRec structure that TermWindow uses internally for housekeeping and for storage of other structures, including the handles for the terminal and control records.

```
struct TermWindowRec
{
   WindowRecord       fWindowRec;        // so it can be a WindowPtr
   short              fWindowType;       // for application use
   TermHandle         fTermHandle;       // CTB terminal handle
   TermEnvironRec     fTermEnvirons;     // environment info
   ControlHandle      fVertScroll;       // vertical scroller
   ControlHandle      fHorizScroll;      // horizontal scroller
   Point              fMinWindowSize;    // min. width/height of window
};
```

```
typedef struct  TermWindowRec TermWindowRec;
typedef         TermWindowRec *TermWindowPtr;
```

Six prototype routines, used to put TermWindow to work in an application, are also defined in the header file TermWindow.h.

```
pascal    OSErr     InitTermMgr(void);
pascal    OSErr     NewTermWindow(TermWindowPtr *termPtr,
                          const Rect *boundsRect,
                          ConstStr255Param title,
                          Boolean visible,
                          short theProc,
                          WindowPtr behind,
                          Boolean goAwayFlag,
                          Str31 toolName);
pascal    OSErr     DisposeTermWindow(TermWindowPtr termPtr);
pascal    Boolean   IsTermWindowEvent(TermWindowPtr termPtr,
                          const EventRecord *theEventPtr);
pascal    void      HandleTermWindowEvent(TermWindowPtr termPtr,
                          const EventRecord *theEventPtr);
pascal    OSErr     WriteToTermWindow(TermWindowPtr termPtr,
                          Ptr theData, Size *lengthOfData);
```

HOW TO USE TERMWINDOW

The six TermWindow routines are easy to use. After normal Macintosh manager initialization, you'll initialize the Terminal Manager with a call to InitTermMgr and then call NewTermWindow. The NewTermWindow function allocates the TermWindowPtr, terminal handle, and control handles. It also creates a Macintosh window complete with scroll bars and then attaches the terminal emulation region to the window with a call to TMNew. (See the next section for more on initialization.)

There are two functions to handle events, IsTermWindowEvent and HandleTermWindowEvent. These should be placed in your application event loop. IsTermWindowEvent determines whether the incoming event is for the emulation window by looking at the EventRecord structure, and it also provides time to the terminal emulator by calling TMIdle. HandleTermWindowEvent is a dispatcher that routes the event to routines that in turn call the Terminal Manager to process the event. These routines are discussed in more detail in the section "Handling Events."

The WriteToTermWindow routine, discussed later in this article under "Writing Text," uses the Terminal Manager to display your data in the terminal emulation window. And finally, DisposeTermWindow performs window and structure cleanup.

INITIALIZATION

The InitTermMgr routine prepares to initialize the Terminal Manager by checking the status of three Gestalt selectors: gestaltCTBVersion, gestaltCRMAttr, and gestaltTermMgrAttr. (We don't have to check for Gestalt, since MPW 3.2 contains the code to make Gestalt work.) The gestaltCTBVersion selector tells us which version of the Communications Toolbox is available, thereby letting us know that it exists. The gestaltCRMAttr and gestaltTermMgrAttr selectors tell us, respectively, whether the Communications Resource Manager (which must be initialized for tool resource handling) and Terminal Manager are available for use. InitTermMgr then calls the Communications Toolbox initialization routines if each Gestalt call returns a value of true. It all looks like this:

```
pascal    OSErr  InitTermMgr(void)
{
   OSErr        result = noErr;
   Boolean      hasCTB, hasCRM, hasTM;
   long         gestaltResult;

   hasCTB = (Gestalt(gestaltCTBVersion, &gestaltResult) ?
                           false : gestaltResult > 0);
   hasCRM = (Gestalt(gestaltCRMAttr, &gestaltResult) ?
                           false : gestaltResult != 0);
   hasTM = (Gestalt(gestaltTermMgrAttr, &gestaltResult) ?
                           false : gestaltResult != 0);
   if ((hasCTB) && (hasCRM) && (hasTM))
      if (noErr == (result = InitCRM()))
         if (noErr == (result = InitCTBUtilities()))
            if (noErr == (result = InitTM()))

   return (result);
} /*InitTermMgr*/
```

You may wonder whether the Communications Toolbox requires that the Macintosh Toolbox managers be started up at initialization time. The Communications Toolbox managers are loaded in the system heap, so you may have other reasons for starting them up in your initialization routine, but TermWindow's only requirement is that InitTermMgr be called at some point before NewTermWindow. Because the NewTermWindow routine has the potential to allocate nonrelocatable memory, calling InitTermMgr and NewTermWindow at initialization removes the possibility of heap fragmentation.

NewTermWindow begins by validating each parameter that was passed and assigns default values if necessary (see Table 1 below; refer to "The Header File" earlier in this article for the NewTermWindow declaration). You might notice that a good deal of the parameter list to NewTermWindow is very similar to that for the NewWindow

function in the Macintosh Toolbox. The NewTermWindow parameter list is designed to provide as much window control as possible when calling NewWindow, while also adding the functionality for the terminal emulation region. Calls to NewControl attach scroll bars to the window being created.

Table 1
NewTermWindow Defaults

Parameter Name	Default Value
termPtr	Pointer allocated for TermWindow storage
boundsRect	FrontWindow window size
title	"\pStatus"
visible	True
theProc	ZoomDocProc
behind	FrontWindow
goAwayFlag	False
toolName	TMChoose user setup dialog box

Once the Macintosh window is ready, NewTermWindow attaches the terminal emulation region to the window with a call to TMNew. Parameters to the TMNew routine tell the terminal tool, via the Terminal Manager, how to set up the emulation. (Terminal tools, not the Terminal Manager, implement the emulation.) The basic TMNew prototype is as follows:

```
pascal TermHandle TMNew(const Rect *termRect, const Rect *viewRect,
         TMFlags flags, short procID, WindowPtr owner,
         TerminalSendProcPtr sendProc,
         TerminalCacheProcPtr cacheProc,
         TerminalBreakProcPtr breakProc,
         TerminalClikLoopProcPtr clikLoop,
         TerminalEnvironsProcPtr environsProc,
         long refCon, long userData);
```

In TermWindow's case, NewTermWindow sets termRect and viewRect to the portRect of the window's grafPort minus the scroll bar area and sets the flags parameter to 0L. (This enables the terminal tool to put up custom menus or provide error alerts to the user.) The procID parameter is a terminal tool reference number (obtained with TMGetProcID) that tells the Terminal Manager which tool to use. The owner parameter is set to the WindowPtr of our Macintosh window. The procedure pointers, refCon, and userData are all set to nil or 0L.

```
termPtr->fTermHandle = TMNew(&termRect, &termRect, 0L, procID,
                          (WindowPtr)(*termPtr), nil, nil, nil,
                          nil, nil, 0L, 0L);
```

HANDLING EVENTS

Your application's main event loop should use the two event-handling routines, IsTermWindowEvent and HandleTermWindowEvent, to process events for the emulation window and to determine whether the event has already been handled. I use the following fragment just after calling WaitNextEvent; it sets the gotEvent flag to false when TermWindow has processed the event, so that I don't try to handle the event twice.

```
if (IsTermWindowEvent(&gTheEvent, gTermWindowPtr)) {
   HandleTermWindowEvent(&gTheEvent, gTermWindowPtr);
   gotEvent = false;
}
```

IsTermWindowEvent uses FindWindow or the message field of the event record to determine whether the event is for the terminal window. (See *Inside Macintosh* Volume I, page 250, for details.) IsTermWindowEvent is also a convenient place to give the terminal tool idle time; it calls TMIdle to give the tool a chance to draw text or blink the cursor. (Some terminal tools also have the ability to display blinking text; that would be done here also.)

The HandleTermWindowEvent routine is fairly straightforward, especially if you've written window- and scroll-handling code before. As is true when handling normal windows, the what field of the event record defines the work to be done. Terminal Manager routines exist for most of this work, so handling events is primarily a matter of calling the right routine at the appropriate time. For example, window activation and deactivation are communicated to the tool with a call to TMActivate.

```
TMActivate(termPtr->fTermHandle,
         (0 != (theEventPtr->modifiers & activeFlag)));
```

Figure 2 illustrates how a message like TMActivate is routed to accomplish its goal. The Terminal Manager receives the TMActivate call and routes the tmActivateMsg message to the terminal tool. The terminal tool then takes the opportunity to call Macintosh Toolbox routines such as InsertMenu or RemoveMenu (if the tool uses a custom menu, as the VT102 tool does) to keep the screen up to date.

Update events are handled by a call to TMUpdate sandwiched between BeginUpdate and EndUpdate. You'll just need to pass TMUpdate the update region freshly calculated by BeginUpdate. Of course, if you check for an empty region first, you won't have to call TMUpdate at all.

```
BeginUpdate((WindowPtr)termPtr);
   if (nil != ((WindowPtr)termPtr)->visRgn)
      TMUpdate(termPtr->fTermHandle, ((WindowPtr)termPtr)->visRgn);
EndUpdate((WindowPtr)termPtr);
```

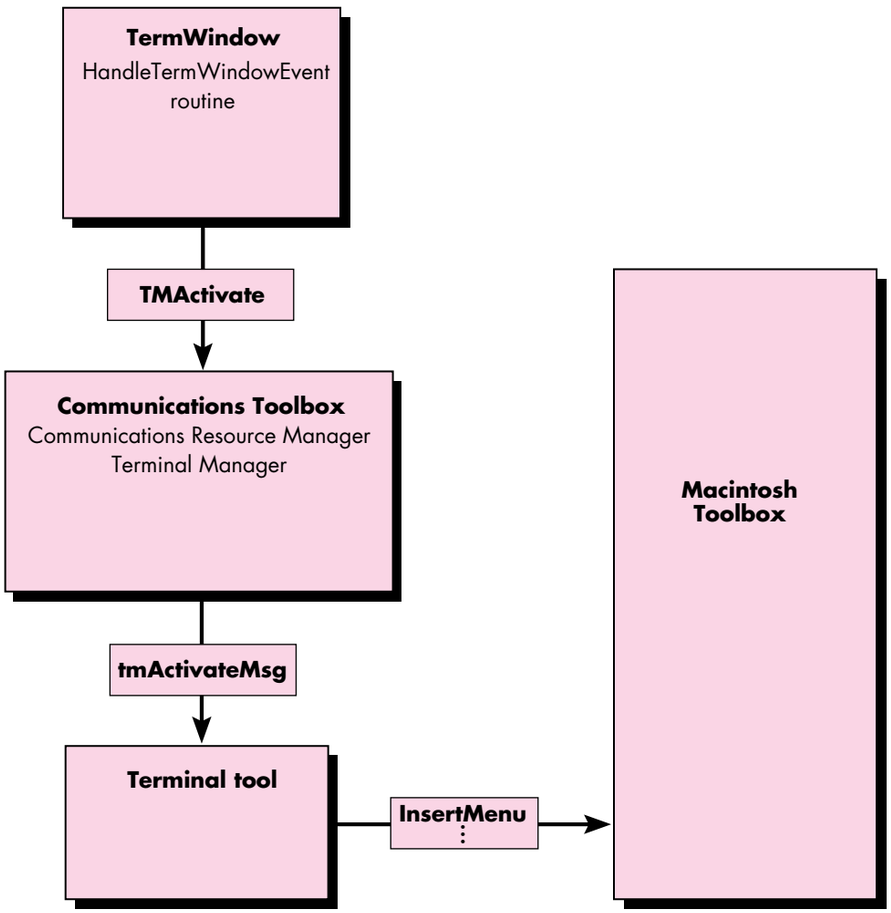


Figure 2
Calling TMActivate

Mouse-click handling is also fairly traditional. FindWindow is used to determine where in the window the click took place, and Terminal Manager routines are called to let the terminal tool know what to do. When a zoom or grow event causes the size of the emulation rectangle to change, the window's portRect gets passed to TMResize. If FindWindow returns inContent, FindControl is used to determine the control in which the click occurred, so that TermWindow will know whether to scroll horizontally or vertically. The partCode returned from FindControl tells how much to scroll by indicating the part of the control where the click took place. If

FindControl returns nil for the ControlHandle parameter, the click is in the emulation region and TMClick is called.

WRITING TEXT

Writing data is easy via the WriteToTermWindow routine. Here's an example of a WriteToTermWindow call with tempString declared as a char array:

```
sprintf(tempString, "Hello, world");
dataLength = strlen(tempString);
osResult = WriteToTermWindow(termPtr, tempString, &dataLength);
```

In a debugging situation, you might want to do something like the following to keep track of heap size:

```
gAppHeapRef -= FreeMem();
if (gAppHeapRef) {
   sprintf(tempString, "\t\t#M# App. heap grew by %ld bytes",
            gAppHeapRef);
   dataLength = strlen(tempString);
   osResult = WriteToTermWindow(termPtr, tempString, &dataLength);
}
```

WriteToTermWindow hands the data off to the terminal tool, via a call to TMStream. You might be tempted here to think that the data should appear in the window immediately, but it doesn't—it's simply put in the terminal tool buffer. The terminal tool waits for a TMUpdate or TMIdle before actually writing to the window. Another point to remember when working with terminal tools is that display fonts are controlled by the terminal tool; in fact, many use specific terminal fonts.

NOW IT'S YOUR TURN . . .

That's really all there is to using this simple text window. Now that you have some base code to work from, you might want to add the communication abilities needed to talk to another computer by using Connection Manager calls like CMRead or CMWrite and telling the terminal tool when to use them with procedure pointer parameters to TMNew. How about extending TermWindow to write all data displayed to a file? Or if you're really up to a challenge, try adding a scroll-back cache to store data that gets scrolled out of the emulator. Just scrolling the data around is not too difficult, but brush up on your region handling when you try to work with selections.

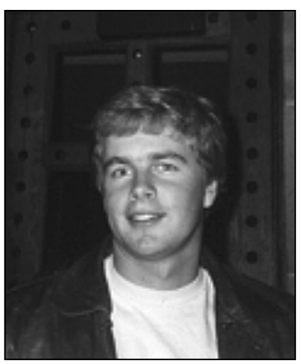
I hope you find the TermWindow package useful. Put it to work, add some features, and pass it on. Everything should evolve over time.

For more details on using the Macintosh Communications Toolbox, see *Inside the Macintosh Communications Toolbox,* by Apple Computer, Inc. (Addison-Wesley, 1991).•

THANKS TO OUR TECHNICAL REVIEWERS
James Beninghaus, Mary Chan, Byron Han•

TRACKS: A NEW TOOL FOR DEBUGGING DRIVERS

Here's a tool that gives you access to what you really need to know while debugging a driver. With Tracks, you decide what kind of information you want to track—variable contents, who called the current function, timing information, and more—all while your driver's running. When a problem arises, you can easily tell where your driver's been and what it's been doing, so you can find out just what went wrong.



BRAD LOWE

If you've ever written a device driver, you know how hard it is to keep track of what's going on. Learning the value of variables and other data as the driver runs usually requires a lot of time in a debugger.

When a driver crashes, trashing the stack in the process, it's often impossible to determine the last routine that was executed. Finding the bug can take many hours, especially if the crash appears only periodically. Even after you've found the bug, each crash requires recompiling, building, restarting, and retesting. Anything to help locate bugs more quickly and accurately could save a lot of time and frustration.

That's why Tracks was written. Whether you're writing your first or your fiftieth driver, it can help you track down those nasty bugs that always show up. The simple macros in Tracks make it easy to log all kinds of information from a driver written in C or C++. You can record strings, data blocks, longs, and even formatted data types. Tracks can write debugging information directly to disk as it comes in, or it can keep the information in a circular buffer and dump it to disk on command—a MacsBug dcmd (debugger command) lets you do this even after a crash.

You can completely control what information is logged, and your driver won't even know it. If you know a routine works, you can turn off calls from it at any time—including while your driver's running.

On the *Developer CD Series* disc, you'll find TestDrvr, a sample driver that demonstrates how to implement Tracks functionality in a simple (and useless) driver.

BRAD LOWE attends Chico State, where he claims to be majoring in fashion merchandising, although informed sources say he's been sighted frequently in computer science classes. An Eagle Scout, he enjoys hiking, skiing, mountain biking, and scuba diving. His newest toy is a paraglider, which he flies—on his free weekends—over obscure regions of Northern California.•

Also enclosed is the complete source for the Tracks utilities, as well as all the necessary support tools.

In the following sections, you'll find out about how Tracks works, what kind of information the Tracks macros log, and what the code does. You'll also get some pointers on installing and using Tracks. If you're eager to start using Tracks, take a look at "Tracks in Action."

HOW TRACKS WORKS

Tracks works somewhat like a message service that can accept telephone calls on 128 different lines from the target driver. You decide where to install the lines and what kinds of messages each line will deliver. You can control which lines to listen to (or not) and where to save incoming messages.

The invocations of macros—or calls—that send information to Tracks are called *tracepoints*. You assign each tracepoint a number between 0 and 127, called a *diagnostic ID (diagID)*, and a name. The diagID represents one bit in a 128-bit flag that can be set or cleared from the Tracks control panel device (cdev). When a tracepoint is encountered, data is logged only if the corresponding bit has been set.

Being able to set or clear tracepoints on the fly allows you to tailor the type of information being traced. By assigning a meaningful name to each tracepoint, you'll

TRACKS IN ACTION

TestDrvr is a simple driver skeleton that checks the status of the keyboard. If the Option key is down, it logs one type of data, and if the Command key is down, it logs another type of data.

To see Tracks in action, follow these condensed instructions:

1. Put DumpTracks into your MPW Tools folder and TestDrvr into your System Folder or Extensions folder.
2. Put Tracks into your System Folder or, in System 7, into your Control Panels folder.
3. Restart your Macintosh.
4. Open the Tracks cdev, click the Driver Name button, and locate the file TestDrvr.
5. Turn on Tracks and click the Level 1 button to turn on tracepoints 0-31 (only 0-3 are used).
6. Press the Command key or the Option key to begin to log data. The Bytes Buffered field should change.
7. Click Write Buffer to send TestDrvr output to disk. Only data written to disk can be examined.
8. Start up MPW and type "DumpTracks" to see what was just traced.

Look over the TestDrvr source code if you haven't already done so. Don't forget to remove TestDrvr when you're done. For information about the output from the example, see the section "Examining Tracks Output" under "Using Tracks."

Tracks was written by Jim Flood and Brad Lowe for Orion Network Systems, Inc., a subsidiary of Apple Computer, to help develop and debug Orion's SNA•ps Access driver, part of the SNA•ps product family. SNA•ps allows you to connect your Macintosh to an IBM SNA (System Network Architecture) network and communicate with SNA-based hosts, midrange systems (such as AS/400), and even personal computers.•

know which ones to set or clear, and the name of the tracepoint will be recorded with any Tracks output. Tracks *breakpoints* are tracepoints that will drop you into your debugger.

GROUPING INFORMATION

Because the diagID doesn't have to be unique, a tracepoint can represent a single Tracks call, a type of Tracks call, or a grouping of Tracks calls. A type of Tracks call, for instance, might be all error-reporting calls. A grouping might be all tracepoints in a particular routine.

This kind of flexibility allows you to group your information into logical and functional units. It's up to you to create as many or as few tracepoints as you need. For instance, if you're working on a new routine, you may set a whole bunch of Tracks calls all to the same diagID. When you test the routine, you can set some or all of the other switches to off and focus on the messages from that routine. Later, when you know the function works, you can keep that switch off.

Numbering for ease of use. There aren't any limits on how you group your diagIDs. You might assign all messages to one tracepoint or simply start at 0 and increment by 1 from there. The key is that once you know something works, you want to be able to turn off tracing in that area. By assigning unique diagIDs to groups of Tracks calls, you can quickly tailor your tracing.

For convenience, there are four groups of 32 tracepoints each (0-31, 32-63, 64-95, and 96-127) that you can turn on or off with a click. (The Tracks cdev contains buttons for levels 1 through 4, which correspond to these four groups.) Most new users start out tracing all information. But as more and more Tracks output is added, information overload can be a problem, and it's great to be able to limit Tracks information easily.

PartCodes are used to identify consecutive Tracks calls that have the same diagID. PartCodes should start at 0 and increment by 1 for each additional Tracks call with the same diagID. For example, say you wanted to dump the contents of all three parameters you receive on entry to a function. You'd probably want all these to have the same diagID. The first Tracks call should have a partCode of 0, the next call a partCode of 1, and so on. The partCode makes it more evident if some Tracks information is lost. Data can be lost if the circular buffer fills before writing to a file, and data can be locked out if Tracks is already in use.

THE TRACKS MACROS

To log data from your driver, you call one of five simple macros from your driver code. Each macro logs a different kind of information. All the calls must have access to your driver's global storage and follow the numbering conventions just described for the diagID and partCode.

SOME HELP WITH TERMS

breakpoint A tracepoint that enters your debugger.

diagnostic ID (diagID) A number between 0 and 127 that represents one bit in a 128-bit flag. In Tracks, a diagID is assigned to a tracepoint. The flag determines whether trace data will be logged or not. A diagID can represent a single Tracks call or a grouping of calls.

DumpTracks An MPW tool that lets you see Tracks output.

partCode A number used to identify consecutive Tracks calls that have the same diagID.

tracepoint An invocation of a macro in your driver code that sends out information to the Tracks driver.

Tracks A programming utility—containing the Tracks cdev, INIT, and driver—used to debug drivers in development.

```
T_STACK(diagID);
```

T_STACK, one of the most useful calls, records the current function and who called it. If the driver is written in C++, a special unmangler automatically prints out the arguments that were passed to the function. If called from every major routine, T_STACK will leave the proverbial trail of bread crumbs. T_STACK's partCode is always 0.

```
T_DATA(diagID, partCode, &dataBlock, sizeof(dataBlock));
```

T_DATA is used to dump a block of memory, formatted in hexadecimal and ASCII.

```
T_TYPE(diagID, partCode, recordPtr, sizeof(Record), "\pRecord");
```

T_TYPE records a data structure. The address, size, and a Pascal string with the name of the structure must be passed to the macro. The format of the data structure must be defined in an 'mxwt' resource, stored in your driver or in your MacsBug Debugger Prefs file. If the resource to define the structure isn't found, the data will be treated as a T_DATA call. Since the templates are used only to format data, you don't need to use MacsBug.

```
T_PSTR(diagID, partCode, "\pA string you'd like to see");
```

T_PSTR simply records a Pascal string.

```
T_PSTRLONG(diagID, partCode, "\ptheLong = ", theLong);
```

T_PSTRLONG records a Pascal string and a long. Usually the string is used to tell you what follows. Feel free to cast whatever you can get away with to the long.

For more information on the 'mxwt' templates, see the MacsBug documentation.•

A LOOK AT THE TRACKS CODE

This section is for folks who are really wide awake and ready for the gritty details. (If you're not one of those folks, you may want to jump ahead to the "Installing Tracks" section.) The Tracks file contains a cdev, an INIT, and the Tracks driver. The Tracks driver has three key responsibilities: maintaining the cdev, sending messages to the target driver, and accepting data from the target driver via the trace procedure (TraceProc). Figure 1 shows the flow of data between Tracks and the target driver.

MAINTAINING THE CDEV

The Tracks driver's first responsibility includes sending status information to the cdev and responding to cdev commands like "clear buffer" and "write file." Because the cdev displays the status of fields that can change at any time, the cdev monitors the driver and updates fields as they change.

The Tracks driver doesn't always need periodic (accRun) messages. When the driver gets a message to turn its periodic write-to-file flag on or off, the driver sets or clears its dNeedTime bit in the dCtlFlags. (Recall that BitClr, BitSet, and BitTst test bits starting at the high-order bit.)

```
BitClr(&dCtl->dCtlFlags, 2L);   /* Clear bit 5 = dNeedTime bit. */
BitSet(&dCtl->dCtlFlags, 2L);   /* Set bit 5 = dNeedTime bit. */
```

SENDING MESSAGES TO THE TARGET DRIVER

The Tracks driver can send one of two messages to the target driver: "enable tracing" or "disable tracing." The enable message passes the target driver a function pointer that points to an address within the Tracks driver code as well as a pointer that points to the Tracks driver's own globals. The target driver needs to save both of these because they're needed by the Tracks macros. The macros use the function pointer to call the Tracks driver directly, passing it the globals pointer along with tracing data.

When the target driver gets the disable message, the saved function pointer is set to nil. (For the code to handle enable and disable messages, see the "Installing Tracks" section.) The Tracks macros in the target driver check to see if the function pointer is nil, and if it isn't, the target driver calls the function pointer within Tracks with arguments that correspond to the particular Tracks function.

The macro that checks and invokes a non-nil function pointer is defined in the following code. The macros used in the target driver's code reference this macro. Notice that for a Tracks call to compile, it needs to access your globals by the same name, in this case by the name **globals**. Macros are used so that they can easily be compiled out of the final product.

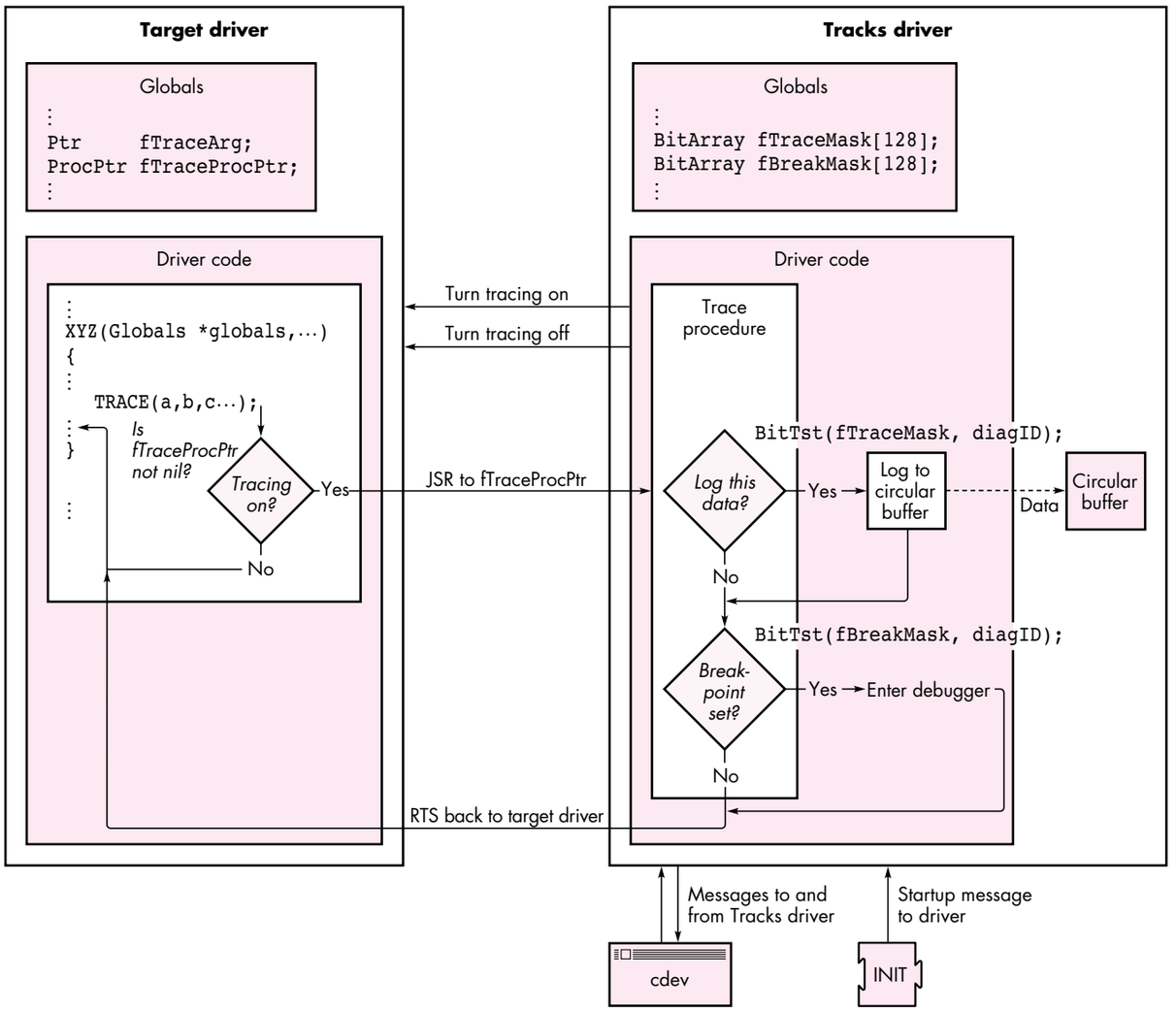


Figure 1
How Tracks Interacts With the Target Driver

```
#define TRACE(diagID, partCode, formatID, data1, data2, data3) \
{ register ProcPtr func; \
func = globals->fTraceProcPtr; \
if ( func != nil ) \
(*((pascal void (*)(long, unsigned char, unsigned char, unsigned char, \
long, long, long))func)) \
(globals->fTraceArg, diagID, partCode, formatID, \
data1, data2, data3); }
```

ACCEPTING DATA FROM THE TARGET DRIVER

The actual routine the macro executes, located in the Tracks driver, is shown below. The address of this routine was passed to the target driver in the enable message, and the first argument (long refcon) is actually the pointer to the Tracks driver's globals, which the Tracks driver expects the target driver to pass back to it each time. The macro calls right into the Tracks driver code.

```
pascal void TraceProc(long refcon, unsigned char diagID, unsigned char
   partCode, unsigned char formatID, long data1, long data2, long data3)
{
register TraceGlobals     *globals;
register Boolean          okLocked;
register Boolean          breakOnExit = false;

   globals = (TraceGlobals *)refcon;  // Set up driver globals.

   if (diagID < 128)      // Valid diagIDs range from 0 to 127.
   {
      // Check the need for a break on exit (breakpoint was set).
      breakOnExit = BitTst((Ptr)globals->fBreakMask, (long)diagID);

      // Check to see if the information passed should be logged.
      if (BitTst((Ptr)globals->fTraceMask, (long)diagID))
      {
         // The tracepoint was set--check that the buffer is ready.
         if (globals->fBufferEnabled)       // Is the buffer ready?
         {
            // Test and set "locked-out" flag.
            okLocked = UTLock(&globals->fTraceLock);

            // If trace request was locked out, set locked-out flag.
            if (okLocked)
               // Log incoming data to circular buffer.
               HandleTraceData(globals, diagID, partCode, formatID,
                  data1, data2, data3);
```

```
            else
               globals->fLockedOutFlag = true; // Locked out!
         }
      }
   }

// Handle a breakpoint, if any.
   if (breakOnExit)
   {
      // We can assume there's a debugger installed.
      if (globals->fBreakOnceThenClear)
      {
         BitClr((Ptr)globals->fBreakMask,(long)diagID);
         // Signal cdev that debug mark was turned off.
         globals->fDebugMarkUnset = true;
         DebugStr("\pTrace User Breakpoint (Once)");
      }
      else
         DebugStr("\pTrace User Breakpoint");
   }
   return;
}
```

The above routine checks to see if the diagID specified is enabled (checked in the cdev). If it is, HandleTraceData handles the data passed in the way indicated by the formatID. The formatID specifies what type of data is being passed—a Pascal string, a Pascal string and a long, a data block, a stack peek request, or a formatted type dump. Adventurous programmers *could* add their own formats (for instance, to record floating-point numbers) by modifying the HandleTraceData routine and DumpTracks and then creating a new macro. Adding a new format isn't trivial, though. Usually, it's easier to make an existing format do the job.

Since this routine can be called at interrupt time, it needs to test and set a "locked-out" flag, which it does with an assembly language routine called UTLock that uses the 68000's BSET instruction. BSET helps ensure that the routine won't get into trouble by executing more than one instance of itself. If the routine gets locked out, DumpTracks will notify you that some data was lost.

The routine also checks to see if a breakpoint has been set for that diagID. If it has, just before the routine exits, it invokes the debugger. Two kinds of breakpoints are supported—"once-only" and "immortal." The once-only kind of breakpoint flag is cleared after being tripped. Tracks breakpoints can be cleared or set only through the Tracks cdev—not from your debugger.

An interesting routine in drvr.c is StackPeek, which is called by T_STACK macros. StackPeek examines the stack frame to find what the current procedure is and who called it. StackPeek searches backward until it finds the return address of the function that called T_STACK. From there, it searches the actual code, looking for the last instruction (an RTS, a JMP(A0), or an RTD), which is followed by the name of the function. It then finds the length of the function name and repeats the process for the caller, which is just one stack frame deeper.

INSTALLING TRACKS

The following are instructions for adding Tracks capabilities to a driver written in C or C++. Currently the only way to view Tracks output is through the DumpTracks MPW tool.

1. Install the Tracks files on your hard drive. Put a copy of Tracks into your System Folder or, in System 7, into your Control Panels folder. Put DumpTracks into your MPW Tools folder and add the 'dcmd' resource in Tracks.dcmd to your Debugger Prefs. Finally, make a copy of TracksInfo.h and put it with your driver code.

2. Add two variables to your driver's global storage area. These must be accessible from every place you want to trace from. You may need to redo your globals so that these variables are always able to be accessed in the same way. The globals should look like this:

```
typedef struct
{
   your stuff
   ProcPtr   fTraceProcPtr;
   Ptr       fTraceArg;
}  Globals, *GlobalsPtr;
```

3. In your initialization routine, you'll need to set the fTraceProcPtr to nil.

```
globals->fTraceProcPtr = nil;
```

   Tracks calls attempted before this is done will result in fireworks. If your driver's global storage isn't referenced by a parameter called **globals**, you can change the word "globals" in the file TracksInfo.h to whatever the global storage is referenced by. The Tracks macros require you to be consistent in your global storage references.

4. Include TracksInfo.h in the header file where you define your global storage block.

5. Add two csCodes to your driver's code. They need to look like this:

```
case kInstallTrace:
globals->fTraceProcPtr = ((TraceDataPtr)paramPtr)->TraceProc;
globals->fTraceArg = ((TraceDataPtr)paramPtr)->TraceGlobals;
break;

case kRemoveTrace:
globals->fTraceProcPtr = nil;
globals->fTraceArg = nil;
break;
```

6. Add two resources to your driver's .r or .rsrc file: 'DrvN' and 'STR#'.

   Resource 'DrvN' ID 128 contains a Pascal string with the name of your driver (which starts with a period). The 'DrvN' resource lets the cdev know which driver to send the "turn on" and "turn off" messages to.

   Resource 'STR#' ID 777 is a string list that should contain the names of tracepoints you create. It's for the tracepoint names in the cdev and for DumpTracks output—not for use by your driver. The 'STR#' resource can be partially filled, blank, or even missing.

   When you add a Tracks call with a new diagID, you'll want to give it a name and add it to the string list. Changes show up the next time the cdev is opened or DumpTracks is used. If the 'STR#' resource is missing, the tracepoint name will show up in the cdev as a number—the diagID. DumpTracks will warn you when there's no name associated with the diagID.

   Warning: The diagIDs range from 0 to 127, and in ResEdit the string list is set up to start at 1. This means that if you add a Tracks call with a new diagID of 5, you need to change entry number 6 in the string list.

Add a few Tracks macros to your code, rebuild your driver, and you're set to start using Tracks. When you're ready to ship your driver, simply #define GOLD in TracksInfo.h and remove the two Tracks resources and any 'mxwt' resources from your driver.

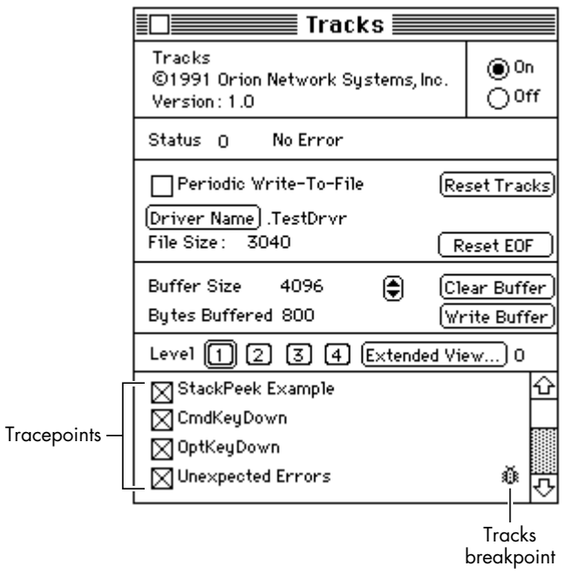


Figure 2
A Look at the Tracks cdev

USING TRACKS

Tracks can be controlled via dcmd or cdev. The dcmd lets you turn tracing on and off and write the circular buffer to disk. The Tracks cdev, shown in Figure 2, lets you control all the Tracks functions.

TRACING AND SETTING TRACEPOINTS

To begin tracing data, open the cdev, click the Driver Name button, and select a driver that has Tracks code installed at the Standard File prompt. If the target driver is set up properly, you'll see the name of the driver next to the button. To turn on tracing, click the On button and check the tracepoints you want traced. Tracepoints are represented by the list of 128 checkboxes. As soon as information is retrieved, the number in the Bytes Buffered field will change. To stop sending data to Tracks, click the Off button.

SETTING BREAKPOINTS

To set breakpoints from Tracks, either click in the Tracks cdev just to the left of the scroll bar, opposite the desired tracepoint, or Option-click a checkbox. A tiny bug will appear, indicating a breakpoint. When a breakpoint is hit, you'll need to step a few

times to return from the Tracks code to your driver. Since the default type of breakpoint is once-only, a breakpoint must be reset each time after it's encountered.

SETTING THE BUFFER

Like all good circular buffers, the Tracks buffer will hold the most current data. The default (and minimum) setting is 4K. If you want to change this size, you need to turn off Tracks and clear the contents of the buffer before clicking the arrows button. Generally, it's better to have a large buffer if you can afford it. But if you aren't logging a lot of data, and periodic write-to-file is turned on, you can have a small buffer and not lose any information.

An excellent use of the circular buffer is to catch sporadic bugs that might not occur for hours (or days). For example, set up a test to run continuously until the problem is detected. Plan to let the test run over the weekend with write-to-file turned off. When you come in on Monday, the circular buffer will have the last 4K of data—or whatever size buffer you used—leading up to and including the occurrence of the problem.

WRITING THE BUFFER

Before you can examine any Tracks data, you need to send it to a file by clicking the Write Buffer button. To clear the file, click the Reset EOF button. The data is always written to the Tracks Prefs file, which hangs out in your System Folder or, in System 7, in your Preferences folder. Use the Reset EOF button instead of throwing the Tracks Prefs file away, since settings information is also stored there.

If you check the Periodic Write-To-File box, data will be written to the disk approximately every second (60 ticks), assuming there's data to send. Be forewarned that data can come out at an alarming clip—in minutes you can create a multimegabyte file. The periodic writes-to-file occur even when the cdev is closed, until you turn it off or your hard drive becomes full.

If your driver crashes, you can write the circular buffer to disk via the Tracks dcmd. Just type "Tracks write" from your dcmd-supporting debugger.

EXAMINING TRACKS OUTPUT

Once data has been written to your Tracks Prefs file, you can examine it using DumpTracks. Figure 3 shows a sample (from the TestDrvr example) of a couple of simple Tracks calls and the type of output you'll get. Notice that each routine that wants to use a Tracks macro needs to have a pointer to the globals passed as an argument with the same name—in this case, **globals**.

The first line of a record holds a time stamp. Because other calls with the same diagID will follow immediately, it's shown only when the partCode is 0.

Output

```
++ Tracks Record 1:33:32 PM at Mon, Aug 12, 1991 ++
++ DiagID 00 ( StackPeek Example ) ++
(  0- 0) STACKPEEK
'HandleCmdKeyDown'
 called by 'DRVRENTRY'
++ Tracks Record 1:33:32 PM at Mon, Aug 12, 1991 ++
++ DiagID 01 ( HandleCmdKeyDown ) ++
(  1- 0) FORMATTED TYPE DUMP
DCtlEntry
 'dCtlDriver       ' 0xD4E8
 'dCtlFlags        ' 31840
 'dCtlQHdr         ' QHdr
 'qFlags           ' 0
 'qHead            ' 0x0
 'qTail            ' 0x0

 'dCtlPosition     ' 0
 'dCtlStorage      ' 0x4978
 'dCtlRefNum       ' -131
 'dCtlCurTicks     ' 54430
 'dCtlWindow       ' 0x0
 'dCtlDelay        ' 15
 'dCtlEMask        ' 0
 'dCtlMenu         ' 0
```

Corresponding Code

```
void HandleCmdKeyDown(Globals *globals, DCtlPtr dCtl)
{
 /* This will record who called this routine. */
 T_STACK(kStackPeek1);
 // Record a formatted data type.
 // The template is defined in MacsBug 6.2's Debugger Prefs.
 // File in resource 'mxwt'.
 T_TYPE(kCmdKeyDown, 0, dCtl, (long)sizeof(DCtlEntry),
 (long)"\pDCtlEntry");
}
```

Output

```
++ Tracks Record 1:33:34 PM at Mon, Aug 12, 1991 ++
++ DiagID 00 ( StackPeek Example ) ++
(  0- 0) STACKPEEK
'HandleOptKeyDown'
 called by 'DRVRENTRY'

++ Tracks Record 1:33:34 PM at Mon, Aug 12, 1991 ++
++ DiagID 02 ( HandleOptKeyDown ) ++
(  2- 0) PSTR + LONG
'EventCount = '  2178    0x882
(  2- 1) PSTR
"KeyMap = "
(  2- 2) DATA
                                          ---- ASCII -----
00:  0000 0000 0000 0004  0000 0000 0000 0000    ................
```

Corresponding Code

```
// This routine is called when the Option key is down.
void HandleOptKeyDown(Globals *globals)
{
 /* This will record who called this routine. */
 T_STACK(kStackPeek1);
 // Log periodic event count.
 T_PSTRLONG(kOptKeyDown, 0, "\pEventCount = ", (long)
 globals->EventCount);
 T_PSTR(kOptKeyDown, 1, "\pKeyMap = ");
 T_DATA(kOptKeyDown, 2, theKeyMap, (long)sizeof(KeyMap));
}
```

Figure 3
Comparing Tracks Output With the Calls

The second line shows the diagID and the name of the corresponding tracepoint, shown in parentheses.

The "(diagID - partCode) TYPE_OF_TRACE" line (the third line) is followed by the data for that Tracks call.

Notice that the T_TYPE call formats the contents of the driver's DCtlEntry. To be able to display formatted types, DumpTracks needs to read the 'mxwt' format from Debugger Prefs. Also notice that the DCtlEntry has a QHdr structure inside it, which was also displayed.

ON YOUR OWN

Debugging a device driver can be time consuming and difficult. Tracks provides you with a tool to help keep your drivers under control. How you set up tracing really depends on what kinds of things you'd like to monitor—error conditions, your own driver statistics, or whatever. If you suspect bugs will be, or are, a major source of headaches, you'll save time by adding lots of Tracks calls.

Take a look at the TestDrvr sample source code. Once you get Tracks going in your own code, you should find that you're debugging your drivers in a fraction of the time it used to take.

RECOMMENDED READING

- *Inside Macintosh* Volume II, Device Manager chapter (Addison-Wesley, 1985).
- *Designing Cards and Drivers for the Macintosh Family*, Second Edition (Addison-Wesley, 1990).
- "Using Object-Oriented Languages for Building Non-Applications in MPW" by Allan Foster and David Newman (MacHack '91 proceedings, available on CompuServe).
- "Writing a Device Driver in C++ (What? In C++?)" by Tim Enwall, *develop* Issue 4, October 1990. (One caveat if you want to build the sample driver: Be sure to use to MPW 3.1 and System 6.)
- *Black Holes and Warped Spacetime* by William J. Kaufmann (W. H. Freeman and Co., 1979).
- *The Pleasantries of the Incredible Mulla Nasrudin* by Idries Shah (E. P. Dutton & Co., 1968).

THANKS TO OUR TECHNICAL REVIEWERS
Neil Day, Jim Flood, Craig Hotchkiss, Gordon Sheridan•

THE VETERAN NEOPHYTE

SILICON SURPRISE

DAVE JOHNSON

Many of the things that are important, many of the phenomena that drive the world, are based on very simple rules. Huge numbers of independent entities interacting in a simple way at their local level can exhibit surprisingly complex behavior. The amazing and endlessly fascinating thing is that the end result is not at all obvious if you look only at the local rules.

Weather, for instance: get a bunch—and I mean *lots*—of gas molecules and water vapor together, and weather just happens (I've heard that really big closed buildings, like hangars and roofed stadiums, experience "weather" inside). As far as the molecules are concerned, there's no such thing as weather; they just sort of bump around and interact with their neighbors, and the result is wind, or clouds, or rain.

Another good example is evolution (one of my favorite topics): throw a bunch of replicating things into an environment with limited but necessary (for replication) resources, and evolution just happens. As far as the replicators are concerned, there's no such thing as evolution; they simply do their best to replicate, and the result is trees, or dogs, or us.

Chemistry is another example that comes to mind: throw a bunch of atoms together, and chemistry just happens. Again, as far as the atoms are concerned, there's no such thing as chemistry; they simply attract and repel each other, sticking together or flying apart, swapping electrons around, and the result is diamonds, or dynamite, or rust.

The examples go on and on, you can find them almost anywhere you care to look. Scientists call it "emergent behavior": simple, local rules, repeated ad infinitum (in time, or space, or even some other dimension), surprisingly often produce behavior that's unexpected, even unpredictable, from just the rules. One of the things I like so much about computers is that they're superlative tools for exploring emergent behavior.

There are three things in particular that make computers so good for this task: they can do arithmetic unsupervised, once they're told what to do; they can do their arithmetic inside a logical structure; and they can do it *really fast.* This combination is extremely powerful and, more important, is unique to computers. Before computers, no one ever saw good pictures of fractals—though a few mathematicians knew they were there—and the reason is simply that no one had the patience to slog through the incredibly tedious, repetitive arithmetic needed to generate pictures of them. Computers allowed mathematicians to write a recipe for the math, and then just wait a little while for the results. In this sense, computers are a kind of microscope that allows people to see certain things *for the very first time.*

Today there's a huge and burgeoning branch of research, often and aptly termed the "sciences of complexity," that has only become possible with the aid of computers. Emergent behavior is just one aspect of this larger field. The study of complexity is suggesting all kinds of brand-new approaches in long-established fields. Medicine, sociology, psychology, economics, biology, neuroscience, mathematics, physics—all have been affected. Computers have also given rise to completely new fields of inquiry: artificial intelligence, artificial life, chaos theory, neural networks, genetic algorithms, even the study of computation itself. The list of applications and repercussions seems to be endless.

DAVE JOHNSON once spent the better part of a day at the public library researching rock skipping (a.k.a. gerplunking or dapping). He found two official organizations, one annual event, and a handful of articles in various magazines. Although he sent very nice letters to the organizations asking for further information, he never heard from them. The currently recognized world record is 29 skips. Rock skipping is still poorly understood by scientists.•

It's amazing to me still, and probably always will be, that doing arithmetic inside a logical structure is a necessary *and sufficient* condition to simulate anything that can be described precisely. (Even things that can't be described precisely can be "precisely approximated"; a fact that makes engineers rejoice but mathematicians gag.) Simply doing arithmetic very fast and automatically produces a blazing, frothing torrent of diversity, a veritable fire hose of creation.

What's even more fascinating to me is that computers themselves are beginning to exhibit many of the properties that characterize complex systems, including emergence. All they do, really, is arithmetic. (Of course, if you want to get down deep, all they do is shove electrons around, but that's a little too abstract, even for me.) But look at all the things computers are used for today, and think of all the things they *could* be used for. Admittedly, this progression and diversification is driven by humans—it wouldn't happen without us—but the number and variety of computers and software that exist have arisen without a grand design, without an overall plan. It has truly begun to evolve.

Early computer programs directly reflected the computer's capabilities. Most were basically number crunchers, since at heart the computer is a number cruncher. Computers were, after all, invented to do long, time-consuming calculations quickly and automatically (it helps a lot during wartime). And that's *still* all they do, but the programs have changed dramatically.

Programmers soon began to abstract their programs away from sheer arithmetic—and thus from the machine—and began to use the arithmetic to simulate other things, both strange and ordinary. Word processing, computer graphics, spreadsheets, databases: all these arrived on the scene. There was (and still is) a wild divergence away from simply doing arithmetic. In theory, according to mathematical proofs, computers can simulate *any* logical system. There are certainly plenty of logical systems to go around, and plenty more to invent.

So the progress of computing is a kind of human-driven evolution, with human use being the "fitness function" (that is, the function that determines how well a particular entity is doing). Humans also drive the mutation and recombination, since they're the ones inventing and modifying programs. And that's where programmers come into the picture. If we're dealing with an evolutionary process, and we want it to continue as fast as possible (we do, don't we?), we should provide the things that drive evolution most strongly: diversity, large numbers, and strong selection pressure.

Selection pressure is amply provided by the marketplace; applications that aren't useful, or are too expensive or buggy, die quick ignominious deaths. The large numbers that we need are already there, and getting larger. We can help increase them by moving away from the current tendency toward huge, multipurpose, feature-crammed applications and trying to get closer to the concept of independent, single-purpose tools. (Besides, small programs are easier to develop, easier to support, and easier for people to learn.)

This "granulation" also helps increase diversity, in that it breaks up the different functions of an application into independent entities, with "lives" of their own. But even more effective at increasing diversity is thinking of new things. Only by trying new stuff, by constantly exploring the landscape of possibilities, by endlessly diversifying, do we make progress. Today's applications are only the tiniest subset of what's possible.

Admittedly, there are very real practical limits: computers are only so fast (so far); developers need to make a living, so their programs have to sell (excepting, of course, those of you lucky enough to work in research and academia: you can't use this excuse); and, probably most important, programming computers well turns out to be *really hard!* But none of these limits are insurmountable. Computers are getting faster at an incredible rate, new markets are opening up as the number and diversity of computer users increase, and

programming is getting easier. (Obviously the joy of programming has very little to do with the mechanics of communicating with the machine: just look at all the assembly hackers and UNIX folks in the world. Come to think of it, maybe a lot of the fun is figuring out how to say what you want with a painfully limited vocabulary.)

A characteristic trait of complex systems is their sensitive dependence on initial conditions. Ask any meteorologist. A tiny whisper of change can cascade into a complete transformation of the system. The evolution of computing is careening along at a very high speed, with a lot of inertia, and in a lot of directions; but a gentle shove in just the right place might profoundly affect the outcome. Where's the right place to push? If I knew, I wouldn't be working for a living. But if we all just start pushing everywhere we can think of, as often as we can, then we're helping computing reach its next incarnation, whatever *that* may be. I can't wait to find out.

RECOMMENDED READING

- *Artificial Life,* edited by Christopher G. Langton (Addison-Wesley, 1989).
- *Chaos* by James Gleick (Penguin Books, 1987).
- *Great Mambo Chicken and the Trans-Human Condition* by Ed Regis (Addison-Wesley, 1990).
- *The Tenth Good Thing About Barney* by Judith Viorst (Atheneum, 1971).

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 *If I send Apple events to myself and specify kAEQueueReply to AESend, the event doesn't get put in the queue as I requested. It shows up immediately in the reply parameter. According to Inside Macintosh, if I specify kAEQueueReply I should treat the reply event as undefined. Please help; my application falls apart because it never receives the event it's supposed to. If this is a bug, will the behavior be changed in the future?*

A This isn't a bug; it's an undocumented "feature" of the Apple Event Manager (AEM). If you send an Apple event to yourself, the AEM *directly dispatches* to the handler you've installed for that event. So Apple events you send to yourself don't come in through WaitNextEvent. This means that if you reply to an Apple event you sent yourself, your 'ansr' handler will get called directly.

This was not an arbitrary decision, though it can have some confusing ramifications for an application. Two factors influenced the decision—the first minor, the second major:

- Speed. The AEM has all the handlers for your application in a hashed table, and can dispatch *very* quickly to them, so for performance reasons direct dispatching was implemented.
- Event priorities and sequencing. Apple events have a lower priority than user-generated events (keystrokes, clicks); they come in right before update events. This created a potentially serious problem for applications that sent Apple events to themselves.

If all Apple events came through the event loop, you could easily create the following scenario:

1. The user selects a menu item, the application sends an Apple event to itself in response, and this Apple event requires a reply or will cause other Apple events to be sent.
2. The user clicks the mouse in an application window.

The mouse click has a higher priority than the reply or any Apple events that are sent in response to the first Apple event, and gets posted ahead of the Apple event in the event queue. This means that the mouse click happens and conceivably changes the current context of the application (perhaps switching windows, for example); then when the Apple events sent by the menu item handler are processed through the queue, the application state is *not* the same as it was when the menu selection was made, and the menu selection may be totally inappropriate for the current configuration.

So, to prevent a loss of sequencing, the AEM directly dispatches. Any non-Apple events that happen while you're sending and processing an Apple event

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, Mark Baumwell, Mike Bell, Jim "Im" Beninghaus, Rich Collyer, Neil Day, Tim Dierks, Godfrey DiGiorgi, Steve Falkenburg, C. K. Haun, Dave Hersey, Dennis Hescox, Dave Johnson, Rich Kubota, Edgar Lee, Jim Luther, Joseph Maurer, Kevin Mellander, Jim Mensch, Bill Mitchell, Guillermo Ortiz, Greg Robbins, Gordon Sheridan, Bryan "Stearno" Stearns, Forrest Tanaka, Vincent Tapia, John Wang, and Scott "Zz" Zimmerman for the material in this Q & A column.•

to yourself will be queued and won't interrupt the Apple event process you've initiated. What this means in the case you're describing is that queued replies don't happen when you're sending to yourself. The AEM will directly dispatch to your 'ansr' handler, bypassing WaitNextEvent processing of Apple events to prevent any other events from breaking the chain of Apple events you may be processing. This isn't a major problem, but it's something you need to be aware of if you're expecting some other events to be processed before you get a reply or other Apple event.

Q *Under System 7, but not System 6, HiliteMode doesn't work when the foreground and background colors are similar. Is this a bug?*

A Yes, it's a bug. The problem you encounter exists whenever the background and foreground color map to the same color table index. If the foreground color is the same as the background color, highlighting is ignored. Therefore, you should always make sure the foreground and background colors are different when using HiliteMode.

Q *When my hardware device sends data to a Macintosh at 57.6 kilobaud, characters can get lost if LocalTalk is on and a file server is mounted. Does Apple know about the problem? Is there a solution?*

A The coherency of a standard serial connection is not guaranteed at any baud rate. Guaranteed delivery requires more complicated protocols like those employed by AppleTalk or by file transfer protocols like Kermit. At high baud rates, particularly over 19.2 kilobaud, a Macintosh serial connection may not be reliable depending on a number of factors.

Every single character that's sent or received by the Macintosh serial port requires an interrupt to alert the processor that a character is available at the receiver or that the transmitter is ready to send a character. A nasty drawback of interrupts is that they can be masked—equal or higher priority interrupts may be in service or other pieces of code running on the machine can disable interrupts altogether. If the period of time interrupts are not serviced becomes too great, the 3-byte receive buffer in the serial chip overflows with incoming data and characters are lost. This is known as a hardware overrun.

AppleTalk works a little differently. When a packet comes across the network, an interrupt alerts the processor to that event. AppleTalk code then disables interrupts and polls in the entire packet without multiple successive interrupts. This is necessary because of the high speed at which data is received: 230.4 kilobaud. The overhead of processing an interrupt for each byte of data would be prohibitive.

A “worst case” situation is that a 603-byte packet (the largest possible AppleTalk packet) sent to the LocalTalk port takes approximately 26 milliseconds to receive. Compare this with asynchronous serial data transmitted at 9600 baud, which would take only 1 millisecond to receive. A very realistic situation arises in which 26 bits could be lost during a concurrent AppleTalk/Serial Device transmission. This possibility is increased when there are routers on the network, which send out large RTMPs (Routing Table Maintenance Packets).

During the time that interrupts are disabled by AppleTalk, AppleTalk code attempts to poll regular serial data as well (from the modem port only) to prevent the hardware overrun problem. Unfortunately, at very high serial baud rates there may not be enough time to do both; characters may be lost anyway.

This same loss of characters may also occur in conjunction with the Sony driver. If a disk is inserted that requires special attention (like formatting), the odds of losing characters is increased. The Sony driver has built-in checks like the AppleTalk driver (although not as frequent). So you can still lose data at high transfer rates if a disk is inserted during the transfer.

The character dropout also occurs on systems that aren’t currently running AppleTalk but are receiving serial transmissions from high-speed devices (57.6 kilobaud) and are performing CPU tasks that require a high amount of memory access (such as a CPU like the Macintosh IIsi or IIci running on-board video in 8-bit mode, or a CPU using virtual memory).

This is simply a performance problem, asking the machine to process more data over two separate ports than the processor speed and hardware architecture can allow. Any number of factors affect serial performance at high baud rates. Turning off AppleTalk helps. Turning on 32-bit addressing helps because it reduces interrupt handler overhead. Turning on VM hurts performance for the opposite reason. Running on a faster machine helps. Running on a Macintosh IIfx or Macintosh Quadra 900 with a serial I/O Processor (IOP) helps a lot because the IOP handles the serial port regardless of whether the main processor is busy doing something else.

There is, however, a way to ensure reception of high-speed serial data without character loss—through the use of a Serial NB card or a third-party NuBus™ card with a dedicated processor necessary to provide higher speeds. These cards in essence operate as dedicated port-handling circuitry. They’re able to perform necessary buffering while the processor is servicing other interrupts. This is the reason that network cards such as EtherTalk and TokenTalk can accomplish transactions without data loss; they do at least some of their own buffering until being serviced to avoid interference with other operations such as receiving serial data.

Alternatively, you may want to develop a custom card yourself that exactly fits the needs of your product. In this case you should look into the Macintosh Coprocessor Platform (MCP) and Apple Real-Time Operating System Environment (A/ROSE) as a possible basis of this line of development. Development packages for both these products—the Macintosh Coprocessor Platform Developer's Kit (#M0793LL/B) and the A/ROSE Software Kit (#M0794LL/B)—are available from APDA.

Q *How can I make FSSpec file information comply with what was an SFReply information block? Is there a way to convert FSSpec information—as passed, for example, via an Open Documents Apple event—to a vRefNum as understood by an SFReply record? We want to keep our tried-and-true non-System 7 file management logic and convert from FSSpec to SFReply-type format.*

A Not wanting to make a good bit of file system code obsolete is understandable; however, while it's unlikely that Apple will dispense with support for old SFGetFile or SFPutFile functions in the near future, the use of SFReply-style data structures in internal calls has no development future.

The vRefNum field of the SFReply record was originally (in *Inside Macintosh* Volume II days) a volume reference number; later, with the creation of HFS in 1986, it became a working directory reference number for purposes of backward compatibility. In HFS, a file or directory entity on a volume is specified with a volume reference number, a directory ID, and a name. An FSSpec contains this latter information.

Converting from FSSpec to SFReply requires that your application manage the manipulation of working directory entities, which has disadvantages from the point of view of the system and compatibility. There are several difficulties with working directory references:

- There's a system-wide limit on their number.
- If you have a working directory reference to which no file buffers are open and some other application closes that working directory without your knowledge of it, your internally stored reference number is invalid and you have no way of knowing about it.
- The documentation about where, when, and how to close a working directory is somewhat ambiguous.
- An FSSpec can refer to either a file or a directory while an SFReply can refer only to a file.

Developer Technical Support urges you to take the time to remove dependencies on SFReply data structures as soon as is feasible.

Q *Can I add a media to a QuickTime movie that is not video or audio? If so, is there anything special I need to do to add text notes that can potentially accompany each frame in my "movie," which can follow the video frames if a user edits the movie in any way?*

A QuickTime version 1.0 allows for only video and sound media. There's no way to install your own type, even in a case so obvious as the one you mention. Adding other media types is a high QuickTime priority and is likely to make it in a future release, but currently there's no mechanism to do it.

Q *Inside Macintosh Volume V (page 445) states that SGetCString automatically allocates memory for holding the requested string. Does this mean that repeated calls to SGetCString will eventually exhaust memory and, if so, what's the correct way to release the memory?*

A SGetCString automatically allocates memory (via NewPtr) for holding the requested string. It copies the string into that pointer and returns the pointer in spResult. You need to call DisposPtr on spResult to deallocate the pointer.

Q *Installing a VBL task doesn't seem to work on Macintosh systems with on-board video. I get the slot number for a video card by extracting the second nibble of the base address of the screen's pixMap; however, on-board video computers report that the video card is in slot $B but the video card doesn't seem to exist in any slot. Is there a better way to determine a video card slot number? Is there any way to attach a VBL task to on-board video? How can I determine whether the main monitor is using on-board video?*

A You should be getting the slot number for the video from the AuxDCE entry, not from the base address of the pixMap (*Inside Macintosh* Volume V, page 424). You'll need to get the gdRefNum for the desired monitor from the device list (*Inside Macintosh* Volume V, Chapter 5). For example, to install a VBL task for the main screen, do something like this:

```
GDevHand := GetMainDevice;
IF GDevHand = NIL THEN HandleError
ELSE
  BEGIN
     mainGDRefNum := GDevHand^^.gdRefNum;
     DCEHand := AuxDCEHandle(GetDctlEntry(mainGDRefNum));
     IF DCEHand = NIL THEN HandleError
     ELSE retCode := SlotVInstall(@myVBLTask,DCEHand^^.dCtlSlot);
  END;
```

This should work regardless of the kind of video used.

Q *When using "-model far," I get the following error from the linker:*

```
### While reading file "HD:MPW:Libraries:Libraries:Runtime.o"
### Link: Error: PC-relative edit, offset out of range. (Error 48)
%__MAIN
(260) Reference to: main in file: xxx.c.o
```

What does it mean?

A In your Link command, you should put Runtime.o and the object file containing your main program close together, and as early in the list as possible. This is because %__MAIN (the part of Runtime.o that actually receives control when your application is launched) uses a PC-relative BSR instruction to transfer control to your "real" main. (OK, so it's "32-bit-almost-everything"!)

Q *Do all System 7-savvy programs need to run with background processing enabled?*

A Yes, System 7-savvy applications should have the SIZE resource's background processing bit set. (It's not documented explicitly that you need to have this bit set.)

All System 7 Apple event-aware applications need to be background-capable, since there are many instances where Apple events will come in to you while you're in the background, and there will be many times (as new applications are developed) when you will *not* come to the front to process a series of events; you'll work in the background as a client for another application.

You don't want to hog a lot of system time when you have nothing to do in the background, but with the Edition Manager you do need to be able to receive events while you're in the background. However, you can still be system-friendly when you do this. Here's one way: When you're switched into the background, set your sleep time to MAXLONGINT and make sure you have an empty mouse region. This way, you'll be getting null events *very* rarely, and you won't be taking much time away from other applications, but you can still react to events sent to you by other parts of the system. Then when you come forward, you can reset your sleep time to your normal, low, frontmost sleep.

Note that WaitNextEvent is implemented when running System 6 without MultiFinder, but there's no DA Handler ensuring that DAs receive time. In this case, large sleep values prevent DAs from receiving timely accRun calls—the Alarm Clock DA stops ticking, for example. A compromise that doesn't hog too much processing time is to use sleep values only as large as 30-60 ticks for System 6.

Q *Is there a way to test whether a particular key is down independently of the event record? My application needs to check the Option key status before entering the main event loop.*

A The call GetKeys(VAR theKeys:KeyMap) returns a keyMap of the current state of all the keys on the keyboard. The call is documented in *Inside Macintosh* Volume I on page 259. The Option key will appear as the 58th bit (counting from 0) in the map. In MacsBug you can see this with a DM KeyMap, which returns the following:

```
0000 0000 0000 0004 0000 0000 0000 0000
```

It's important to understand that the keyMap is an array of packed bits. You need to test whether the Option key bit is 1 or 0. The key code 58 = $3A is the 58th bit of the keyMap. This number can be determined from the keyboard figure on page 251 of *Inside Macintosh* Volume I and pages 191-192 of Volume V. (If in counting the above bits you get 61 instead of 58, remember that the bits within each byte are counted right to left.)

With the above information you should be able to determine the status of any key on the keyboard within your program without waiting for an event. GetKeys, however, should be called only for special situations. Normal keyboard processing should be done through events; otherwise, your application risks incompatibilities with nonstandard input devices.

Q *Read calls at interrupt time often result in a "hang," waiting for the parameter block to show "done." This happens if the interrupt occurred during another Read call. I've tried checking the low-memory global FSBusy, and that decreases the occurrence of this problem but does not eliminate it. When is it safe to make the Read call?*

A The problem you're experiencing is a common one known as "deadlock." The good news is that you can *always* make Read calls at interrupt time! The only requirement is that you make them *asynchronously* and provide a completion routine, rather than loop, waiting for the ioResult field to indicate the call has completed. This will require that you use the lower-level PBRead call, rather than the high-level FSRead.

The low-memory global FSBusy is *not* a reliable indicator of the state of the File Manager. The File Manager's implementation has changed over time, and new entities patch it and use the hooks it offers to do strange and wonderful things. File Sharing really turns it on its ear. The result is that when FSBusy is set, you can be sure the File Manager is busy, but when it's clear you can't be sure it's free. Therefore, it would be best if you ignore its existence.

If you need to have the Read calls execute in a particular order, you'll have to chain them through their completion routine. The basic concept is that the completion routine for the first Read request initiates the next Read request, and so on until you're done reading.

By the way, never make synchronous calls at interrupt time (and, contrary to the popular misconception, deferred tasks are still considered to be run at interrupt time) or from ioCompletion routines, which may get called at interrupt time.

Q *Is there a limit on the values set in the Name Binding Protocol (NBP) interval and count fields when used with PLookupName and PConfirmName calls? How do the interval and count work? If a device is not on the network and I send a PLookupName with interval = 20 and count = 20, will I wait 400 seconds before PLookupName returns?*

A Since the interval and count parameters for NBP calls are both 1-byte, the values used are limited to the range of 0-255 ($00-$FF). Here's what the values do:

- interval = retransmit interval in 8-tick units. This value is used to set up a VBL task. A value of 0 should not be used because that would mean the VBL task would never be executed and would be removed from the VBL queue.
- count = total number of transmit attempts. Each time the interval timer expires, this value is decremented by 1. When it reaches 0, the NBP call completes. So if a value of 0 is used, the packet will be retransmitted 255 times (or transmitted 256 times).

Three things can happen to make the LookupName, RegisterName, or ConfirmName calls complete:

- PKillNBP can be called to abort one of the calls (see *Inside Macintosh* Volume V, page 519).
- maxToGet matches are returned or the return buffer is filled. Here's how this works: Each time an NBP lookup reply (LkUp-Reply) packet is received, an attempt is made to add all the NBP tuples found in that LkUp-Reply packet to the return buffer. If all the tuples cannot be added to the buffer because there isn't enough room, the call completes with as many tuples as could fit and the numGotten field will contain the number of matches in the buffer. If all the tuples from the LkUp-Reply packet are added to the buffer, numGotten (the number of matches in the buffer) is compared to the value passed in the maxToGet field. If numGotten is greater than or equal to maxToGet, the call completes and the numGotten

field will contain the number of matches in the buffer. Since the buffer can fill before maxToGet matches are received and since LkUp-Reply packets can return multiple tuples, you may get more or fewer matches than you asked for with maxToGet.

- The count is decremented to 0. You can use this equation to determine how long the call would take to complete this way:

```
IF count = 0 THEN count := 256;
TimeToCompleteInTicks := count * interval * 8;
```

The RegisterName and ConfirmName calls always complete after they receive the first LkUp-Reply packet to their request, so you could look at them as always having a maxToGet of 1 (maxToGet is *not* one of the parameters for those two calls).

Q *Why does System 7 get rid of my 'vers' resources in my desk accessories when dropped into the System Folder icon?*

A The System 7 Finder takes over some of the job of the old Font/DA Mover program. The Font/DA Mover took great pains to keep all resources owned by the DA together with the driver resource. This is mentioned in Macintosh Technical Note #6, "Shortcut for Owned Resources," as well as in *Inside Macintosh* Volume I, page 109.

Under System 7, the 'vers' resource used for Finder information is not owned by the desk accessory, so when the Finder copies the desk accessory it skips the resources extraneous to the DA—including the 'vers' resource.

The simplest thing to do is to provide a System 7-ready version of your DA (see *Inside Macintosh* Volume VI, page 9-32). The Finder will not strip off resources if the file type is already 'dfil' when the DA is dragged to the System Folder. You can also provide a version of the DA in a suitcase for System 6, or you can instruct System 6 users to hold down the Option key while clicking Open in Font/DA Mover if they want to install your DA.

Q *Are CRMSerialRecord's inputDriverName and outputDriverName fields of type StringHandle or are they Pascal string pointers?*

A As shown on page 183 of *Inside the Macintosh Communications Toolbox*, the CRMSerialRecord data structure contains two fields, inputDriverName and outputDriverName, declared as type StringHandle. These two fields are erroneously described as "pointer to Pascal-style string" in the documentation further down the page. The correct declaration is type StringHandle.

Q *How many ways are there to assemble an industry-standard Mr. Potato Head, part number 2250?*

A With the standard Mr. Potato Head accessory set, there are 1,139,391,522 different ways to assemble Mr. Potato Head. Some of our favorite configurations are as follows:

- The demon cyclops Potato Head: assemble as normal, but turn the eyes sideways and tape sharp objects to his hands.
- The Australian Potato Head: assemble upside down (in Australia, this is known as the U.S.A. Potato Head).
- The "Gaping Wonder" Potato Head: assemble backwards, using the part storage compartment as a mouth.

Q *What is csCode 100? A control call with csCode 100 seems to come into my block device driver whenever the Macintosh system fails to recognize the file system on my media. Should my driver specify valid non-HFS formats such as ISO 9660?*

A The csCode 100 you're seeing is a ReadTOC command bound for what the operating system thinks is a CD-ROM drive. What's happening here is that after deciding your device doesn't contain an HFS partition, Foreign File Access (FFA) is attempting to identify the file system residing on that device; looking at the Table of Contents on a CD is one of the ways that it attempts to go about that. This csCode is documented in the *AppleCD SC Developer's Guide*, which is available through APDA (#A7G0023/A).

If you remove the Audio CD File System Translator (FST) from your System Folder, you'll find that the csCode no longer gets issued. The reason is that the ReadTOC call returns information about audio CDs. Your driver doesn't even have to support this call; if FFA sees a paramErr (which you should return upon seeing an unrecognized csCode) it knows it can't be looking at an audio CD.

You needn't worry about adding any support to your driver for ISO 9660 or High Sierra. FFA's FSTs will just request certain blocks from your driver which contain information that identifies the disk in question as ISO or HFS.

Q *What's the story with RealFont and TrueType? I'm finding that, of the standard System 7 TrueType fonts, only Symbol and Courier get a TRUE result from RealFont for 7-point.*

A You're correct in your observation of RealFont for the 7-point size of certain TrueType fonts. The explanation is hidden in *TrueType Spec—The TrueType Font Format Specification* (APDA #M0825LL/A), page 227.

The font header table contains a lowestRecPPEM field, which indicates the "lowest recommended pixel number per em," or, in other words, the smallest readable size in pixels. As it turns out, the Font Manager in its wisdom uses this information for the value it returns from RealFont. Note that for higher-resolution devices, a point size of 7 does *not* correspond to 7 pixels; but since the unit "point" is 1/72 inch and the screen resolution is (approximately) 72 dpi, the result corresponds to reality in this case.

The value for lowestRecPPEM can be arbitrarily set by the font designer. We all know that small point sizes on low-resolution devices never look great, and even less so for outline fonts. Courier and Symbol have lowestRecPPEM = 6, while the other outline fonts in the system have lowestRecPPEM = 9. This doesn't mean that Courier and Symbol (TrueType) in 7-point size look better than Times® or Helvetica under the same conditions. It means the font designer had higher standards (or was in a different mood) when choosing lowestRecPPEM = 9.

Q *Is trackbox in the MPW C library broken? It always returns 0 (false).*

A Yes, the glue for the MPW C library trackbox is broken. In fact, the glue for many of the lowercase Toolbox calls is broken. Fixing lowercase glue routines is a never-ending challenge. This probably won't be fixed; instead, expect to see all lowercase glue routines removed from future versions of MPW. What does this mean to you? Use only the proper mixed-case interfaces (the ones spelled just like in *Inside Macintosh*) at all times. This also will serve to make your code smaller and faster, since the mixed-case interfaces make direct Toolbox calls instead of calls to glue routines in many cases.

Incidentally, in case you're wondering, the C library trackbox doesn't work because the glue clears a long for the result instead of a word and pulls a long off the stack, so the result is in the wrong byte of the register on return.

Q *After an application in the System 7 Application or Apple menu is selected, sometimes control doesn't switch from our application to the selected application. What could be wrong and how can I zero in on the problem?*

A The symptoms you've described—the process menu not switching layers—is exactly what happens if you have an activate or update event pending that you're not acting on. Here are ways that pending activate or update events can be handled incorrectly:

- You're not calling BeginUpdate and EndUpdate (the most likely problem).
- Your application isn't asking for all events.

- Your application has dueling event loops.
- The word in your code that contains the everyEvent constant is trashed (unlikely).

Q *I have a problem with Balloon Help for modeless dialogs. The balloons don't show up most of the time unless the user clicks the mouse over the dialog item of interest.*

A The problem you're having is that you're not calling isDialogEvent with null events. The most likely cause for this is that you're not getting null events, or you have your sleep time way too high in WaitNextEvent. If you're not getting null events, it's probably due to unserviced update events in the event queue. If you can't ensure that you'll get null events back from WaitNextEvent, you must fudge them instead. This way the TextEdit insertion point will blink properly as well.

Q *What's the correct method for a custom MDEF to dim menu items when running under System 7? What's the preferred method for determining whether to draw in a gray color or paint the items with a gray pattern?*

A The proper method for dimming text in menu items is to use the grayishTextOr transfer mode when drawing text. This is documented on page 17-17 of *Inside Macintosh* Volume VI and is what Apple uses. This mode takes into account both color and black-and-white screens. A simple method for dimming nontext items is to set the OpColor to gray and then draw the nontext item in Blend mode.

Q *Under System 7 my filter procedure for displaying invisible data files no longer works. How can I use Standard File to display the names of invisible files of a specific type under System 7?*

A System 7 can show invisible files in the standard SFGetFile dialog box; however, not all System 6 Standard File package calls are handled the same in System 7.

When using invisible files under System 7, you should perform type filtering within a filter proc and not with the typeList field of the SFGetFile call. System 7 no longer allows a typeList for detecting invisible files. The actual check for invisible files of a particular type or types should be done within the file filter proc.

The SFGetFile call below displays only folders and invisible 'TEXT' files in the standard SFGetFile dialog box. With the numTypes parameter set to -1, all types of files will be passed to the filter proc.

```
SFGetFile( where, "", myFilterProc, -1, typeList, nil, &reply );
```

In this example, the filter proc's return value depends on the file's type and Finder flags.

```
pascal Boolean myFilterProc( fp )
FileParam *fp;
{
  if ((fp->ioFlFndrInfo.fdFlags & fInvisible) &&
        (fp->ioFlFndrInfo.fdType == 'TEXT'))
     return FALSE;
  else
     return TRUE;
}
```

Q *The System 7 Help, Keyboard, and Application menus at the right of the Macintosh menu bar don't have text titles that can be included in a command string in the way that "File" or "Edit" can be. Do these menus always have the same menu ID?*

A The menu ID numbers of the Help, Keyboard, and Application menus are always the same, so the menus can always be identified by their IDs:

```
kHMHelpMenuID = -16490;
kPMKeyBdMenuID = -16491;
kPMProcessMenuID = -16489;
```

Q *Do String2Date and Date2Secs treat all dates with the year 04 to 10 as 2004 to 2010 instead of 1904 to 1910?*

A Yes, the Script Manager treats two-digit years less than or equal to 10 as 20xx dates if the current year is between 1990 and 1999, inclusive. Basically, it just assumes that you're talking about 1-20 years in the future, rather than 80-100 years in the past. The same is true of two-digit 9x dates, when the current year is less than or equal to xx10. Thus, in 2003, the date returned when 3/7/94 is converted will be 1994, not 2094. This is all documented in *Macintosh Worldwide Development: Guide to System Software*, available from APDA (#M7047/A).

Q *Is there a universally recognized wildcard character for the Macintosh, like the "*" in the MS-DOS world? Furthermore, for Boolean logic, should my application accept Pascal syntax (such as .NOT., .AND., .OR.), C syntax (such as !, &&, ||), or still another convention? My users aren't programming gurus.*

A First, see if there's a friendlier way to implement the wildcard's function. Take a look at System 7 Finder's Find command, for example. If you find wildcard use is necessary, "*" is common, though for file searching any character other than ":" can be used in an HFS filename.

As for Boolean operators, nonprogrammers prefer a syntax that matches English as closely as possible, so AND, OR, and NOT are better than their C counterparts. However, user testing indicates that the most intuitive, user-friendly way to put Boolean search criteria on a command line is to bring up a dialog with pop-up menus used to form an English sentence describing the search (like System 7's Find). If you can make something like this work for your application, your nontechnical users will love you.

Q *When I pass FALSE in the cUpdates parameter to SetPalette, I still get update events to that window when I modify its palette. What's going on?*

A SetPalette's cUpdates parameter controls whether color-table changes cause that window to get update events only if that window is *not* the frontmost window. If that window is the frontmost window, any changes to its palette cause it to get an update event regardless of what the cUpdates parameter is. When you call SetEntryColor and then ActivatePalette for your frontmost window, the window gets an update event because it's the frontmost window even though you passed FALSE in the cUpdates parameter. Another important point is that windows that don't have palettes always get update events when another window's palette is activated.

Fortunately, system software version 6.0.2 introduced the NSetPalette routine, which is documented in Macintosh Technical Note #211, "Palette Manager Changes in System 6.0.2," and on page 20-20 in the Palette Manager chapter of *Inside Macintosh* Volume VI. This variation of SetPalette gives you the following options in controlling whether your window gets an update event:

- If you pass pmAllUpdates in the nCUpdates parameter, your window gets an update event when either it or another window's palette is activated.
- If you pass pmFgUpdates, your window gets an update event when a palette is activated only if it's the frontmost window (in effect, it gets an update event only if its own palette is activated).
- If you pass pmBkUpdates, your window gets an update event when a palette is activated only if it's not the frontmost window (in effect, it gets an update event only if another window's palette is activated).
- If you pass pmNoUpdates, your window never gets an update event from any window's palette being activated, including that window itself.

Q *I want to determine whether a disk is locked before trying to mount the volume. When I examine bit 15 of ioVAtrb using PBGetVInfo, as suggested on page 104 of Inside Macintosh Volume II, bit 15 is clear for a locked volume such as a CD-ROM, but bit 7 is set. Why is this happening?*

A The reason for your observed discrepancy is that bit 15 is set for a software lock and bit 7 is set for a hardware lock. In the case of the CD-ROM there's no software lock but only a hardware lock, so bit 7 is set and bit 15 is clear. Volumes II and IV of *Inside Macintosh* both say that "only bit 15 can be changed" and should be set if the volume is locked. The fact that you can set it with PBSetVol means that it's a software lock. What the documentation fails to mention is that using PBGetVInfo you can also check bit 7 of ioVAtrb to see if there's a hardware lock. The recommended procedure is to first check the hardware lock (bit 7 of ioVAtrb) and then check the software lock (bit 15 of ioVAtrb).

Q *How do we create a "fixed fractional width" font using ResEdit 2.1? I tried setting FontInfo fields such as ascent and widMax with fractional numbers, but ResEdit refused all noninteger numbers.*

A ResEdit doesn't know how to take fixed-point numbers as input, so it assumes the number you enter for a value like widMax is the integer representation of the fixed-point number. In other words, ResEdit displays a 16-bit fixed-point number as an integer with 256 times the value of the fixed-point number actually used by the Font Manager.

To enter the numbers with ResEdit, you'll need to do the conversion yourself. Take an 8.8 fixed-point number and multiply by 256 to get the integer to enter, or take a 4.12 fixed-point number and multiply by 4096 to get the integer.

This rigamarole is necessary because ResEdit wasn't designed to build fonts from scratch. You may find that third-party tools specifically designed for this task are easier for you. The time you save in building your width table may be worth the cost of the program.

Q *In my Installer script, how can I include the current volume name in a reportVolError alert, as many of the installation scripts from Apple do?*

A The volume name can be included by inserting "^0" as part of the Pascal string passed to the reportVolError error-reporting clause.

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.•

APPLE II Q & A

Q *What's the best way to do text-only printing to a character device through GS/OS?*

A If you want to print text to a GS/OS character device, here's how you do it:

1. Look at all the on-line devices to find which are character devices. You can tell when a device is a character device by examining bit 7 of the "characteristics" word returned by DInfo—the bit is set for block devices and clear for character devices. You don't have to hard-code a list of character device IDs.
2. Present the user with a list of device names followed by their location and generic type in parentheses. You can adjust the names of device types to reflect your use. For example, ".RPM (AppleTalk printer, port 1)," ".DEV2 (serial modem, port 2)," or ".DEV3 (generic character device, slot 5)." You can use bit 3 of the slot returned by DInfo to know whether to return "slot" or "port," too.
3. When the user selects a printing device, call Open on the device name (such as .RPM for an AppleTalk printer). Use Write to write the information, just as you would to a file; then call Close (you can call Flush if you like, but it shouldn't be necessary). That's all you need to do.

Remember not to embed ImageWriter or any other printer-specific codes in the output stream. With the exception of choosing a device and creating a file, this same code could be used to print to *any* text printer or to another device such as a disk or modem. You might give users a "text printing preferences" dialog where they can enter some codes if they like, and you might have built-in sets for ImageWriters and other common printers, but don't make it too complicated. These instructions are very generic and will work well on any setup, not just an ImageWriter II connected through the serial port.

Q *On a ROM 03 Apple IIGS, I save and restore the mouse mode by getting it from ReadMouse, setting the mode to what I need, and restoring the value (with SetMouse) when done. This sometimes kills the mouse—I don't get any mouse movement until my program quits. Help! By the way, this isn't a problem on ROM 01 machines.*

A The ROM 03 mouse firmware doesn't behave as documented in two respects. First, it sometimes returns an illegal mouse mode value from ReadMouse. Specifically, there's garbage in the high nibble of the mouse mode byte. Second, SetMouse returns an error and takes no action when passed an invalid mode, even though the *Apple IIGS Toolbox Reference* says it returns no errors. When you try to restore the invalid mode, nothing happens and the mouse stays in whatever mode you had it in—if it's not what the system needs, the mouse will appear "hung" until someone sets the mode to what the system does need.

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 Matt Deatherage, Jim Luther, Dave Lyons, Jim Mensch, and Tim Swihart for the material in this Q & A column.•

You can work around this problem by masking off the high nibble of the mode result from ReadMouse (AND #$000F in 65816 assembly) before passing it to SetMouse. This problem is fixed in Apple IIGS system software version 6.0—ReadMouse always returns a valid mode under 6.0.

Q *When I call SFMultiGet2, I randomly get error $1705 (bad pathname descriptor in the reply record) even though that error doesn't mean anything for that call. Any ideas?*

A All versions of SFMultiGet2 before Apple IIGS System 6.0 incorrectly look at two of the words in the reply record (offsets $0008 and $000E) to make sure they don't contain the value $0002. That value would be illegal in those positions—in any of the *other* new Standard File calls. SFMultiGet2 doesn't use the same reply record, but pre-6.0 versions of Standard File accidentally check those fields anyway. Make sure the values in bytes that are past the beginning of the 6-byte reply record are not $0002. This is fixed in System 6.0.

Q *Although my application fully supports GS/OS, the Finder uses slashes instead of colons in the pathnames in message #1. This means my application can't open any files that have slashes in the filename.*

A The Finder and its message-passing conventions were originally released before GS/OS was written. Applications depend on the slash (/) as the separator character to be able to parse these pathnames (for example, to find the filename to use in a document window title). If the separator character were to change, many older applications would break.

Finder version 6.0 may support an additional message containing a list of fully expanded GS/OS pathnames. These pathnames use colons as separators and aren't limited to 255 characters. See the Finder 6.0 documentation for details on using this message.

Q *I've written a program that hangs inside Standard File under Apple IIGS System 5.0.4, but works fine under development versions of 6.0. I'm not using any 6.0-specific features. What could be the problem?*

A Standard File before System 6.0 does not behave gracefully if called with both prefix 0 and prefix 8 empty. Try setting one of these prefixes to an existing directory and see if your problem vanishes.

Q *I can't find the ProDOS partition on Volume IX or later of the Developer CD Series discs. What's happened?*

A Apple II information was duplicated on all Developer CDs from Volume III through Volume VIII because without the ProDOS partition, Apple II users couldn't see the information, and without the Apple II folder on the HFS partition, the information couldn't be shared on an AppleShare file server (or Macintosh System 7 File Sharing). Apple IIGS System 6.0 includes an HFS (Macintosh) file system translator, which means that Apple IIGS developers can access the information on the HFS partition, making the ProDOS partition unnecessary.

The Apple II information can be found on the HFS partition with the pathname Dev.CD Vol. IX:Development Platforms (Moof!):Apple II. Note that this means the Apple II folder can only be accessed from ProDOS 8 using AppleShare with long naming on—the path to the Apple II folder is not a legal ProDOS 8 pathname.

Q *Apple II Applesoft's floating-point routine results are sometimes accurate to only two places. For example, the answer returned for PRINT 55555.099-55555.09 is 9.01031494 E-03. How can we get more accurate results?*

A The accuracy loss you're experiencing with the Applesoft floating-point routines is normal. If you convert a number such as 55555.099 to a base 2 floating-point number, you won't get an exact representation using Applesoft's floating-point routines or even 96-bit precision IEEE numerics. Because 9.01031494 E-03 is 0.0090103, you can see that you have accuracy out to three and a half decimal places. The solution is to determine the accuracy that you want and massage the result to give you that accuracy. Here's a sample program that shows common Applesoft rounding techniques:

```
1 REM Round to 3 decimal places of accuracy example
10 Input a
20 Input b
30 If a-b>1000 then 100: REM no 3-digit rounding of numbers >1000
40 Print "Standard Applesoft Non-accurate result:";a-b
50 Print "Truncated result:";INT((a-b)*1000)/1000
60 Print "Rounded result:";INT(((a-b)+.0001)*1000)/1000
70 Goto 10
100 Print "Result has 3 decimal accuracy already:";a-b
110 Goto 10
```

This is the only way that you can get Applesoft to clip the numbers, apart from using a separate floating-point engine. Alternatively, you can do your own conversion from Applesoft internal numeric format to a string in assembly language and have it simulate the above operation when converting the number.

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

IT'S JUST A COMPUTER

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

KON I wrote this program that crashes with a bus error and I can't figure out what's going wrong.

BAL If it's crashing with a bus error, that's easy: just figure out where the bogus bus error address came from.

110 KON Well, that's the problem. I look around and there's not a bad address anywhere.

BAL Let me see.

105 KON OK.

```
Bus Error at 1B586
1B582  BFEXTU(A3){D6; $00}, D0
1B586  *ADDQ.W   #$4,A3
1B588  MOVE.L(A2)+,D1
```

Register A3 has $70E368, and A2 has $70DEDC.

So that's your puzzle; what do you do now?

BAL Hmmm. Suppose I trace a few times.

100 KON No problem; everything seems to work OK.

BAL What if I type "Go" and hope for the best?

95 KON You crash immediately with an address error at $1B5A4. The code is trying to do an RTS, but the stack is trashed.

KONSTANTIN OTHMER AND BRUCE LEAK

As a mere lad, Bruce pulled the programmer's key out of the stone and swore allegiance to the Lady of the Leak. Years later, while good King Bruce was doing penance for his wandering ways, young Kon of Locksley had to defend the crown against the Mongol hordes attempting to draw directly to the screen. Upon his return, King Bruce declared, "Let there be time," and there was. Today young Kon and his sovereign do battle against the forces of evil lurking in your local heap.•

BAL Well that's just a little ways down from where I was before. Anything funny happening in between?

90 KON Nope, just a loop that doesn't touch the stack or anything.

BAL Was the stack OK when I crashed the first time?

85 KON No, the top address on the stack was garbage.

BAL OK, so I run the program again and break just before this routine gets called and check the stack.

80 KON The stack is OK. Everything looks fine. You're at a JSR (A0).

BAL Is A0 OK?

75 KON Yep, it points to the code you were looking at before—at $1B582.

BAL So I step in and look at the stack.

70 KON You crash immediately with a bus error at $82.

BAL Huh? Sounds like someone jumped to $0.

65 KON A0 looked OK. It was just doing a JSR (A0).

BAL Some weird MacsBug bug?

60 KON In this case, no.

BAL All I did was step into a subroutine and I crash somewhere totally different?

55 KON Pretty cool, huh?

BAL Are the registers OK? What does the stack look like?

50 KON Garbage everywhere. The stack has all kinds of noise on it, and the registers seem pretty fragged.

BAL Hmmm. I try it again; this time I set a breakpoint a few instructions before the JSR (A0).

KON OK.

BAL Is everything OK?

45 KON The registers, stack, and code look OK.

BAL So I trace a few instructions, up to the JSR.

40 KON You crash immediately after the first trace with a bus error at $0104B0CA.

BAL Well that address is garbage. Was it in any register or on the stack before I traced?

35 KON Nope.

SCORING

100–110	Be honest.
75–95	Next time we find a bug, *we're* calling *you*.
50–70	So this has happened to you!
25–45	No doubt about it, these puzzles are tough.
0–20	Well, maybe next time . . .•

BAL What happened to the stack?

30 KON There's 56 extra bytes on it now.

BAL What? Is my machine possessed?

KON It's just a computer.

BAL Some interrupt nastiness happening?

25 KON When you crash, you're at interrupt level 1.

BAL Now we're getting somewhere. Does MacsBug enable interrupts when I trace?

20 KON Yes.

BAL So where's the level 1 interrupt vector kept?

15 KON $64.

BAL I DM it and see if it's OK.

10 KON It's $104B07C. Pretty close to where you crashed.

BAL So it sounds like someone is trashing the interrupt vectors, and all the interrupts are held pending when I'm in MacsBug. As soon as I do anything that returns control to the Macintosh, I blow up. So I step spy on $64 and see who trashes it.

0 KON It's a routine that assumed a buffer was being allocated but wasn't. So the buffer pointer was NIL, and the routine wrote all over low memory, including the exception vectors.

BAL Nasty.

KON Yeah. So how could you catch this before the vectors get trashed?

BAL Doesn't EvenBetterBusError catch writes to NIL?

KON Only at VBL time, and the chances of a VBL interrupt happening before the VBL vector gets trashed are mighty slim.

BAL You could initialize pointer variables to a bus error number like $50FFC001 instead of NIL. If you did that, you'd crash at the write and know immediately what was wrong.

KON Cool.

Thanks to Scott Douglass 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.•

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W, X, Y, Z