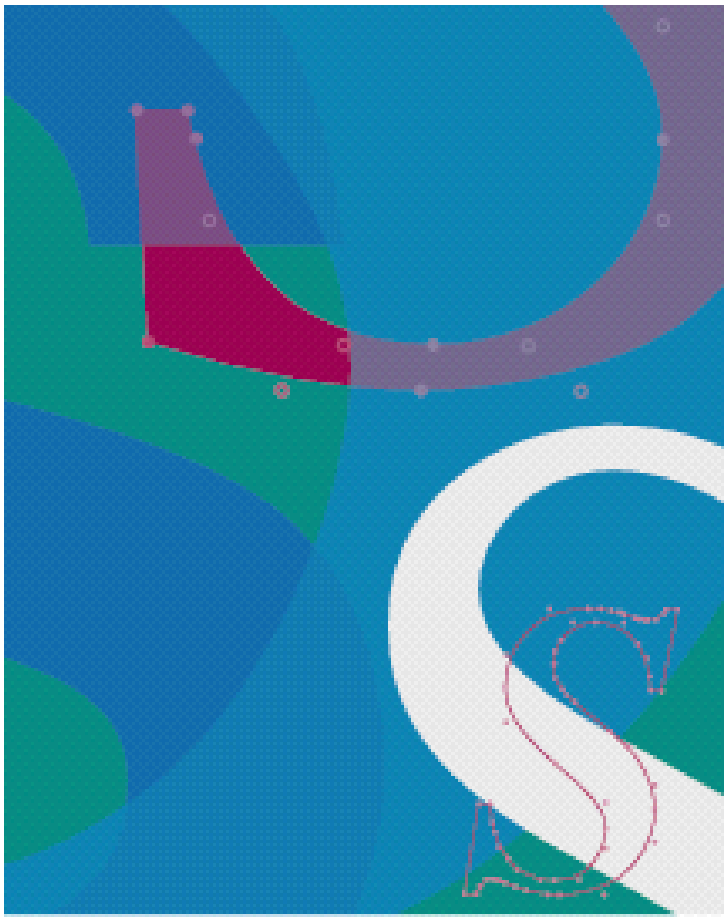
develop

The Apple Technical Journal



Issue 8 Autumn 1991

EDITORIAL STAFF

Editor-in-Cheek *Caroline Rose*

Spirited Guide *Louella Pizzuti*

Technical Buckstopper *Dave Johnson*

Review Board *Pete "Luke" Alexander, Chris Derossi, C. K. Haun, Larry Rosenstein, Andy Shebanow, and Gregg Williams*

Managing Editor *Monica Meffert*

Contributing Editors *Lorraine Anderson, Geta Carlson, Toni Haskell, Judy Helfand, Rilla Reynolds, and 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 *Don Donoughe*

Formatting *Forbes Mill Press*

Printing *Wolfer Printing Company, Inc.*

Film Preparation *Aptos Post, Inc.*

Production *PrePress Assembly*

Photography *Sharon Beals, Ralph Portillo, and Steven C. Johnson*

Circulation Management *David Wilson*

Online Production *Cassi Carpenter*

This sinuous cover artwork was made possible by quadratic Bézier curves and TrueType, and was created by Rucker-Huggins (initial studies by Corinne Okada; final arrangement by Cleo Huggins) using Adobe Illustrator 3.0.

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

The Developer CD Series Disc Volume X accompanies this issue of *develop*. Along with the code described in this issue, it contains all back issues of *develop* and other software and documentation that will make your programming life easier and more interesting.

CAROLINE ROSE

Dear Readers,

I'd like to bring up a subject that's been on my mind lately and a matter of interest to me—and probably to many of you—for quite some time now. It's the subject of what we called the "paperless office" as far back as the Seventies, when I worked in the same group as Doug Engelbart, inventor of the mouse. I laughed to myself back then when I'd hear predictions that in ten years or so, manuals would be obsolete. Who needs information in any form other than electronic? Printed manuals persist, but they're definitely an endangered species. At Apple and many other companies like it, the trend is toward "on-line only" dissemination of information: it uses the technology in zippy ways, it costs less, and it saves trees. Who needs paper, anyway?

I myself don't care if I ever see most memos, notes, reports, and similar daily jottings in print. I prefer to file the majority of this stuff on-line rather than in my physical file cabinets, which I mostly use to hold rice cakes and pistachio nuts. When I need a reminder of some technical information like the meaning of a parameter or the definition of a word, I like quick on-line access to it as much as the next person. And Balloon Help is great when I'm wondering what a particular command or button is for.

But when I don't know how to do something at all, or how different pieces fit together, I prefer to read printed documentation. I'm speaking here of the background material that's needed to get you launched on a particular product in a way that will make you really know what it's about. (Whether this information is needed at all could be the subject of another editorial.) Call it "concepts" versus "reference." For me, there's nothing like reading about concepts in a real book when and where I want. The image that comes to mind is "curling up in front of a fire." I'd much sooner do that with a good novel than a technical manual, but still I like to pick a place and time away from my computer to take in the concepts. It's quieter and more comfortable, especially on my eyes, and it's a more pleasing visual and tactile experience. I learn more that way. Later, I might want to look up some conceptual material on-line, but for first-time reading and learning, I want hard copy.

At the last company I worked for we surveyed a lot of developers on this, and most of them seemed to agree. We decided to divide our technical documentation along those exact lines: concepts versus reference; concepts would always be available as a printed manual while reference would be on-line only (or primarily, at least).

CAROLINE ROSE has been writing computer documentation ever since "peripheral storage" meant paper tape. After a seven-year digression into programming, she returned to writing and joined Apple to document the inner workings of a new computer named Macintosh. In what proved to be another (five-year) digression, she left Apple to launch NeXT Computer's documentation effort—a real learning experience. She's thrilled to be back at Apple among old friends and new. Caroline loves to read, swim, hike, travel, dance, sing, and spend time with her best friend, Cleo (see photo). This summer she got her feet wet (literally) on a backpacking trip in Utah, through a tributary of the Escalante River and some pretty spectacular canyons. Her new wilderness goals are hiking up Half Dome and rafting the Colorado. •

This all ties in with the fact that Apple Associates and Partners no longer receive a printed copy of *develop* as part of their regular mailing; they have to subscribe to *develop* to receive it in print. The mailing has been simplified to be just a CD-ROM disc and a 16-page publication in newspaper format that points to things on the disc. Those developers who don't want a lot of paper don't have to deal with it; those who do can order it. So Associates and Partners will cast their vote for paper *develop* by subscribing to it. (Letters from all of you expressing your opinions—especially when they agree with mine—are of course more than welcome.)

Speaking of electronic media, those of you who were receiving *develop* with its CD-ROM disc bound into it will notice that the disc corresponding to this issue has been packaged separately in its own case (made of partially recycled fiberboard and plastic). This should put a stop to the problem of mangled discs. Also, the disc is no longer the *Developer Essentials* disc, but the *Developer CD Series* disc, the same disc that Apple Associates and Partners receive. We hope this will make life easier for us and less confusing for you (not to mention that you'll get more goodies on the disc than before!).

I'll end with a vaguely related trivia question: What word was used instead of "click" to describe the action of pressing a button on that first mouse? If you've got any good trivia questions of your own, send them along to us. We need all the help we can get.

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

THREADS PACKAGING

I'm impressed with threads and futures and I think they'll be of real use to me in a commercial product I'm working on. I have a simple problem. I don't think we can ship your INIT with our application. Doing so constitutes an unnecessary invitation to support hassles (as well as a just plain unaesthetic package, in my opinion). Is there any chance that you could repackage the current threads and futures stuff as straight libraries (in THINK C form, please)?

—Richard Reich

The latest version of threads and futures, which is on this issue's CD, has been augmented so that you have your choice of how to package the code. Just use the INIT as is, or copy the code resources into your application. Let me know if you need any more help. My number is (408)974-0355.

—Michael Gough

THREADS IN A BLACK BOX

I was really pleased to see the article on threads in Issue 6 of *develop*. There have been several instances in my programming experience on the Macintosh when I had some long involved processing that was not easily restructured to pass control to the interface or to other applications. The Threads Package seems like the best way to handle this problem that we are likely to have without major changes in the Macintosh OS.

However, I was disappointed to discover that Semaphore, one of the examples from the CD, crashes. If you click before the program is finished, it exits normally, but if you let it run its course, it crashes after it's done beeping. I hope that a corrected version of the Threads Package will be available on a future *develop* CD.

If the source code for this package had been provided on the disc, I would at least have a chance of understanding the source of the bug and correcting it. Instead I must rely on the possibility that a corrected version of the object code will be provided with a future issue of *develop*. We can count on Apple to provide updates for object libraries supplied with MPW; hence we have no need for the source. Can we rely on the same level of support for object libraries without source distributed on *develop* CDs?

I've seen *develop* evolve from a journal with good articles on programming techniques into a journal with articles that were basically blurbs for source code, and now into a journal with articles that are blurbs for black boxes provided on the CD. I still think *develop* is an extremely useful resource for Macintosh programmers. However, I urge you to try to include source code when you possibly can.

At any rate I am looking forward to seeing a corrected version of the Threads Package in a future issue of *develop*—I intend to give it a try even if it *is* a black box!

—Dennis C. De Mars

You're right, the Semaphore example crashes, and it is a bug in the Threads Package. I guess I always clicked out early during the final testing. Issue 7 has a follow-up article to Threads that includes an updated (and fixed) Threads Package.

COMMENTS

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 name and company name as well as address and phone number. Letters may be excerpted or edited for clarity (or to make them say what we wish they did). •

To address your concern about develop *heading toward "black boxiness": The Threads Package is the only* develop *article ever that hasn't included source code. We debated for a long time whether that was OK, and many people expressed the same concerns you did. We decided that Threads was unique enough and useful enough to justify it. The reaction has been extremely positive, so I think we did the right thing.*

I assure you that we won't make a habit of featuring black boxes, and that as long as Michael Gough keeps supporting Threads, the latest CD will contain the latest version. Threads is not an official Apple product, though, so use it in your application at your own risk. It's possible that we will be able to publish the source code someday.

Thank you for taking the time to write! People's comments are the best barometer we have for how we're doing.

—Dave Johnson

CD SETUP (AND MORE!) ON CD

How about including on the *Developer Essentials* CD-ROM a copy of the latest version of the *CD Setup* disc that comes with the AppleCD SC drive? It would be a convenient way to get a current version of the AppleCD SC software, and a lot of people who are browsing your CD have an AppleCD SC anyway.

—Kazimir C. Stusinski

The latest version of the CD Setup disc is on this issue's CD. Note that it's now the "Developer CD Series" disc, the same disc that Apple Associates and Partners receive. The contents of Developer Essentials was only a subset of the Developer CD Series disc; now you can have it all!

—Caroline Rose

EVASIVE SNIPPETS

The Snippets section of *develop* always has such neat-sounding code fragments in it . . . if only I could find them! I always peruse the *Developer CD Series* disc that arrives with each issue of *develop*, but many of the snippets can't be found, either by my old-fashioned hand searching or by using the HyperCard® stack that, hopefully, really does have a complete index of the contents.

It's safe to say that it seems that most can't be found. Assuming they *are* there—somewhere—I'd like to suggest that the code you list in the Snippets section be included in the *develop* folder. Even a directory or stack in that folder that would point me to them would be a terrific savings in time. Keep up the good work!

—Chris Gibson

Snippets have had a rough time. They didn't make it onto the Issue 6 CD at all. And, although they were on the Issue 7 CD, they were not mentioned in the Contents Catalog stack on the Developer CD Series disc. They can be found with the DTS sample code. That seems the most logical place for them, since they're not strictly a develop *thing: they come from DTS (Developer Technical Support); we just describe them (and we've decided to stop that—they're now described in a text file in with the snippets). Sorry for the confusion!*

—Dave Johnson

ONE UGLY DUDE

I just wanted to comment that I was totally shocked when I saw Harry Chesley's picture in *develop* this month.

"This is one ugly dude," I thought. Perhaps his picture was sabotaged. Or maybe he really is a Vulcan with three Adam's apples!

—Dan Wood

Actually, that picture wasn't supposed to be printed at all. It's top secret, the result of an internal Apple project in the area of desktop bioengineering.

But now that the cat's out of the bag, I guess I can mention that we're expecting to ship the product second quarter next year. It attaches to the SCSI port of the Macintosh. You stick one of your fingers in a hole in the front, and it modifies your DNA. The actual physiological changes take about a week to materialize.

—Harry Chesley

P.S. The three Adam's apples are a bug, which we're planning to fix in the next rev.

WHERE'S LOUELLA, REALLY?

I sent mail to your predecessor, Louella, at louella@applelink.apple.com, but your system denied knowing about her. Any help you could give me in addressing e-mail to her would be very much appreciated.

—An Admirer

In Issue 7, I joked that Louella had retired to raise flowers in Holland. Well, I was close: she has left Apple to paint and write until her money runs out. We'll miss her terribly but will remember her always as our "Spirited Guide." She'll be living wherever the living is easy (and cheap), but letters will always get to her through this address: 932 Rosette Court, Sunnyvale, CA 94086. Sorry, she will no longer be electronically plugged in.

—Caroline Rose

UPDATES TO CODE ON THE DISC

On the *Developer CD Series* disc, the code that corresponds to articles in *develop* is kept up to date wherever possible. Bugs that are reported are fixed, changes are made to maintain compatibility, and so on. The latest CD always contains the most up-to-date versions of the code for *all* issues of *develop*, so if you're going to use any code, get it from the most recent CD you have—even if the article is in a past issue.

A notable change on this disc is that the QuickTime example from Issue 7 has been updated to work with the beta release of QuickTime, and has a couple of new features as well. Also, we've included the beta QuickTime extension, so you can try the samples immediately. (Remember, this is prerelease software that cannot be distributed or included with shipping applications.)

Another change we've made, to be sure we give you the latest possible information, is to include descriptions of code "snippets" on the disc itself rather than describe them in *develop*. The Snippets folder (which is in with the DTS sample code) now includes text files that describe all of the snippets. The snippets are batched by date of release, so you can easily tell what's new.

CURVES AHEAD: WORKING WITH CURVES IN QUICKDRAW

Imagine being able to build into your application the capability to draw freehand curves. Imagine being able to save these curves so that they can be loaded into other programs like MacDraw or printed using the LaserWriter. And imagine being given the key to an abundant supply of previously defined curves to play with. Imagine no more . . . this article reveals all.

MIKE REED AND KONSTANTIN OTHMER

QuickDraw is a heck of a fine drawing engine for the Macintosh, but it does have its limitations. In particular, it supports a limited set of geometric primitives for drawing: lines, rectangles, rounded-corner rectangles, ovals, arcs, and polygons (see Figure 1). If you want your application to provide the capability of drawing contours consisting of curves and straight lines, you're out of luck.

Sure, you can use the arc primitive to draw a curve, but if you want to connect the curve to anything, you've got a problem. An arc represents a segment of an ellipse and is specified by a bounding rectangle (defining the ellipse) and start and stop angles (see Figure 2). Because an arc is not specified by starting and ending points, it's hard to know the exact points where QuickDraw will begin and end drawing the arc. Thus, the arc does not lend itself to being combined with other arcs or lines.

A more useful curve primitive would be one that describes its start and end positions as points. The quadratic Bézier is just such a curve. Applications such as MacDraw® use this type of curve to allow the drawing of freehand curves, and the Macintosh itself uses this type of curve in an internal procedure to describe TrueType fonts.

In this article we give you the lowdown on the quadratic Bézier. We show the coding and the data structures used by programs like MacDraw to draw this kind of curve, and we show how your application can interchange data about this kind of curve with MacDraw and with devices equipped with a PostScript® interpreter. And since the quadratic Bézier happens to be the same curve that TrueType uses (in combination with other shapes) to draw outline fonts, we show how to extract curve data from TrueType fonts.

7

MIKE REED AND KONSTANTIN OTHMER have become such regular contributors to *develop* that they scarcely need introduction. Still, we've just discovered something new about them: they dabble in doggerel. Consider this sample:

Late into the night, by the glow of a candle,
Two men are found working on mischief and
scandal.
Their mice are a-clicking, their keyboards in motion,
They're working on something of mythic proportion.
We move closer in, to get a good look,
And notice they're writing a get-rich-quick book.
We wonder what topic could hold their attention
And keep them from working on some new
invention.
Their title reveals what the work will envelop—
How to Get Rich: Just Write for develop.•

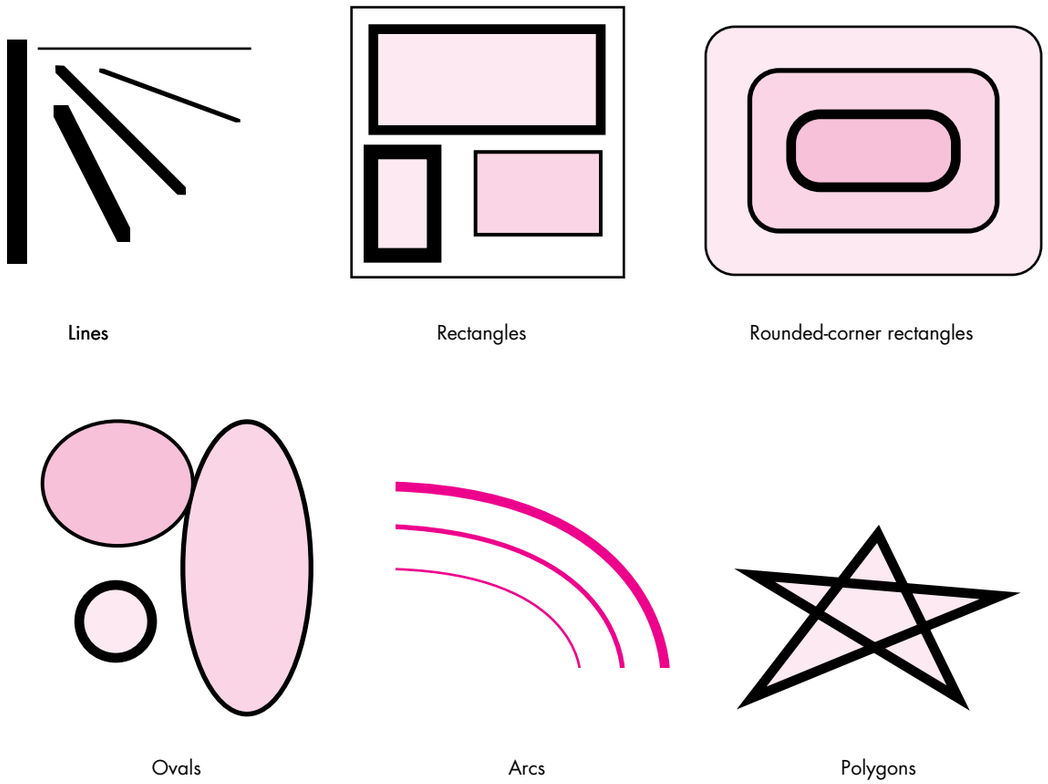


Figure 1
QuickDraw's Geometric Primitives

DRAWING CURVES AND PATHS

The quadratic Bézier has a couple of properties that make it useful as a basis for drawing curves in QuickDraw. First, it can be rotated easily by changing just the starting, ending, and middle points and not the underlying equation itself. Second, it can easily be subdivided into any number of shorter curves that become flatter and flatter, until in effect it can be drawn with a series of straight lines. Indeed, the basic technique for drawing a curve using the existing QuickDraw primitives is by subdividing the curve into a series of line segments. If you're interested in the mathematics behind this, see "Parametric Equations, Anyone?"

This section begins by showing sample C code that implements the subdivision algorithm that produces a curve. We then move on to consider how to produce a

```
{  Rect r;

   SetRect(&r, 0, 0, 144, 144);
   FrameArc(&r, 0, 90);
   PenPat(gray);
   FrameRect(&r);
}
```

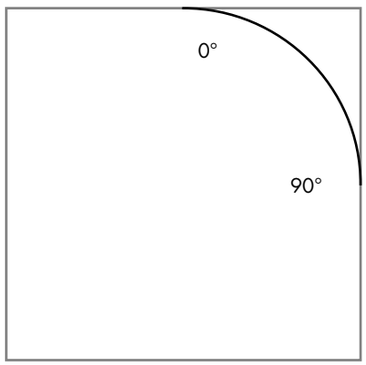


Figure 2
How an Arc Is Specified in QuickDraw

combination of curves and straight lines, known in the lingo as a *path*. Then we talk about how to combine paths to produce shapes. Note that for a curve, as for every geometric primitive in QuickDraw, you always have two options: you can either frame it or fill it. We show you how to do the framing; you can do the filling using the call FillPoly or PaintPoly.

FRAMING A CURVE

The code that implements the subdivision algorithm to produce a curve takes a value for the number of times the curve should be subdivided before it's drawn as straight lines. This number can be dynamically computed based on the size of the curve and the quality-versus-speed trade-off the application wants to make. The code uses fixed-point coordinates to maintain high precision during the subdivisions.

To begin, let's define a few macros to help us use fixed-point coordinates in an integer-based graphics system.

```
#define FR(x)            ((x) + 0x8000 >> 16)
#define ff(x)            ((long)(x) << 16)
#define fmoveto(x,y)     MoveTo(FR(x), FR(y))
#define flineto(x,y)     LineTo(FR(x), FR(y))
#define AVE(a,b)         (((a) + (b)) / 2)
```

FR	The same as FixRound: takes a Fixed and makes it a short.
ff	The reverse of FixRound: takes a short and promotes it to a Fixed.
fmoveto	The same as MoveTo, but takes Fixed coordinates.
flineto	The same as LineTo, but takes Fixed coordinates.
AVE	Averages two numbers, either Fixed, short, or long.

For an explanation of the Fixed data type, see *Inside Macintosh* Volume I, Chapters 3 and 16.•

PARAMETRIC EQUATIONS, ANYONE?

Though you can draw curves without understanding the mathematics behind the operation, some people find this kind of thing interesting. This explanation is for those people (you know who you are).

A quadratic Bézier (or parabola) can be defined by the parametric equation

$$f(t) = a(1 - t)^2 + 2bt(1 - t) + ct^2$$

where t is a scalar and a, b, and c are points.

This parametric formulation has the advantage of being rotationally independent: since t is the independent variable, and not x or y, there is no bias for curves that are symmetric about the x- or y-axis. Thus, to rotate the curve, you only need to rotate a, b, and c into their new positions, while the equation for the curve stays the same.

To better understand the equation, take a look at its geometric representation in Figure 3. You'll note there that as the curve is drawn from point a to point c, t varies from 0 to 1. The curve at a is tangential to the line ab, and the curve at c is tangential to the line bc. Its maximum displacement from the line that could be drawn from a to c is reached at point q, where t is 0.5. In addition, the curve at q is parallel to the (imagined) line ac.

Perhaps the most useful property of the curve in this form is the easy way it can be decomposed into a pair of smaller curves, each of the same form. This is called *subdivision* and is the basis for drawing curves in QuickDraw.

Suppose we subdivide the curve at point q, as shown in Figure 4. The point q is

$$q = f(0.5) = (a + 2b + c) / 4$$

The tangent (that is, the first derivative) of the curve at q is parallel to the line ac.

$$f'(t) = -2a(1 - t) + 2b(1 - 2t) + 2ct$$

$$f'(0.5) = c - a$$

The line from the point $b' = (a + b) / 2$ to q is tangential to the curve at q.

$$q - b' =$$

$$(a + 2b + c) / 4 - (a + b) / 2 =$$

$$(a + 2b + c - 2a - 2b) / 4 =$$

$$(c - a) / 4$$

By symmetry, the same holds for $b'' = (b + c) / 2$.

Thus, the formulas for the two curves that make up the left and right halves of the original curve are as follows:

Left: $a' = a$

$b' = (a + b) / 2$

$c' = (a + 2b + c) / 4$

Right: $a'' = (a + 2b + c) / 4$

$b'' = (b + c) / 2$

$c'' = c$

The equations for the new points are especially nice, since the arithmetic can be performed with only shifts and adds, making it very efficient.

Notice in Figure 4 that each of the resulting smaller curves is flatter than the original curve, by a factor of 4. This means that the distance from the midpoint of the curve to the midpoint of the straight line drawn from the start point to the endpoint is reduced by 4. Thus, if the curve is subdivided enough times, drawing the curve will be equivalent to drawing a line from the start point to the endpoint for each of the little curves. This is the basis for drawing the curve in QuickDraw.

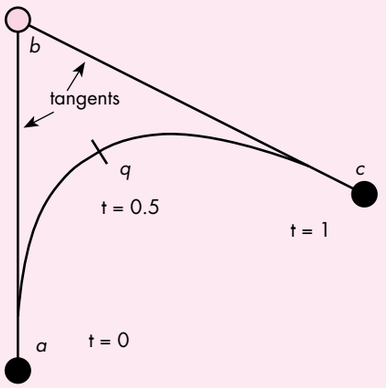


Figure 3
Geometric Representation of Our Bézier

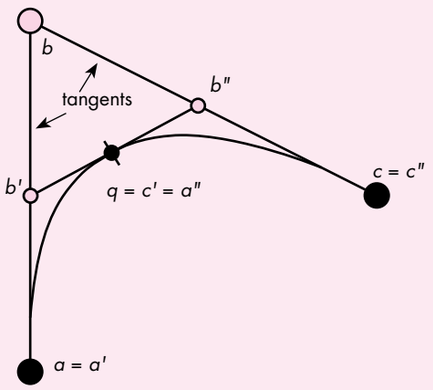


Figure 4
Our Bézier, Subdivided

To represent fixed-point coordinates, we define a struct called point. Note that this is similar to QuickDraw's Point, but uses Fixed numbers instead of integers.

```
typedef struct {
   Fixed x;
   Fixed y;
} point;
```

To represent a curve, we need three points: a start point, a control point, and an endpoint. (These correspond to *a*, *b*, and *c* in Figure 3.)

```
typedef struct {
   point start;
   point control;
   point end;
} curve;
```

The function FrameCurve (below) draws a quadratic Bézier using subdivision. If the level for the FrameCurve routine is 0, a LineTo (using flineto) operation is performed; otherwise, the curve is subdivided and FrameCurve is called recursively, once for the left half of the curve and once for the right. FrameCurve assumes the caller has already called fmoveto on the start point of the curve. The second routine, ExampleCurve, calls FrameCurve requesting four levels of subdividing. Thus, the final curve consists of 2^4, or 16, lines. It's shown in Figure 5.

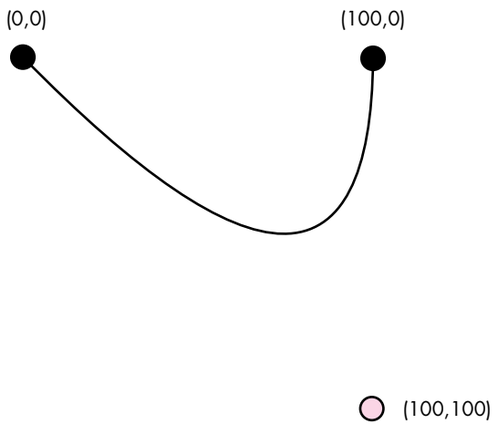


Figure 5
The Curve Drawn by ExampleCurve

```
void FrameCurve(curve *cur, int level)
{
   if (level)
   {  curve left, right;

      left.start = cur->start;
      left.control.x = AVE(cur->start.x, cur->control.x);
      left.control.y = AVE(cur->start.y, cur->control.y);
      right.control.x = AVE(cur->control.x, cur->end.x);
      right.control.y = AVE(cur->control.y, cur->end.y);
      left.end.x = right.start.x = AVE(left.control.x,
         right.control.x);
      left.end.y = right.start.y = AVE(left.control.y,
         right.control.y);
      right.end = cur->end;

      FrameCurve(&left, level-1);
      FrameCurve(&right, level-1);
   }
   else
      flineto(cur->end.x, cur->end.y);
}
```

```
void ExampleCurve()
{
   static curve myCurve = {ff(0), ff(0), ff(100), ff(100), ff(100),
      ff(0)};

   fmoveto(myCurve.start.x, myCurve.start.y);
   FrameCurve(&myCurve, 4);
}
```

FRAMING A PATH

Drawing contours such as font outlines requires drawing a combination of straight lines and curves. Such a combination is known as a *path*. A path is defined by the following data structure:

```
typedef struct {
   long  vectors;              /* The number of points in the path. */
   long  controlBits[anyNumber];
   point vector[anyNumber];    /* The points. */
} path;
```

A path is similar to a polygon except that it has a set of control bits that determine whether each point is on or off the curve. There's one control bit for each point, beginning with the most significant bit for point 0. If the bit is set, the corresponding point is an off-curve point and therefore the control point for a curve. If the bit is clear, the corresponding point is an on-curve point and therefore an endpoint for either a line segment or a curve segment. Two consecutive on-curve points form a straight line.

Here's a routine that takes an index and the control bits and returns TRUE (nonzero) if the point is on the curve:

```
Boolean OnCurve(long *bits, long index)
{
   bits += index >> 5;   /* Skip to the appropriate long. */
   index &= 31;          /* Mask to get index into current long. */
   return (*bits & (0x80000000 >> index)) == 0;
}
```

Two consecutive off-curve points imply an on-curve point at their midpoint, as shown in Figure 6. This path consists of two curve segments. The first is defined by *a*, *b*, (*b* + *c*) / 2 and the second by (*b* + *c*) / 2, *c*, *d*.

This ability to store a series of off-curve points allows a path to describe an arbitrarily complex shape without having to store unneeded intermediate points. However, this is just a storage nicety. When we draw the path, we need it broken down into a series

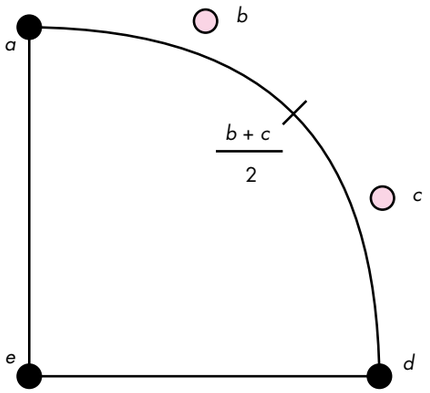


Figure 6
On-Curve Point Implied by Two Off-Curve Points

of line and curve segments. This is done with an iterator function called NextPathSegment. It's called continuously, each time filling a record that is either a line segment or a curve segment, until it returns FALSE.

```
typedef struct {
   int   isLine;
   curve c;

   /* Private. */
   long  index;
   long  ep;
   long  *bits;
   point *p;
} pathWalker;

void InitPathWalker(pathWalker *w, path *aPath)
{
   w->index = 0;
   w->ep = aPath->vectors - 1;
   w->bits = aPath->controlBits;
   /* Skip past the control bits to point to the first point. */
   w->p = (point *)(w->bits + (aPath->vectors + 31 >> 5));
}

int NextPathSegment(pathWalker *w)
{
   long prevIndex, nextIndex;
```

```
   if (w->index == 0)     /* 0 means this is the first segment. */
   {  if (OnCurve(w->bits, w->ep))
         w->c.start = w->p[w->ep];
      else
      {  if (OnCurve(w->bits,0))
         {  w->c.start = w->p[0];
            w->index = 1;
         }
         else  /* Start at an implied on-curve point. */
         {  w->c.start.x = AVE(w->p[0].x, w->p[w->ep].x);
            w->c.start.y = AVE(w->p[0].y, w->p[w->ep].y);
         }
      }
   }
   else  /* Start where we previously left off. */
      w->c.start = w->c.end;

NEXT_SEGMENT:
   /* Compute the point index before and after the current one.
    * This wraps around, since we assume the contour is closed. */
   prevIndex = w->index == 0 ? w->ep : w->index - 1;
   nextIndex = w->index == w->ep ? 0 : w->index + 1;

   if (OnCurve(w->bits, w->index))
   {  if (OnCurve(w->bits, prevIndex))
      {  w->isLine = true;      /* This means we have a line. */
         w->c.end = w->p[w->index];
      }
      else if (w->index++ <= w->ep)
         goto NEXT_SEGMENT;
   }
   else
   {  w->isLine = false;     /* This means we have a curve. */
      w->c.control = w->p[w->index];
      if (OnCurve(w->bits, nextIndex))
         w->c.end = w->p[nextIndex];
      else
      {  w->c.end.x = AVE(w->p[w->index].x, w->p[nextIndex].x);
         w->c.end.y = AVE(w->p[w->index].y, w->p[nextIndex].y);
      }
   }

   return w->index++ <= w->ep;  /* Return TRUE if there are still
                                 * more segments. */
}
```

The FramePath routine uses a pathWalker to traverse the path and draw it as it goes.

```
path *NextPath(path *aPath)
{
   return (path *)((long *)aPath + 1 + (aPath->vectors + 31 >> 5) +
      aPath->vectors * 2);
}

path *FramePath(path *cont)
{
   pathWalker walker;

   InitPathWalker(&walker, cont);
   /* The first segment is special, since it calls fmoveto. */
   if (NextPathSegment(&walker))
   {  fmoveto(walker.c.start.x, walker.c.start.y);
      if (walker.isLine)
         flineto(walker.c.end.x, walker.c.end.y);
      else
         FrameCurve(&walker.c, kCurveLimit);
   }
   /* Keep looping until we run out of segments. */
   while (NextPathSegment(&walker))
      if (walker.isLine)
         flineto(walker.c.end.x, walker.c.end.y);
      else
         FrameCurve(&walker.c, kCurveLimit);
   /* Return the next path, used if this path is one of several within
    * a series of paths. */
   return NextPath(cont);
}
```

Now we can draw the path shown in Figure 6 that demonstrates consecutive off-curve points.

```
void ExamplePath()
{
   long myPath[] = {
      5,              /* Five points. */
      0x60000000,     /* The second and third are off-curve points. */
      0,0,ff(10),0,ff(20),ff(10),ff(20),ff(20),0,ff(20) /* x,y data */
   };

   FramePath((path *)myPath);
}
```

FRAMING A SHAPE MADE OF SEVERAL PATHS

To describe a shape that contains several disjoint paths (such as an outline letter *o*), we use a simple data structure that's just a composite of several path structures:

```
typedef struct{
   long  contours;
   path  contour[anyNumber];
} paths;
```

Drawing such a shape (called in the vernacular a *paths*) is straightforward:

```
void FramePaths(paths *aPath)
{
   long ctr = aPath->contours;
   path *cont = aPath->contour;

   while (ctr--)
      cont = FramePath(cont);
}
```

The following code draws the paths shown in Figure 7.

```
void ExamplePaths()
{
   long myPaths[] = {
      5,          /* Five contours. */
      3, 0xE0000000, 0, ff(16), 0, ff(8), ff(14), ff(12),
      3, 0xE0000000, ff(8), 0, ff(16), 0, ff(12), ff(14),
      3, 0xE0000000, ff(24), ff(8), ff(24), ff(16), ff(10), ff(12),
      3, 0xE0000000, ff(16), ff(24), ff(8), ff(24), ff(12), ff(10),
      16, 0x11110000,
           ff(8), 0, ff(12), ff(4), ff(16), 0, ff(16), ff(8),
           ff(24), ff(8), ff(20), ff(12), ff(24), ff(16), ff(16), ff(16),
           ff(16), ff(24), ff(12), ff(20), ff(8), ff(24), ff(8), ff(16),
           0, ff(16), ff(4), ff(12), 0, ff(8), ff(8), ff(8)
   };

   FramePaths((paths *)myPaths);
}
```

SAVING PATHS IN PICTS

Now that you know how to give your application the capability to draw all sorts of curved shapes on the screen or on a QuickDraw printer, you might wonder whether you can cut and paste these shapes into other applications or send them to a

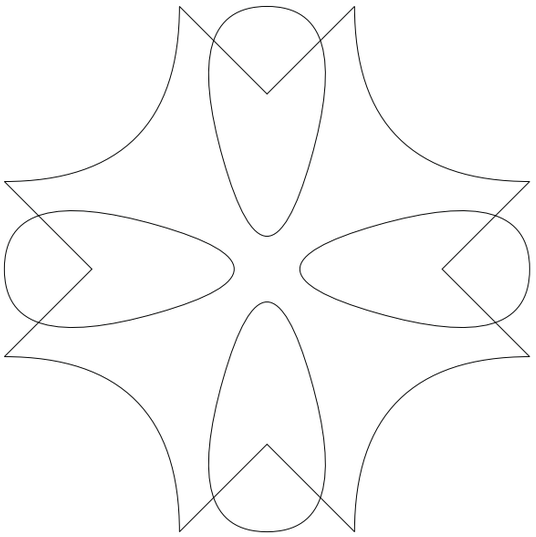


Figure 7
A Shape Made of Several Paths

PostScript printer. The answer is yes, thanks to picture comments. Picture comments encapsulate non-QuickDraw graphics data that other "smarter-than-QuickDraw" applications can interpret.

Fortunately, there's a picture comment that takes quadratic Bézier information directly: PolySmooth. When this comment is encountered in a PICT it indicates that the endpoints of the following lines are control points of a quadratic Bézier. Unfortunately, the comment assumes that all the control points lie off the curve. This is a major drawback of the PolySmooth comment and forces us to break a path down into curves and lines, rather than allowing us to put an entire path in at once.

When a picture interpreter (such as the LaserWriter driver) sees a PolySmooth picture comment, it interprets the following points (put in the picture with a LineTo opcode) as off-curve control points. Since DrawPicture ignores picture comments completely, the clipping rectangle is set to empty so that no drawing will occur. The picture interpreter now has the control points and can render the curve however it sees fit.

To save a path in a picture, we start with the PolyBegin picture comment. This comment indicates that a special polygon follows. If the path is closed, we add the picPlyClo comment. Then we set the clipping to empty so that DrawPicture will not

For more information about picture comments, see Technical Note #91, "Optimizing for the LaserWriter—Picture Comments." This tech note explains how picture comments work and documents a number of standard picture comments supported by the LaserWriter printer driver.•

render the following data. Next we add the PolySmooth comment, followed by the control points.

When the whole polygon is in the picture, we restore the clipping to its previous state and add the PolyIgnore picture comment. Anyone reading picture comments will know to ignore the following QuickDraw polygon. DrawPicture, which ignores all picture comments and skips over the smooth polygon since the lines are all clipped out, will draw the polygon just as it should.

Thus, we have something like the following (where an asterisk indicates a picture comment):

Save the current clip
*PolyBegin
Set the clip to the empty rectangle to turn off drawing
*PolySmooth
Record the PostScript version of the curves (just the control points)
*PolyIgnore
Restore the original clip to turn drawing back on
Record the QuickDraw version of the curves (subdivided into short lines)
*PolyEnd

Since the PolySmooth picture comment allows only off-curve points, it's necessary to break the path down into segments. This is done by the AddPathsToPict routine, which calls AddSegmentToPict for each path fragment (either a single quadratic Bézier or a line). AddSegmentToPict copies two sets of points into the PICT, one that contains the actual control points of the curve or line segment (for PostScript printing or pasting into MacDraw) and another that is the QuickDraw rendering of the curve or line.

In AddSegmentToPict (below), cur is the current segment to be added; isLine is a Boolean identifying whether the segment is a curve or a line segment; and delta specifies an amount to offset the data when recording the PostScript version, to account for the difference between PostScript's centered pen and QuickDraw's upper left pen.

```
void AddSegmentToPict(curve *cur, point *delta, int isLine)
{
   /* Real programs check errors. */
   Handle       verbHdl = NewHandle(1);
   RgnHandle    origClip = NewRgn();
   Rect         emptyRect = {0, 0, 0, 0};
```

```
    **verbHdl = kPolyFrame;
    GetClip(origClip);

    PicComment(POLYBEGIN, 0, 0);
    ClipRect(&emptyRect);     /* This turns drawing off. */
    PicComment(POLYSMOOTH, 2, verbHdl);
       /* Record the endpoints for PostScript. */
       fmoveto(cur->start.x + delta->x, cur->start.y + delta->y);
       if (isLine)
          flineto(cur->end.x + delta->x, cur->end.y + delta->y);
       else
          flineto(cur->control.x + delta->x, cur->control.y + delta->y);
       flineto(cur->end.x + delta->x, cur->end.y + delta->y);

    PicComment(POLYIGNORE, 0, 0);
    SetClip(origClip);        /* This turns drawing back on. */
       /* Record the lines for QuickDraw. */
       fmoveto(cur->start.x, cur->start.y);
       if (isLine)
          flineto(cur->end.x, cur->end.y);
       else
          FrameCurve(cur, kCurveLimit);
    PicComment(POLYEND, 0, 0);

    DisposeRgn(origClip);
    DisposHandle(verbHdl);
}
```

AddPathsToPict is relatively simple. It walks through each path, and each segment within each path, and records the segments by calling AddSegmentToPict.

```
AddPathsToPict(paths *myPaths)
{
    point penDelta;
    long  i;
    path  *cont;

    /* Compute half the pen's thickness as a delta, since PostScript's
     * pen is centered and QuickDraw's hangs to the right and down. */
    penDelta.x = ff(thePort->pnSize.h) / 2;
    penDelta.y = ff(thePort->pnSize.v) / 2;

    /* Record the curve data. */
    cont = myPaths->contour;
```

```
   for (i = 0; i < myPaths->contours; i++)
   {  pathWalker walker;

      /* This loop looks a lot like FramePath. */
      InitPathWalker(&walker, cont);
      while (NextPathSegment(&walker))
         AddSegmentToPict(&walker.c, &penDelta, walker.isLine);
      cont = NextPath(cont);
   }
}
```

A word about rounding: We've kept all our data in Fixed, even during the subdivision process, up until calling LineTo; still, when we record the data into a PICT, we're forced to throw away information since the PICT records only integer coordinates. The upshot of this is that a given series of paths may draw much better in your application than when it's been put into a PICT and pasted into another application.

EXTRACTING CURVES FROM TRUETYPE

Because TrueType uses the quadratic Bézier as its curve primitive, as mentioned earlier, the outlines in a TrueType font represent a rich source of curve data for programmers. In fact, a program demonstrated at Apple's Worldwide Developers Conference in May of this year uses TrueType fonts as a basis for turning text typed by the user into outlines that can be rotated, filled, stretched and shrunk, and transformed in other amusing ways. All it takes to produce such a program is to convert TrueType data from its native storage structure into a paths data structure. We show you how to do that here, and then discuss the sample program you'll find on the *Developer CD Series* disc that draws outlines extracted from TrueType fonts.

CONVERTING THE DATA

For space reasons, the data for TrueType outlines is not stored in a paths data structure but instead is compressed as byte deltas. Code provided on this issue's *Developer CD Series* disc fills out a glyph outline data structure given a TrueType font and a glyph ID. The glyph outline data structure looks like this:

```
typedef struct {
   long     contourCount;
   long     pointCount;
   point    origin;
   point    advance;
   short    **endPoints;    /* [contourCount] */
   Byte     **onCurve;      /* [pointCount] */
   Fixed    **x;            /* [pointCount] */
   Fixed    **y;            /* [pointCount] */
} GlyphOutline;
```


Complete information on the TrueType format is provided in *The TrueType Font Format Specification*, available from APDA (#M0825LL/A).•

A glyph outline is a bit less compact than a paths data structure, but contains enough information to be converted into one. One difference from the paths data structure is that in a path, if a control bit is set, the point is treated as an off-curve point. In a glyph outline, if the onCurve byte is set, the point is treated as an on-curve point. Another difference is that in a glyph outline, the points for all the contours are stored contiguously, rather than separated into discrete path structures.

The utility function PackControlBits takes an array of bytes, each representing a point, and packs them into a bit array, suitable for a path. It then returns a pointer to the long after the last control long.

```
long *PackControlBits(long *p, Byte *onCurve, long count)
{
   unsigned long mask = 0x80000000;

   *p = 0;
   while (count--)
   {  if (!mask)
      {  mask = 0x80000000;
         *++p = 0;
      }
      if (!*onCurve++)
         *p |= mask;
      mask >>= 1;
   }
   return p + 1;
}
```

The function OutlineToPaths takes a glyph outline and returns a pointer to a path that represents the outline. NewPtr is called, so when the application has finished using the path it should call DisposePtr.

```
paths *OutlineToPaths(GlyphOutline *out)
{
   long size, *p, *origP;

   /* First compute how big the resulting path will be. */
   size = sizeof(long);  /* paths.contours */

   {  long i, sp = 0;
      for (i = 0; i < out->contourCount; i++)
      {  long pts = (*out->endPoints)[i] - sp + 1;
         size += sizeof(long);          /* path.vectors */
         size += (pts + 31 >> 5) << 2;  /* path.controlBits */
         size += pts << 3;              /* path.vector[] */
```

```
            sp = (*out->endPoints)[i] + 1;
         }
      }

      /* Now allocate the paths. */
      origP = p = (long *)NewPtr(size);
      /* Real programs check errors. */

      /* Record the number of contours. */
      *p++ = out->contourCount;
      {  long  i, sp = 0;
         Fixed *x = *out->x;
         Fixed *y = *out->y;
         short *ep = *out->endPoints;
         Byte  *onCurve = *out->onCurve;

         /* For each contour, record the point count,
          * record the control bits, then the points. */
         for (i = 0; i < out->contourCount; i++)
         {  long pts = *ep - sp + 1;
            *p++ = pts;
            p = PackControlBits(p, onCurve, pts);
            onCurve += pts;
            while (pts--)
            {  *p++ = *x++;
               *p++ = *y++;
            }
            sp = *ep++ + 1;
         }
      }
      return (paths *)origP;
}
```

ABOUT OUR SAMPLE APPLICATION

The sample application QD Curves on the *Developer CD Series* disc uses paths to draw the outlines of TrueType text. It can put the outlines onto the Clipboard so that they can be pasted into another application for editing or printing. In addition, the application uses some of the other TrueType access functions to build variable-length font and style menus and display information about a font, such as its version number, copyright notice, and trademark (see Figure 8).

To display a path, the application determines how large the path currently is and scales it to fill the window. The utility functions OffsetPaths, ScalePaths, and GetPathsBounds are used in positioning and scaling paths.

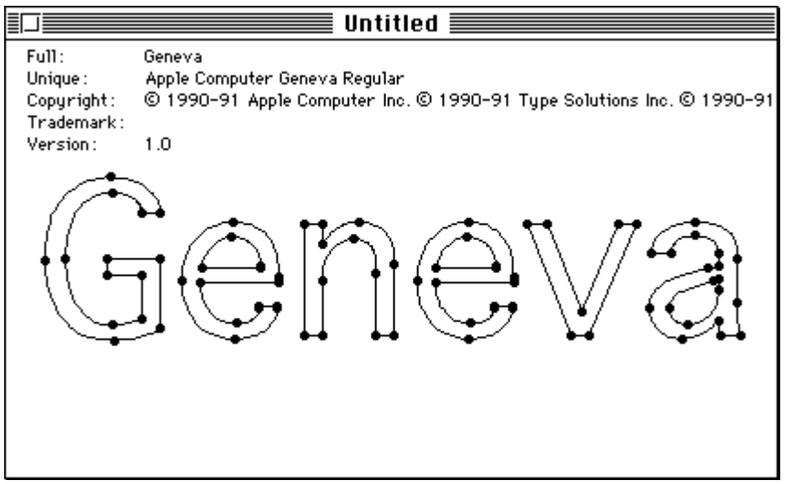


Figure 8
Font Information Displayed by QD Curves

```
void OffsetPaths(paths* p, Fixed dx, Fixed dy)
{
   long ctrs = p->contours;
   path *aPath = p->contour;

   while (ctrs--)
   {  long  pts = aPath->vectors;
      /* Skip the control bits. */
      Fixed *coord = (Fixed *)aPath + 1 + (pts + 31 >> 5);

      /* Apply the offsets; remember, x comes before y. */
      while (pts--)
      {  *coord++ += dx;
         *coord++ += dy;
      }
      /* The next path follows the end of the current path. */
      aPath = (path *)coord;
   }
}

void ScalePaths(paths *p, Fixed sx, Fixed sy)
{
   long ctrs = p->contours;
   path *aPath = p->contour;
```

```
   /* Apply the scales; remember, x comes before y. */
   while (ctrs--)
   {  long  pts = aPath->vectors;
      /* Skip the control bits. */
      Fixed *coord = (Fixed *)aPath + 1 + (pts + 31 >> 5);

      while (pts--)
      {  *coord = FixMul(*coord, sx);
         coord++;
         *coord = FixMul(*coord, sy);
         coord++;
      }
      /* The next path follows the end of the current path. */
      aPath = (path *)coord;
   }
}

void GetPathsBounds(paths *p, Rect *r)
{
   long ctr = p->contours;
   path *cont = p->contour;

   /* Begin with the minimum rectangle. */
   r->left = r->top = 32767;
   r->right = r->bottom = -32678;

   while (ctr--)
   {  long  *bits = cont->controlBits;
      /* Skip the control bits. */
      long  *coord = (long *)(bits + (cont->vectors + 31 >> 5));
      long  pts = cont->vectors;

      while (pts--)
      {  short x = FR(*coord++);
         short y = FR(*coord++);

         if (x < r->left)
            r->left = x;
         else if (x > r->right)
            r->right = x;
         if (y < r->top)
            r->top = y;
         else if (y > r->bottom)
            r->bottom = y;
      }
```

```
        /* The next path follows the end of the current path. */
        cont = (path *)coord;
    }
}
```

Note that what is returned is the bounds for the control points of the paths, not necessarily the bounds of the actual paths being drawn (see Figure 9). That requires a slightly more complex, though useful, function, which we leave to you as an exercise. (Hint: Find the *x* and *y* extrema for the curve. To do this, find the local extrema in *t* by setting the derivative of the equation in *x* or *y* equal to 0, and solve for *t*.)

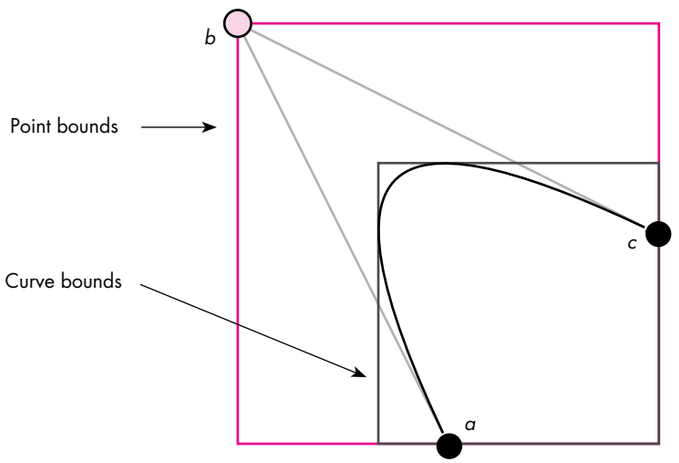


Figure 9
Bounds of the Control Points Versus Bounds of the Curve

As part of its display options, the application will also mark the on-curve points in the paths.

```
void MarkPaths(paths *aPath)
{
    long   ctr = aPath->contours;
    path   *cont = aPath->contour;
    Point loc;
    Rect   r;

    while (ctr--)
    {   long   *bits = cont->controlBits;
        long   *coord = (long *)(bits + (cont->vectors + 31 >> 5));
        long   ptIndex;
```

```
      for (ptIndex = 0; ptIndex < cont->vectors; ptIndex++)
      {  r.left = FR(*coord++) - 2;
         r.top = FR(*coord++) - 2;
         r.right = r.left + 5;
         r.bottom = r.top + 5;
         if (OnCurve(bits, ptIndex))
            PaintOval(&r);
#ifdef mark_off_curve
         else
            FrameOval(&r);
#endif
      }
      cont = (path *)coord;
   }
}
```

This function to mark the control points of a path is a good framework for adding curve editing. We leave that to you also, and suggest that besides simple point-by-point direct mouse editing, you consider providing the capability to do the following:

- do direct curve editing by hit-testing the curve itself instead of just its control points
- select groups of points/segments to move at once, similar to selecting multiple icons in the Finder
- do constrain-based editing, where tangent continuity is maintained between adjacent segments
- interpolate changes to a set of key points across the rest of the path, allowing, for instance, creation of a bold character by simply moving a few points on the stems and then smoothing out the rest of the outlines

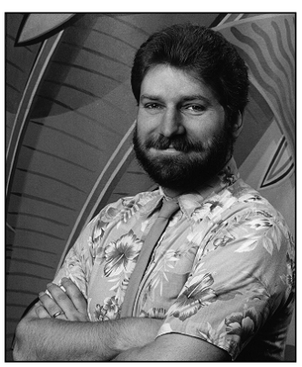
NOW IT'S UP TO YOU

This article has given you the tools to do some fancy work with curves in your applications. We've given you routines for drawing curves and paths using QuickDraw, discussed how to save these in PICTs so that they can be loaded into other programs like MacDraw or printed using the LaserWriter, and shown how to extract paths from TrueType fonts, giving you an abundant supply of path data. Now it's up to you to play off this knowledge by adding curve editing, text rotation, and other means of transforming your new freehand curves.

THANKS TO OUR TECHNICAL REVIEWERS
Pete "Luke" Alexander, Guillermo Ortiz, Sean Parent•

VALIDATING DATE AND TIME ENTRY IN MACAPP

MacApp's TEditText class checks strings entered by the user, displaying an error message when an invalid string is encountered. This article shows how TEditText's validation and error notification schemes can be made more flexible, and demonstrates this flexibility in TEditText subclasses for the entry of dates and times. You can try out these classes in the sample program that's on the Developer CD Series disc.



JAMES PLAMONDON

My favorite high school teacher, Mrs. Whalen, had a sign under the wall clock in her classroom that read "Time passes—but will you?" Back when I was in the Class of '78, there were many times I wished that I could set the clock back (during a tough quiz) or forward (on a warm afternoon). Although I can't offer any such hope to the Class of '91, I can at least provide MacApp developers with classes that make the entry of dates and times as easy as I ever dreamed.

I wrote these classes during some recent work on a MacApp application that involved the entry and validation of dates and times. After considering and rejecting all sorts of controls—controls that looked like little monthly calendars, 24-hour clocks, and so on—I settled on simple editable text boxes. I thought that with these boxes, those pesky localization issues that plagued the other designs wouldn't be a problem, because I could use the Macintosh Script Manager to handle the different date and time formats described by the international resources in the operating system. I also figured that if I used MacApp's TEditText class, writing editable text boxes for date and time entry would be trivial. An override here, a little data there, and voilà—done. It wasn't the first time I've been wrong.

But to understand TEditText's flaws, first you have to know how it works.

TEDITTEXT REVEALED

TEditText is a TControl subclass. It encapsulates the Toolbox's TextEdit routines. A TEditText view is to one of MacApp's TDialogViews what an editText item is to one

JAMES PLAMONDON was born, is hanging around for a while, and will soon die, like everybody else. But he's trying to have fun in the meantime. As a software engineer at Power Up Software Corporation of San Mateo, California, he has worked on an as yet unreleased MacApp product he can't talk about, and he has recently begun working on the Microsoft Windows version of this product—but he can't talk about that either. The founder of the Bay Area MacApp Developer's Association (BAMADA), his interests include raising his four kids, following international politics, writing the Great American Computer Game, and trying to convince Apple to port MacApp to Windows. (If only he could get his kids to write the international version of his game using MacApp for Windows, his life would be complete.)•

of the Dialog Manager's dialogs: it allows the user to enter strings into a box in a dialog box. In addition, TEditText extends the functionality of editable text items to include the notion of validation. If an invalid string is entered into a TEditText view, an alert is displayed, notifying the user of the problem.

The validation process implemented by TEditText centers on its Validate method. In TEditText, a valid string is any string that's not longer than the maximum allowed length, which is specified by the application's author. If the string is valid, the Validate method returns the constant value kValidValue; otherwise—that is, if the string is too long—it returns the error code kTooManyCharacters. TNumberText, a subclass of TEditText that handles the entry and validation of integer numbers, can return additional error codes—kValueTooSmall, kValueTooLarge, or kNonNumericCharacters.

The only place Validate is ever called in MacApp is from the TDialogView method DeselectCurrentEditText. If Validate returns a value other than kValidValue, that value is assumed (in a call to TDialogView.CantDeselect) to be an index into a string list resource called kInvalidValueReasons. It's expected that the string at that index will describe the error encountered. This string is then displayed in an error alert that tells the user why the string entered is invalid. Figure 1 shows the alert displayed when the user types too many characters into a TEditText view.

Figure 1
MacApp's Validation Error Alert

My dad used to say "Whenever a guy's telling you what he's gonna do *for* you, start worrying about what he's gonna do *to* you." It wasn't long before I realized that TEditText was like that. I had hoped that it would be easy to extend the checking done in TEditText.Validate to include checking for a valid date or time, but it wasn't. To add this kind of checking, I was going to have to rewrite Validate from scratch—just the kind of thing object-oriented programming is supposed to prevent.

DON'T FIX WHAT AIN'T BROKE

When in the course of application programming it becomes necessary to replace a mechanism written by the MacApp engineers, one should declare the causes that impel this decision. I hold these truths to be self-evident:

1. That the reuse of existing code is preferable to the addition of new code.
2. That the addition of new code is preferable to the alteration of existing code.
3. That the alteration of existing code is preferable to missing a deadline.

MacApp's approach to text validation fails to meet a number of these criteria. First, it assumes that new error strings will simply be added to the STR# resource called kInvalidValueReasons, with new error codes indexing the added strings. However, this won't work: TDialogView.CantDeselect uses a constant, kNoOfDefaultReasons, to indicate the number of strings in this resource. It can only be changed by altering and recompiling MacApp—a violation of self-evident Truth #2.

Also, the error-code-equals-string-index scheme can be a problem when one combines existing class libraries; two different TEditText subclasses, written independently, may use the same error codes (and string indices) to indicate different problems. Resolving this conflict would probably require changing and recompiling at least one of the conflicting classes.

Further, the use of error strings can cause problems during localization since not all languages can stick an arbitrary string into a sentence and have the result make any sense. Static error strings also give little context—they may not be able to display the invalid string, or a valid example string, to help the user figure out what went wrong.

For all of these reasons, MacApp's use of a single error string list—with Validate's result being used as an index into this list—seems inappropriate. Each class should instead build its own error strings in any manner it sees fit, using its own string lists as necessary.

That's not all. The error alert displayed when invalid strings are encountered has only one button. But what if two or more alternative actions can be taken in response to the entry of an invalid string?

Consider the following validation case (which has nothing to do with dates or times). Assume that the user needs to enter the name of a category—like Work, School, or Personal—into an editable text box. If the string the user enters matches the name of an existing category (for example, "Work"), the string is valid; otherwise—for example, if the user types "Wirk"—the string is invalid.

In addition, we want to allow the user to add new categories to the list by entering them into the editable text box. To do this, we must distinguish those entries that are simply mistyped (like “Wirk”) from those intended to become new category names (like “School” or “Personal”). In effect, we need to present the user with a two-button dialog box like that in Figure 2.

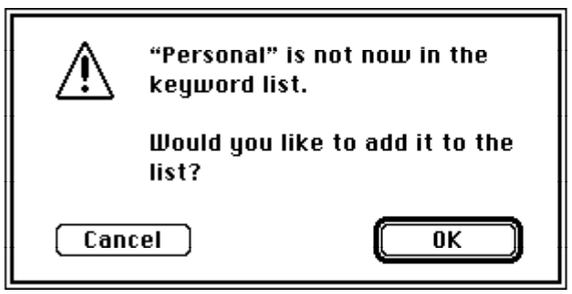


Figure 2
Two-Option Validation Error Alert

Unlike the default MacApp validation alert, which has only one button, the dialog box in Figure 2 allows the user to decide whether the entered string—“Personal,” in this case—is valid or invalid.

TVALIDTEXT: THE UNAUTHORIZED BIOGRAPHY

So to extend validation to dates and times, I decided to write two new classes, TDateEditText and TTimeEditText. After writing these classes, I realized that they had so much validation code in common that it made sense to put this code in a common superclass. I called this superclass TValidText.

TValidText is a pretty simple extension to TEditText. It adds three notions to TEditText—strictness, required value, and an invalid entry alert ID. It also significantly enhances the text validation process.

TValidText is an example of an “abstract class”—a class that’s never expected to be instantiated directly. It exists only to factor out the code that’s expected to be common to its subclasses. All of TValidText’s subclasses will simply inherit its validation and error reporting code, while overriding a few methods to implement their own specific validation tests and error messages.

The class declaration for TValidText is as follows:

WHAT ABOUT MACAPP 3.0?

MacApp is an evolving system, just like the Macintosh itself. With the coming of System 7, a new and more powerful version of MacApp is in the works: MacApp 3.0.

Totally rewritten in C++, this new version of MacApp is still in development; however, an early version, MacApp 3.0b1, is available to Macintosh developers on the *Essentials•Tools•Objects* CD #5.

MacApp 3.0 will offer a lot of new features, most of them directed toward support for System 7. A discussion of most of these features is far beyond the scope of this article. With regard to text validation, MacApp 3.0's display of validation error alerts is much improved.

In MacApp 3.0, the TargetValidationFailed method has been added to TEvtHandler to allow each class to handle validation errors in a class-specific manner. An override of TargetValidationFailed in TDialogTEView calls another override of the same method in TEditText, which displays the error alert and restores the TEditText's previous contents. Thus the validation error alert is posed by TEditText, not by TDialogView, as was the case in MacApp 2.0. That's a big improvement.

Unfortunately, the nature of TEditText's Validate routine remains unchanged in MacApp 3.0. If your application requires more flexible validation than that provided by MacApp, you may still need to use some of the techniques described in this article.

```
TValidText = OBJECT(TEditText)
      fStrict:     BOOLEAN;
      fRequired:   BOOLEAN;
      fAlertID:    INTEGER;
      •
      •  See the Developer CD Series disc for method declarations.
      •
END;  {TValidText}
```

TValidText's fStrict field, a Boolean variable, determines whether or not strict checking will be used when validating. This field exists here because both the date and time classes needed the concept. TValidText itself doesn't use fStrict, except to get and set its value. It might be more general to implement it as a scalar (maybe a signed byte) to provide multiple strictness levels. We'll look at strictness again in the discussion of the date and time classes later in this article.

The fRequired field answers the question of whether an empty string is valid or not. As far as TValidText is concerned, if fRequired is true, an empty string is invalid; otherwise, it's valid. TValidText's subclasses may add additional conditions to the notion of validity by overriding the method IsValid and calling the inherited version. Both the date and time editing classes do this, as we'll see later.

The fAlertID field contains the resource ID of the alert to be displayed when the current text doesn't pass validation. It may contain the value phInvalidValue (defined

The fRequired field might better be called fEmptyStringValid or something to this effect. I called it fRequired to match a comment in the method TNumberText.Validate in the UDialog unit of the MacApp 2.0 source code.•

in UDialog), or the resource ID of any other ALRT resource. It would be easy to override the routines involved to display a MacApp dialog box rather than a Toolbox alert, in which case fAlertID could be the ID of the appropriate view resource.

THE NATURE OF VALIDITY

The TValidText declaration introduces a new method, IsValid:

```
FUNCTION  TValidText.IsValid(
                VAR   theText:        Str255;
                VAR   whyNot:         INTEGER)
                      :BOOLEAN;
```

In addition to returning a Boolean indicating the validity of the given string, IsValid returns in whyNot an indication of why the string is invalid (or the value noErr, if it's valid). This is very similar in functionality to TEditText's Validate routine, with one major difference: the string being validated is passed in as an argument. Where TEditText.Validate assumes that it's supposed to validate the string currently being edited, TValidText.IsValid can be used to test arbitrary strings for validity.

I overrode the Validate method in TValidText to make it a flow-of-control method. It validates the current string and displays the error alert when necessary, as follows:

```
FUNCTION  TValidText.Validate:LONGINT; OVERRIDE;
   VAR
      parentResult:   LONGINT;
      theText:        Str255;
      whyNot:         INTEGER;

   BEGIN
   {Make sure the current text passes the superclass's validation.}
   parentResult := INHERITED Validate;
   IF (parentResult <> kValidValue)
   THEN
      Validate := parentResult
   ELSE
      BEGIN
      GetText(theText);
      IF IsValid(theText, whyNot)
      THEN
         Validate := HandleValidText(theText)
      ELSE
         Validate := HandleInvalidText(theText, whyNot);
      END;  {else}
   END;  {Validate}
```

This structure places the responsibility for handing invalid cases in the class itself, rather than relying on MacApp's code for mapping error codes to error strings in TDialogView.CantDeselect (never trust a method with a contraction in its name). Given this structure, you can change any step in the validation process without changing the nature of validation itself by overriding IsValid, HandleValidText, or HandleInvalidText. That's the whole idea behind flow-of-control methods.

HandleValidText simply returns kValidValue (defined in UDialog), after notifying its superview that the text is valid. Two lines of code—no fuss, no muss.

HandleInvalidText has to do a little more, but not much. It calls the method ValidationErrorAlert to notify the user of the problem. Although the default alert has only an OK button, I've also added support for a Cancel button. If the user clicks OK, HandleAlertAccepted is called; otherwise—if the user clicks Cancel—HandleAlertCancelled is called.

```
FUNCTION  TValidText.HandleInvalidText(
                VAR   theText:     Str255;
                      theError:    INTEGER)
                      :LONGINT;
   BEGIN
   IF ValidationErrorAlert(theText, theError)
   THEN
      HandleInvalidText :=
            HandleAlertAccepted(theText, theError)
   ELSE
      HandleInvalidText :=
            HandleAlertCancelled(theText, theError);
   END;  {HandleInvalidText}
```

In either case, a handler routine is called. Again, this kind of flow-of-control method, which calls other methods to do the dirty work, is a very useful addition to the object programmer's repertoire.

ValidationErrorAlert is equally trivial, consisting of only two lines. The first is a call to PrepareErrorAlert, while the second displays the alert itself, returning TRUE if the user accepts the dialog box and FALSE if the user cancels out of it.

PrepareErrorAlert is also only two lines of code:

```
PROCEDURE TValidText.PrepareErrorAlert(
                VAR   theText:     Str255;
                      theError:    INTEGER);
   {This routine sets up the dialog that is displayed by
    ValidationErrorAlert.}
```

```
VAR
   theString:      Str255;

BEGIN
{Get the best string to describe the given error.}
ErrorToString(theError, theString);
ParamText(theString, '', '', '');
END;  {PrepareErrorAlert}
```

PrepareErrorAlert converts the given error code to a string by calling ErrorToString, and then calls ParamText to get the string into the dialog. The error code was generated by IsValid way back in the Validate method.

The essential feature of these routines is that they're all teeny-tiny pieces of code, each with a single, well-defined goal. Any one of them can be overridden in isolation, to tweak the validation mechanism one way or another. I think you'll find it to be a big improvement over TEditText's validation mechanism.

VALIDATING THE DATE

The TDateEditText class allows the user to enter a date string and have the Macintosh Script Manager's LongDateTime routines figure out what date it is, in a convenient, internationally compatible manner. It can display the resulting date in any of the three formats supported by the Script Manager: short (9-13-91), abbreviated (Fri, Sep 13, 91), or long (Friday, September 13, 1991).

TDateEditText overrides four TValidText methods to implement date validation: IsValid, HandleValidText, ErrorToString, and PrepareErrorAlert.

IsValid looks pretty complicated, and it is—by my standards, anyway. It has to set up not only its Boolean return value, but also an error code if the given text is not valid. This latter chore is complicated by the optional strict checking, embodied in fStrict. The Script Manager provides two different levels of error messages when converting dates (and times) to strings. It will take almost anything you give it and make a date out of it, but it will warn you about leftover characters, nonstandard separators, and the like. Strict checking for dates means that only perfectly formed date strings will be accepted, while nonstrict checking means that so long as a date can be extracted from the string, you don't want to hear the Script Manager complain about how hard it was to get it.

```
FUNCTION  TDateEditText.IsValid(
                VAR   theText:     Str255;
                VAR   whyNot:      INTEGER)
                      :BOOLEAN;
                      OVERRIDE;
```

```
VAR
   theError:    INTEGER;
   valid:       BOOLEAN;
   dateSecs:    LongDateTime;

BEGIN
IF (NOT INHERITED IsValid(theText, theError))
THEN
   BEGIN
   valid  := FALSE;
   whyNot := theError;
   END
ELSE
   BEGIN
   {Use the Script Manager to convert the date string to a
    LongDateTime.}
   theError := StringToDate(theText, dateSecs);
   IF fStrict
   THEN
      valid := (theError = noErr) | (theError = longDateFound)
   ELSE     {Error codes >= noErr mean a valid date was found.}
      valid := (theError >= noErr);
   IF (theError = dateTimeNotFound) &     {Date isn't found,}
      (NOT fRequired) &                   {empty strings are OK,}
      (Length(theText) = 0)               {and the string is empty.}
   THEN     {Empty string is OK if entry isn't required.}
      valid := TRUE;
   IF valid
   THEN
      whyNot := noErr
   ELSE
      whyNot := theError;
   END;  {else}
IsValid := valid;
END;  {IsValid}
```

HandleValidText just sets the fDateSecs instance variable to reflect the date of the given string, and then calls the inherited version of the HandleValidText routine in TValidText.

Likewise, ErrorToString catches those errors that it knows about and converts them to strings; others, it just passes on to the inherited version of ErrorToString. Don't you love inheritance?

```
PROCEDURE TDateEditText.ErrorToString(
                  theError:    INTEGER;
            VAR   theString:   Str255);
                  OVERRIDE;
   {This routine sets theString to the string that best
    explains the given error. It's intended to be called
    only from PrepareErrorAlert.}
   VAR
      strIndex: INTEGER;

   BEGIN
   CASE theError OF
      {These are the error codes returned by the Script Manager's
       string-to-date routine.}

      {strIndex 1 contains the default string, "invalid date".}
      leftOverChars:        strIndex :=  2;
      sepNotIntlSep:        strIndex :=  3;
      fieldOrderNotIntl:    strIndex :=  4;
      extraneousStrings:    strIndex :=  5;
      tooManySeps:          strIndex :=  6;
      sepNotConsistent:     strIndex :=  7;
      tokenErr:             strIndex :=  8;
      cantReadUtilities:    strIndex :=  9;
      dateTimeNotFound:     strIndex := 10;
      dateTimeInvalid:      strIndex := 11;

      OTHERWISE             strIndex :=  0; {Not our error.}
      END;  {case theError}
   IF (strIndex > 0)
   THEN  {It's an error we know how to describe, so handle it.}
      GetIndString(theString, kInvalidDateReasons, strIndex)
   ELSE  {Never heard of it - ask our superclass to handle it.}
      INHERITED ErrorToString(theError, theString);
   END;  {ErrorToString}
```

TDateEditText.PrepareErrorAlert (below) calls ErrorToString to convert the given error code to a string. This string will then be displayed in the validation error alert (see Figure 3). It also converts the current system date to a string to be displayed in the alert, where it will serve as an example of the proper date format.

```
PROCEDURE TDateEditText.PrepareErrorAlert(
                VAR   theText:        Str255;
                      theError:       INTEGER);
                      OVERRIDE;
```

```
VAR
   errString:      Str255;
   dateSecs:       LongDateTime;
   dateString:     Str255;

BEGIN
{Get the current date, as a string.}
GetCurrentDate(dateSecs);
DateToString(dateSecs, shortDate, dateString);

{Get the best string to describe the given error.}
ErrorToString(theError, errString);
ParamText(errString, dateString, '', '');
END;  {PrepareErrorAlert}
```

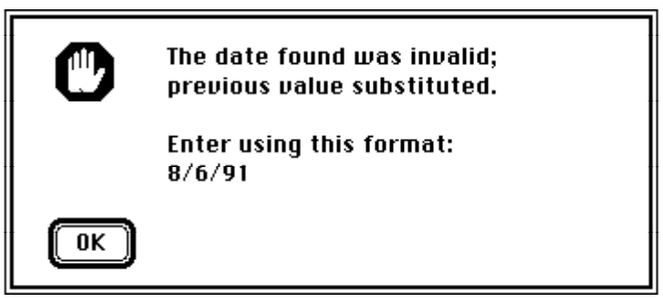


Figure 3
TDateEditText's Validation Error Alert

TDateEditText.PrepareErrorAlert can't call the version of PrepareErrorAlert it inherits from TValidText. The inherited version's call to ParamText would cloud the effect of the override's call. I got around this by duplicating the body of TValidText.PrepareErrorAlert in the override, and not calling the inherited version at all. This duplication is a violation of Truth #1 (reusing code is better than writing new code), but I couldn't figure out how to avoid it—so I just duplicated it, thus adhering to Truth #3 (anything's better than missing a deadline).

Eventually, if an invalid date like February 31 has been entered by the user, TDateEditText displays an alert similar to that shown in Figure 3.

VALIDATING THE TIME

Given the description and discussion of TDateEditText above, the most striking thing about TTimeEditText is its similarity to TDateEditText. That shouldn't be

surprising. The validation of dates has been designed to follow a particular series of steps, which can also be applied to time. Validating time therefore involves subtly tailoring the behavior of a few steps rather than writing the validation logic from scratch. You can see this in the code on the *Developer CD Series* disc, which includes a test application and its source.

WHAT'S DONE IS DONE

If you build the test application yourself and bang on it even a little, you'll find a bug: if you've got invalid text in one editText item, and click on the other, you'll see the invalid entry alert twice. This is very annoying. Fixing this problem requires a change to MacApp, however, because that's where the bug lives. Consider MacApp's TEditText.HandleMouseDown method:

```
FUNCTION TEditText.HandleMouseDown(
                        theMouse:    VPoint;
                VAR     info:        EventInfo;
                VAR     hysteresis:  Point;
                VAR     theCommand:  TCommand)
                        : BOOLEAN;
                        OVERRIDE;
   BEGIN
   {Get the floating TE installed if necessary.}
   IF IsViewEnabled & (gTarget <> fTEView)
   THEN
      DoChoice(SELF, fDefChoice);
   HandleMouseDown := INHERITED HandleMouseDown(
                    theMouse, info, hysteresis, theCommand);
   END;  {HandleMouseDown}
```

The call to DoChoice goes to TDialogView, which attempts to deselect the currently edited item with a call to its Validate method. If validation fails, the validation failure alert is displayed. The subsequent call to INHERITED HandleMouseDown eventually calls DoMouseCommand. This call, in turn, creates and returns a control tracker that eventually calls TDialogView.DoChoice again. TDialogView.DoChoice again attempts to deselect its currently edited item, and the validation again fails (since nothing has changed), displaying the invalid entry alert for the second time.

To fix the problem, we must add an override of DoMouseCommand to TEditText. Just overriding DoMouseCommand in TValidText won't fix the problem, since the flaw is in TEditText itself.

```
FUNCTION TEditText.DoMouseCommand(
                    VAR    theMouse:    Point;
                    VAR    info:        EventInfo;
```

```
                    VAR   hysteresis:  Point)
                          : TCommand;
                          OVERRIDE;
BEGIN
IF (gTarget = fTEView)    {Only true when validation succeeds.}
THEN      {Validation has succeeded, so Do the Right Thing.}
   DoMouseCommand := INHERITED DoMouseCommand(
                                theMouse, info, hysteresis)
ELSE      {Validation failed - stop cold.}
   DoMouseCommand := NIL;
END;  {DoMouseCommand}
```

Thus INHERITED DoMouseCommand is called only when all is as it should be. (I'd like to thank Tom Dinger for suggesting this solution, which is cleaner than my originally proposed change to TEditText.HandleMouseDown.) I added this change to my copy of MacApp, and used it to build the compiled version of the sample application you'll find on the *Developer CD Series* disc—so it demonstrates the fix, not the bug.

. . . AND NOW, THE QUIZ

That about wraps up TValidText, TDateEditText, and TTimeEditText. Their use is demonstrated further in the accompanying sample program. You can use the TValidText class as a common basis for the validation of any quantity-related editable text control—such as controls for numbers, currency, and weights and measures—in the same uniform and flexible manner. If you have any questions, I'm sorry, but your time is up: please put down your pencils, and pass your papers forward. Class dismissed!

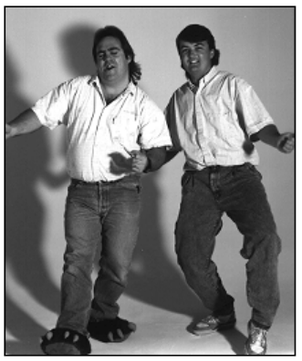
ACKNOWLEDGMENTS

This article wouldn't have been possible without the support of Steve Starr, Marian Cauwet, or Ed Lauing, who have made Power Up Software such a great place to work, or my family, who have made my home such a great place to live. To the former, I give my thanks and respect; to the latter, my love. Many thanks also to my editor, Geta Carlson.

Special thanks to the MacApp engineers, past and present, for writing such a great piece of work. If inflexible string validation is the worst flaw I can find in MacApp, it must be pretty good. And thanks to our technical reviewers: Jesse Feiler of The Philmont Software Mill, author of articles on topics related to this one; Carl Nelson, President of Software Architects and former President of the MacApp Developer's Association (MADA); Bryan Stearns, Macintosh guru and Tech Note author; and David Taylor of Bear River Associates, author of Calendar Creator 1.0, the first shipping retail application written using MacApp 2.0.

THANKS TO OUR TECHNICAL REVIEWERS

Jesse Feiler, Carl Nelson, Bryan Stearns, David Taylor•



Luke speaks

PRINT HINTS FROM LUKE & ZZ

COPYMASK, COPYDEEPMASK, AND LASERWRITER DRIVER 7.0

With the release of System 7 comes a new release of the LaserWriter driver, version 7.0. Yes, the great implementors (GIs) have once again created another version of this driver. This version supports TrueType fonts, it's 32-bit clean, it has the new PostScript file-saving capability, and it remembers the last setting of the Black & White and Color/Grayscale print buttons (HOORAY!!).

Along comes our hero, Dudley Developer. He's been using the new CopyMask and CopyDeepMask calls that are available in QuickDraw in System 7 and he's very excited about printing his new images with the LaserWriter driver. He assumes that since QuickDraw supports the new CopyMask and CopyDeepMask calls, the LaserWriter driver 7.0 will also support them.

Bad assumption. Our hero has not been keeping up with current events. He has not even read "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0" in Issue 6 of *develop*. If he had, he would know that, like previous versions of the LaserWriter driver, LaserWriter driver 7.0 does not directly support the CopyMask and CopyDeepMask calls.

So, in his ignorance, our hero creates a few pictures with the new QuickDraw calls, and sends them off to the LaserWriter. Time goes by, paper comes out, but the picture doesn't look the same as it did on his monitor. It has lost some of the cool effects from CopyMask and CopyDeepMask.

Why, he wonders, won't LaserWriter driver 7.0 print his images with the same effects provided by the CopyMask and CopyDeepMask calls? What was Apple thinking when they created LaserWriter driver 7.0? How could they release a driver that doesn't support the new 32-bit QuickDraw calls? How do they expect him to print his new cool pictures created in System 7 with these calls?

To attempt to understand the problem, Dudley looks between the covers of the Adobe red book (a.k.a. *PostScript Language Reference Manual* by Adobe Systems). Even he knows transfer modes would be required to support the new calls—but alas, he doesn't find any information on them, because PostScript level 1 doesn't understand transfer modes. Unfortunately, the LaserWriter driver won't rewrite PostScript for you; it just merrily converts your QuickDraw calls into their equivalent PostScript call. The LaserWriter driver always uses the srcCopy transfer mode when it prints a pixel map, regardless of the mode used when the picture was created. (Why srcCopy? Because of the limitations of the color model and the lack of transfer mode support provided by PostScript level 1.)

There *is* a method that will allow Dudley to print the images he created, but as usual with printing, he'll need to do a little more work: he'll need to use GWorlds and PrGeneral. Since our hero is not familiar with using GWorlds, he decides to go back through his old issues of *develop*, hoping for an article. Luckily, he finds just what he's looking for in Issue 1: "Realistic Color for Real-World Applications" and "Braving Offscreen Worlds." And the CD contains the sample code that uses GWorlds.

So, after reading the articles and trying the code, Dudley is all set to create an off-screen world to hold his image. He realizes that to get the best print quality, he'll need to make the GWorld bigger than the picture on the screen. The GWorld should be the size of the

PETE "LUKE" ALEXANDER After taking almost a year to check out Ford Explorers in parking lots and dealerships, Luke has, in a surprise move, actually purchased one of his own. At SIGGRAPH in Las Vegas he ran into some Ford engineers; they were surprised, but not seriously hurt. They asked incredulously, "Have you driven a Ford lately?" Luke left wondering what kind of inside information they had. He knew for sure, though, that the animation was his favorite part of the show—which figures, since since he's so used to working with Zz, our resident cartoon character. Not surprisingly, the Las Vegas show he talked about the most had little to do with computers! He also enjoys the show at Gordon Biersch, a local brew pub known for its good beer and yuppie clientele. Luke says he goes there to see how yuppie scum lives, but every once in a while we think he already knows.•

grafPort returned by the application's call to PrOpenDoc, at the printer's resolution. Dudley knows he needs a device-independent method to acquire this information, and he knows he can get it by using PrGeneral's GetRslData opcode. And even better still, he remembers an article about using PrGeneral from Issue 3 of *develop*. The article was titled "Meet PrGeneral," and it contained complete sample code. Yippee! After using the GetRslData opcode to determine the resolutions supported by the currently selected printer, he uses the SetRsl opcode to set the printer to the resolution he wants. When his application calls PrOpenDoc, the printer driver will return a grafPort that's sized correctly for the resolution he chose.

Next, Dudley creates the correctly sized off-screen world and draws his image using CopyMask or CopyDeepMask. He just needs to remember that since the CopyMask and CopyDeepMask calls are not saved in pictures, he needs to make the calls directly into his off-screen world (not relying on DrawPicture). So, when he's ready to print his image, Dudley uses CopyBits to copy it from his GWorld into the printer's grafPort with srcCopy. That's it; his totally cool image has been printed in living color (or anemic gray scales, if he's printing to the LaserWriter).

This wasn't so bad, but our hero is wondering—when will the GIs make this easier? When will all of this incompatibility between QuickDraw and the LaserWriter driver improve? That's a really good question. The GIs tell me things will probably not improve until the new printing architecture is released. In the meantime, you've got this way around the problem, and at least it isn't too ugly!

REFERENCES

- "QuickDraw's CopyBits Procedure: Better Than Ever in System 7.0," *develop* Issue 6, Spring 1991.
- "Meet PrGeneral, the Trap That Makes the Most of the Printing Manager" *develop* Issue 3, July 1990.
- "Realistic Color for Real-World Applications," *develop* Issue 1, January 1990.
- "Braving Offscreen Worlds," *develop* Issue 1, January 1990.
- *Inside Macintosh*, Volume V, Color Manager chapter, Addison-Wesley, 1988.

MACINTOSH DEBUGGING: A WEIRD JOURNEY INTO THE BELLY OF THE BEAST

Macintosh debugging is a strange and difficult task. This article provides a collection of tried-and-true debugging techniques Bo3b and Fred discussed at Apple's Worldwide Developers Conference in May 1991. These techniques can ease your debugging woes and make your life a lot simpler. They're guaranteed to help you find your bugs earlier on, saving you hours of suffering.

BO3B JOHNSON AND FRED HUXHAM

ADAPTED FROM THEIR TALK AT THE WWDC BY DAVE JOHNSON

The first thing you should know is that *debugging is hard.* Drinking gallons of Mountain Dew won't help much, nor will seeking magic formulas or spreading fresh goat entrails around your keyboard and chanting. The only way to get better at it is to do it a lot, and even then it's still hard. What we're going to talk about are a number of techniques that will make debugging a little bit easier.

Notice that the title of this article is "Macintosh *Debugging*" and not "Macintosh *Debuggers*." We're not going to do a comparative review of debuggers. We're not going to show you how to use them. In fact, we recommend that you buy and use *all* the ones described here. Each has useful features that the others don't have. Which you use most often is up to you—pick one as your main debugger and really get to know it, but keep all of them around.

The main Macintosh debuggers are

- MacsBug from Apple
- TMON (we often refer to version 2.8.4 as Old TMON) and TMON Professional (version 3.0, called TMON Pro for short) from Icom Simulations, Inc.
- The Debugger from Jasik Designs (we'll call it "Jasik's debugger" here, because Steve Jasik wrote it, and that's what everybody calls it in conversation)

We'll touch on many of the individual features of these debuggers in this article.

BO3B JOHNSON AND FRED HUXHAM didn't want a bio, except to say that they are cohosts of "Lunch with Bo3b and Fred." We also feel compelled to tell you that in Bo3b's name, the "3" is silent.•

THE INSIDE STORY OF THE DEBUGGER

BY STEVE JASIK

WHY WRITE A DEBUGGER

Since I didn't have the right connections for selling illegal drugs, I had to consider the alternative of selling legal addictive drugs to Macintosh developers.

OK, seriously, I wanted to learn about the 68000 architecture. Given my experience writing compilers and code generators for superscalar RISC mainframes, I decided to write a disassembler for and on the Macintosh. I introduced my first product, MacNosy, in January 1985. It allowed a fair number of developers to discover the innards of the Macintosh ROMs, as well as to curse at me for its original TTY interface.

Unhappy with the state of Macintosh debuggers, I decided to write one of my own, using MacNosy as a foundation. The resulting product, The Debugger, made its international debut in London in November 1986. Since then, it's been expanded to become a system debugger (it runs at INIT time and is available to debug any process), include an incremental Linker for MPW compiled programs, and more.

THE MACINTOSH INTERFACE

The Debugger uses the Macintosh user interface, or at least my interpretation of it. The windows, menus, dialogs, and text processing are standard for the Macintosh.

The only real problem was the switch in context. I had to swap in all of low memory ($0 to $1E00 on a Macintosh II-class machine). This may appear to be a bit expensive, but in comparison with the screen swap, which is a minimum of 22K on a small-screened Macintosh, it's trivial. The biggest problem in this area is that some of the values have to be "cross-fertilized" between worlds, and many of the low-memory globals are *not* documented.

Using the Macintosh interface became a royal pain as the System 7 group extended the system in such a way that the basic ROM code assumed the existence of a Layer Manager and MultiFinder functions. In many cases, I had to "unpatch" the standard code and substitute my own in order to keep The Debugger functional.

MMU PROTECTION

MMU protection was initially designed so that The Debugger would try to protect the system from destruction no matter what program was running. As we implemented the design, we found that this goal was impossible because many of the applications (MPW Shell, ResEdit, Finder) diddled with the system heap. I ended up protecting the rest of the system only when an application that's being debugged is running.

EASE OF USE

Users have had an influence on the design and feature set in The Debugger. For example, the initial version of the watchpoint (memory watch) command was very simple. When a user pointed out the usefulness of an auto reset feature in the command, we added it.

I've tried to use simple commands for the most frequently performed operations in The Debugger. The idea has been to make common things easy to do. Some of the more complicated operations are difficult to keep simple, as the scripting capability is limited. SADE, in contrast, has an extensive scripting capability but is cumbersome to use.

The hardest bugs to find are those that are not reproducible. If you have a crashing bug that can be reproduced 100 percent of the time, you're well on your way to fixing it. But a bug that crashes your application only once every few hours, at seemingly random times . . . well, that kind can take days or weeks to find. Often the ultimate

failure of a program is caused by code that executed long ago. Tracing back to find the real problem can be difficult and extremely time consuming.

The techniques we show you in this article will help turn most of your random bugs into completely reproducible ones. These techniques are designed to make your software crash or to otherwise alert you as close as possible to where your code is doing something wrong.

We explain what each technique is, why it works, and any gotchas you need to be aware of. Then we tell you how to turn it on or invoke it and list some of the common Macintosh programming errors it will catch. Finally, we show a code sample or two. The code samples were chosen for a number of reasons:

- The errors in many of them are subtle. We couldn't tell what was wrong with some of them after not looking at them for a couple months, and we wrote them in the first place.
- The mistakes are common. We've seen people make these same mistakes time and time again.
- They're short. They had to fit on one slide at our Worldwide Developers Conference presentation.

So, on to our first technique . . . .

SET $0 TO $50FFC001

The basic idea here is that the number $0 comes up a lot when things go wrong on the Macintosh. When you try to allocate memory or read in a resource, and it fails, what gets returned is $0. Programs should always check to see that when they ask for something from the Toolbox, they actually get it.

Programs that don't check and use $0 as an address or try to execute from there are asking for trouble. The code will often work without crashing, but presumably it's not doing what it was meant to do, since there isn't anything down there that even remotely resembles resources or data in a program.

Why $50FFC001? Old TMON used this number when we turned on Discipline (more on Discipline later). This fine number has the following characteristics:

- Used as a pointer (address), $50FFC001 is in funny space on all Macintosh computers—that is, it's in I/O space, which is currently just blank. Any relative addresses close by are going to be in I/O space as well, so positive or negative offsets from that as a base will crash, too. These types of offset are common when referencing globals or record fields.

- When used as an address, it will cause a bus error on 68020, ’030, and ’040 machines. Because there’s no RAM there, and no device to respond, the hardware returns a bus error, crashing the program at the exact instruction. Without this handy number, you not only won’t crash, you won’t even know the bug exists (for a while . . .).
- On 68000 machines, $50FFC001 will cause an address error because it’s an odd number. This also stops the offending code at the exact line that has a bug.
- If the program tries to execute the code at memory location $0, it will crash with an illegal instruction, since the $50FF is not a valid opcode. This is nice when you accidentally JSR to $0 and the program tries to run from there. Those low-memory vectors are certainly not code but don’t usually cause a crash until much later.
- It’s easy to recognize because it doesn’t look like any normal number. If a program uses memory location $0 as a source for data, this funny number will be copied into data structures. If you see it in a valid data structure someplace else you know there’s a bug lurking in the program that’s getting data from $0 instead of from where it should.

Many different funny bus error numbers can be used. Take your pick.

AVAILABILITY

You can find various programs that set up memory location $0 in this helpful way, or you can build your own.

- EvenBetterBusError (included on the *Developer CD Series* disc) is a simple INIT that sets memory location $0 to $50FFC003. It also installs a VBL to make sure no one changes it.
- Under System 7, the thing that used to be MultiFinder (now the Process Manager) takes whatever is in memory location $0 when it starts up and saves it. Periodically it stuffs that saved number back in. If it were a bus error number at system startup (from an INIT, say), that number would be refreshed very nicely. With MacsBug, it would be easy to build a dcmd that stuffs memory location $0 during MacsBug INIT, and MultiFinder would then save and restore that number.
- Jasik’s debugger has a flag that allows you to turn the option on or off.
- Old TMON will set up the bus error number when Discipline is turned on. TMON Pro has a script command, NastyO, that will also do this.

- You can put code in your main event loop that stuffs the bus error number into memory location $0. Be sure to remove it before you ship.

ERRORS CAUGHT

The most obvious catch using this technique is the inadvertent use of NIL handles (or pointers). NIL handles can come back from the Resource Manager and the Memory Manager during failed calls. If a program is being sloppy and not checking errors, it's easy to fall into using a NIL handle, and this technique will flush it out. A double dereference of a NIL handle will crash the computer. Something like

```
newArf := aHandle^^.arf;
```

will crash if aHandle is $0 and we've installed this nice bus error number.

This technique will tell when a program inadvertently jumps off to $0 as a place to execute code, which can happen from misaligned stacks or from trying to execute a purged code resource.

By watching for the funny numbers to show up in data structures, you can find out when NIL pointers are being used as the source for data. This is surely not what was meant, and they're easy to find when a distinctive number points them out. These uses won't crash the computer, of course.

CODE SAMPLE

```
theHandle = GetResource('dumb', 1);
aChar = **theHandle;
```

This is easy: the GetResource call may fail. If the 'dumb' resource isn't around, theHandle becomes NIL. Dereferencing theHandle when it's NIL is a bug, since aChar ends up being some wacko number out of ROM in the normal case (ROMBase at $0) and cannot be assumed to be what was desired. This bus error technique will crash the computer right at the **theHandle, pointing out the lack of error checking.

HEAP SCRAMBLE AND PURGE

With this option on, all movable blocks of memory (handles) are moved, and all purgeable blocks are purged, whenever memory *can* be moved or purged—which is different from moving and purging memory whenever it *needs* to be moved or purged. This technique is excellent at forcing those once-a-month crashing bugs to crash more often—like all the time. You should run your entire program with this option on, in combination with the bus error technique, using all program features

TMON, THEN AND NOW

BY WALDEMAR HORWAT

The first version of TMON was released in late 1984. TMON was a summer project for me at TMQ Software when I was a junior in high school. I wrote it because I was dreaming about a one-Macintosh debugger (MacsBug required a terminal at the time) that had a direct-manipulation user interface. Direct manipulation meant more than just having windows—it meant you would be able to change memory or registers simply by typing over your values, assemble instructions by typing in a disassembly window, and so on.

THE ORIGINAL TMON

Memory constraints of the Macintosh 128K forced me to write TMON entirely in assembly language—the original version used only 16K plus a little additional memory to save the screen. TMON used its own windowing system to avoid reentrancy problems with debugging programs that call the system. TMON also included a "User Area," a block of code that could extend TMON. The source code was provided for the standard user areas, and Darin Adler took great advantage of this facility to add numerous features to TMON in his Extended User Area.

Writing TMON took a little ingenuity. I didn't have anything that could debug it, so I wrote the entire program, assembled it, ran it on a Macintosh, and watched it crash. After a couple of dozen builds, I got it to display its menu bar on the screen. By about build 100, I had a usable memory dump window that I could then use to debug the rest of TMON.

TMON PRO

Improving a program written entirely in tight assembly language designed for a Macintosh 128K became intractable, so I switched to MPW C++. Version 3.0 of TMON (TMON Pro) is written half in assembly language and half in C++. Using C++ turned out to be one of the best ways to debug a program: C++ features such as constructors and destructors prevented a lot of pesky programming errors. The downside of using a high-level language is that code size grows explosively—TMON 3.0's code is about ten times larger than TMON 2.8's.

When writing TMON 3.0, I reevaluated earlier design decisions. I opted to continue to concentrate on debugging at the assembly language level for two reasons. First, there are many bugs that can arise on a Macintosh that pure source-level debuggers can't handle. Second, I find that I use TMON at least as much for learning about the Macintosh as I do for debugging.

I sometimes wish I could use the Macintosh windows in TMON. Nevertheless, I decided to remain with TMON's custom windows for reasons of safety. Until the Macintosh has a real reentrant multitasking system that can switch to another task at *any* point in the code, writing such a debugger would either make it prone to crashing if it was entered at the wrong time or require the debugger to be more dependent on undocumented operating system internals than I like.

I found that writing TMON 3.0 was much harder and took much longer than writing the original TMON. Part of this was due to the second-system effect—the product just kept on growing over time. Nevertheless, I also found that writing TMON 3.0 was difficult because of the loss of the Macintosh "standard." There are now over a dozen Macintosh models, using the 68000 through the 68040, some with third-party accelerators, various ROM versions, 24- and 32-bit mode, virtual memory, several versions of the operating system, and numerous INITs, patches, video cards, and other configuration options. These options present unique challenges to a low-level debugger such as TMON, which must include special code for many of them.

Despite the frustration, I think that writing TMON was worth it—it made many developers' lives easier. I plan to continue to evolve TMON in the future and incorporate suggestions for improvements.

and really putting it through its paces. You'll be glad you did. Because this debugger option simulates running under low-memory conditions all the time, it stress-tests the program's memory usage.

AVAILABILITY

All the debuggers have this option, but the one most worth using is in Old TMON and TMON Pro, since it implements both scramble (moving memory) and purge. MacsBug and Jasik's debugger both have scramble, but they're too slow, and neither has a purge option.

ERRORS CAUGHT

This technique will catch improper usage of dereferenced or purgeable handles, mistakes that fall into the "easy to make, hard to discover" category. The technique will also catch blocks that are overwritten accidentally, since there's an implicit heap check each time the heap is scrambled. *Warning:* The bugs you find may not be yours.

CODE SAMPLE

```
aPicture = GetPicture(1);
FailNil(aPicture);
aPtr = NewPtr(500000);
FailNil(aPtr);
aRect = (**aPicture).picFrame;
DrawPicture(aPicture, &aRect);
```

Here, if the picture is purgeable, it might be purged to make room in the heap for the large pointer allocated next. This would make aRect garbage, and DrawPicture wouldn't work as intended, probably drawing nothing. Here's a similar example in Pascal:

```
aPicture := GetPicture(kResNum);
FailNil(aPicture);
WITH aPicture^^ DO
BEGIN
   aPtr := NewPtr(500000);
   FailNil(aPtr);
   aRect := picFrame;
END; {WITH}
```

Here, even if the picture isn't purged, the NewPtr call might move it, invalidating the WITH statement and resulting, again, in a bad aRect.

ZAPPING HANDLES

The idea here is to trash disposed memory at the time it's disposed of in order to catch subsequent use of the free blocks. The technique fills disposed memory with bus error numbers, so that if you attempt to use disposed memory later, the program will crash. A related option is MPW Pascal's -u option, which initializes local and global variables to $7267.

AVAILABILITY

This technique is implemented as a part of Jasik's Discipline option and is also a dcmd, available on the *Developer CD Series* disc, for TMON Pro or MacsBug. You can also just write it into your program by writing bottleneck routines for disposing of memory (such as MyDisposHandle, MyDisposPtr) that fill blocks with bus error numbers just before freeing them. The problem with this is that memory freed by other calls (ReleaseResource, for instance) isn't affected. We recommend the dcmd or Jasik's Discipline.

ERRORS CAUGHT

This technique will catch reusing deallocated memory or disposing of memory in the wrong order. It can also catch uninitialized variables, since after you've been running it for a while, much of the free memory in the heap will be filled with bus error numbers.

CODE SAMPLE

```
SetWRefCon(aWindowPtr, (long)aHandle);
. . .
DisposeWindow(aWindowPtr);
DisposHandle((Handle) GetWRefCon(aWindowPtr));
```

The GetWRefCon will work on a disposed window, but it's definitely a bug. Zapping the handles sets the refCon to a bus error number, forcing the DisposHandle call to fail.

CHECKSUM $0

Once again, we're dealing with the address $0. This technique, however, is sort of the opposite of the first one: it catches writing to $0 rather than reading or executing from it.

AVAILABILITY

This one is easy: you can set up a checksum so that you'll drop into the debugger whenever the value at $0 changes. All the debuggers have a way to do this. Also, EvenBetterBusError sets up a VBL to detect if $0 changes, but since VBL tasks don't run very often (relative to the CPU, anyway), you'll probably be far away in your

code by the time it notices. It's still much better than nothing, though, since knowing the bug exists is the first step toward fixing it.

Note that on the IIci the Memory Manager itself changes $0, so you'll get spurious results. EvenBetterBusError knows about this and ignores it.

ERRORS CAUGHT

The errors caught by this technique are much the same as those caught by the first technique, except that this one catches writes rather than reads. This way, if your code tries to write to address $0 (by dereferencing a NIL handle or pointer), you'll know.

CODE SAMPLE

```
aPtr = NewPtr(kBuffSize);
BlockMove(anotherPtr, aPtr, kBuffSize);
```

This one's pretty obvious: if the NewPtr call fails, aPtr will be NIL, and the BlockMove will stomp all over low memory. If kBuffSize is big enough, this will take you right out, trashing all your low-memory vectors and your debugger, too.

DISCIPLINE

Discipline is a debugger feature that checks for bogus parameters to Toolbox calls. It would of course be nice if the Toolbox itself did more error checking, but for performance reasons it can't. (Be forewarned that some versions of the system have errors that Discipline will catch.) Discipline is the perfect development-time test. It catches all those stupid mistakes you make when typing your code that somehow get past the compiler and may persist for some time before you discover them. It can literally save you hours tracking down foolish parameter bugs that should never have happened in the first place.

AVAILABILITY

Old TMON has an early version of Discipline, but there are no checks for Color QuickDraw calls or later system calls, so its usefulness is limited. There is an INIT version of Discipline (on the *Developer CD Series* disc with MacsBug) that works in conjunction with MacsBug or TMON Pro that's quite usable, if slow and clunky. Jasik's version of Discipline is far and away the best; use it if you can.

ERRORS CAUGHT

As you'd expect, Discipline catches Toolbox calls with bad arguments, like bogus handles, and also sometimes catches bad environment states, like trying to draw into a bad grafPort.

CODE SAMPLE

```
aHandle = GetResource('dumb', 1);
FailNil(aHandle);
. . .
DisposHandle(aHandle);
```

The problem here is that a resource handle has to be thrown away with ReleaseResource, not DisposHandle. Otherwise, the Resource Manager will get confused since the resource map won't be properly updated. Sometime later (maybe much later) Very Bad Things will happen.

32-BIT MEMORY MODE

Running in full 32-bit mode in System 7 forces the Memory Manager and the program counter to use full 32-bit addresses: this is something new on the Macintosh. The old-style (24-bit) Memory Manager used the top byte of handles to store the block attributes (whether or not the handle was locked, purgeable, and so forth). By running your program in 32-bit mode, you'll flush out any code that mucks with the top bits of an address, for any reason, accidentally or on purpose. In the past, many programs examined or modified block attributes directly. This is a bad idea. Use the Toolbox calls HGetState and HSetState to get and set block attributes.

AVAILABILITY

You get 32-bit memory mode with System 7, of course! You use the Memory cdev to turn on 32-bit addressing, available only on machines that have 32-bit-clean ROMs (Macintosh IIfx, IIci, IIsi). You should also install more than 8 MB of RAM and launch your application first, so that it goes into memory that *requires* 32-bit addressing (within the 8 MB area, addresses use only 24 bits). We also recommend using TMON's heap scramble in 32-bit mode, since the block headers are different.

ERRORS CAUGHT

You can inadvertently mess up addresses in a bunch of ways. Obviously, any code that makes assumptions about block structures is suspect. Doing signed math on pointers is another one that comes up pretty often. Any messing with the top bytes of addresses can get you into big trouble, jumping off into weird space, where you have no business.

CODE SAMPLE

```
aHandle = (Handle) ((long) aHandle | 0x80000000);
```

Naturally, this method of locking a handle is not a good idea, since in 32-bit mode the locked bit isn't even there. Use HLock or HSetState; they'll do the right thing.

FULL COMPILER AND LINKER WARNINGS

Always develop your code with full warnings on. When you're compiling and linking your program, any number of errors or warnings will be emitted. The errors are for things that are just plain wrong, so you'll have to fix those immediately. Warnings, however, indicate things that aren't absolutely wrong, but certainly are questionable as far as the compiler or linker is concerned.

We think you should fix every problem as soon as a warning first appears, even if there's "nothing wrong" with the code. If you leave the warnings in, little by little they'll pile up, and pretty soon you'll have pages full of warnings spewing out every time you do a build. You know you won't read through them every time. You'll probably just redirect the warnings to a file you never look at so that your worksheet won't be sullied. Then the one warning that *will* cause a problem will sneak right by you, and much later you'll find out that the totally nasty, hard-to-find bug that you finally corrected was one the compiler warned you about a month ago. To avoid this painful experience, deal with the warnings when they appear, even if they're false alarms.

AVAILABILITY

Use the compiler and linker options that turn on full warnings:

- MPW C++: The "-w2" option turns on the maximum compiler warnings.
- MPW C: Use "-warnings full" ("-w2" does the same thing). In addition, the "-r" option will warn you if you call a function with no definition.
- MPW Linker: The "-msg *keyword*" option controls the linker warnings. *Keyword* is one or more of these: dup, which enables warnings about duplicate symbols; multiple, which enables multiple warnings on undefined references to a label (you can thus find all the undefined references in one link); and warn, which enables warnings.
- THINK C: Because the compile is stopped when a warning is encountered, it forces you to fix all warnings. Some people like this; others don't. We do, but you decide. Be sure that "Check Pointer Types" is turned on in the compiler options.
- Pascal: Most of the things that cause warnings in C are automatically enforced.

If you're coding in C, it's also a good idea to prototype *all* your routines. This avoids silly errors.

ERRORS CAUGHT

The compiler and linker will tell you about lots of things. Some examples are

- the use of uninitialized variables (which is a real bug)
- bad function arguments
- unused variables (these confuse the code and may be real bugs)
- argument mismatches (probably bugs)
- signed math overflow

In C++, overriding operator new without overriding operator delete is probably a bug and unintentional. Even if a warning is caused by something intentional, fix it so that the warning won't appear.

CODE SAMPLE

```
#define kMagicNumber 12345
. . .
short result;
result = kMagicNumber*99;
```

The problem with this code is that the multiplication is overflowing a 16-bit short value. If you have full compiler warnings on, the MPW compiler will let you know this with the following error message:

```
### Warning 276 This assignment may lose some significant bits
```

MEMORY PROTECTION

This is something you've always wanted: a way to get a protected memory model for the Macintosh. With memory protection on, memory accesses outside the application's RAM space would be caught as illegal, giving you the chance to find bad program assumptions and wild references. Only Jasik's debugger has this feature now.

The protected mode is only partly successful, though, since the Macintosh has nothing that resembles a standard operating system. The problems stem from how programs are expected to run, in that references to some low-memory globals are OK, and code and data share the same address space. Given the anarchy in the system, the way Jasik set it up is to allow protection of applications only. The protected mode also protects the CODE resources in the application from being overwritten.

Although this protected mode is not as good as having the OS support protected memory spaces, it's still a giant leap ahead in terms of finding bugs in your programs. By catching these stray references during development, you can be assured that the

user won't get random crashes because of your program. This is an ideal development tool for catching latent bugs that don't often show up. Who knows what a write to a random spot in memory may hit? Sometimes you're just lucky, and those random "stomper" bugs remain benign, but more often they're insidiously nasty.

AVAILABILITY

This tool is currently implemented only in Jasik's debugger. The memory protection is implemented using the MMU, and it slows down the machine by around 20 percent. It's a mixed blessing, since it will crash on any number of spurious errors—use it anyway.

ERRORS CAUGHT

If the application writes to low memory or to the system heap, it's probably not what was desired. A few cases could be deemed necessary, but in general, any references outside the application heap space are considered suspect. Certainly, modifying system variables is not a common task that applications need to support. This memory protection will catch those specific references and give you the chance to be sure that they're valid and necessary.

Writing to I/O space or screen RAM is another problem this technique will catch. Writing directly to the screen is bad form, and only tacky programs (and games, which must do it) stoop this low. Even HyperCard writes directly to the screen; please don't emulate it. Some specialized programs could make an argument for writing to I/O space, since they may have a device they need to use up there. This protection will catch those references and point out a logically superior approach, which is to build a driver to interface to that space, instead of accessing it directly.

CODE SAMPLE

```
*((long*) 0x16A) = aLong;
```

The low-memory global Ticks is being modified. Writing to low-memory globals is a Very Bad Thing to do. This will be caught by memory protection.

LEAKS

A memory leak occurs when a program allocates a block of memory with either NewHandle or NewPtr (or even with Pascal New or C malloc, both of which turn into NewPtr at a lower level), but that block is never disposed of, and the reference to it is lost or written over. If a program does this often enough, it will run out of RAM and probably crash. This leads to the famous statement: "Properly written Macintosh programs will run for hours, even days, without crashing"—a standing joke in Developer Technical Support for so long we've forgotten the original source. Naturally, if the program is leaking in the main event loop, it will crash sooner than if

it leaks from some rare operation. If it leaks at all, it will ultimately fail and crash some poor user.

AVAILABILITY

A simple technique that all debuggers support can tell you whether or not the program is leaking. Do a Heap Total and check the amount of free space and purgeable space that's available. Run the program through its paces and then see if the amount of free space plus purgeable space has dropped. If it has, try again, under the assumption that the program might have loaded some code or other data the first time around. If it's still smaller, it's likely to be a leak. This approach, of course, only shows that you *have* a leak; tracking it down is the hard part. But, hey, you can't start tracking till you know it's there.

There's a dcmd called Leaks (on the *Developer CD Series* disc) that runs under both TMON Pro and MacsBug. The basic premise is to watch all the memory allocations to see if they get disposed of correctly. Leaks patches the traps NewHandle, DisposHandle, NewPtr, and DisposPtr. When a new handle or pointer is allocated on the heap, Leaks saves the address into an internal buffer. When the corresponding DisposHandle or DisposPtr comes by, Leaks looks it up in the list and, if it finds the same address, dumps that record as having been properly disposed of. Now all those records on the Leaks list that didn't have the corresponding dispose are candidate memory leaks.

The Macintosh has a lot of fairly dynamic data, so Leaks often ends up getting a number of things on its list that haven't been disposed of but are not actually leaks. They're just first-time data, or loaded resources. To avoid false alarms, the Leaks dcmd requires that you perform the operation under question three times, in order to get three or more items in its list that are similar in size and allocated from the same place in the program. An operation can be as simple or complex as desired, since every memory allocation is watched. An example of an operation to watch is to choose New from a menu and then choose Close, under the assumption that those are complementary functions. If you do this three times in a row with Leaks turned on, anything that Leaks coughs out will very likely be a memory leak for that operation.

The dcmd saves a shortened stack crawl of where the memory is being allocated, so that potential leaks can be found back in the source code.

One problem with Leaks as a dcmd is that if it's installed as part of the TMON Pro startup, it patches the traps using a tail patch. Tail patches are bad, since they disable bug fixes the system may have installed on those traps. This could cause a bug to show up in your program that isn't there in an unpatched system. It's still probably worth the risk, given the functionality Leaks can provide. The problem doesn't exist with MacsBug, since the traps are patched by the dcmd before the system patches them.

A vastly superior way around this problem is to provide the Leaks functionality as debugging code, instead of relying on an external tool. By writing an intermediate routine that acts as a "wrapper" around any memory allocations your program does, you can watch all the handles and pointers go by, do your own list management to know when the list should be empty, and dump out the information when it isn't. By wrapping those allocations, you avoid patching traps (always a good idea). Be sure to watch for secondary allocations, such as GetResource/DetachResource pairs. You may still want to run Leaks when you notice memory being lost, but your wrappers don't notice it.

ERRORS CAUGHT

Potential memory leaks, but you knew that already.

CODE SAMPLE

```
anIcon := GetCIcon(kIconId);
PlotCIcon(aRect, anIcon);
DisposHandle(Handle (anIcon));
```

This orphans any number of handles, because the GetCIcon call will create several extra handles for pixMaps and color tables. This is an easy error to make, since the GetCIcon returns a CIconHandle, which seems a lot like a PicHandle. A PicHandle is a single handle, though, and a CIconHandle is a number of pieces. Always use the corresponding dispose call for a given data structure. In this case, the appropriate call is DisposCIcon.

STRESS ERROR HANDLING

Here the goal is to see how the program deals with less than perfect situations. Your program won't always have enough RAM or disk space to run smoothly, and it's best to plan for it. The first step is to write the code defensively, so that any potential error conditions are caught and handled in the code. If you don't put in the error-handling code, you're writing software that never expects to be stressed, which is an unreasonable assumption on the Macintosh.

AVAILABILITY

Try running the program in a memory-critical mode, where it doesn't have enough RAM even to start up. Users can get into this unfortunate situation by changing the application's partition size. Rather than crash, put up an alert to tell users what went wrong, and then bail out gracefully. Try running with just enough RAM to start up, but not enough to open documents. Be sure the program doesn't crash and does give the user some feedback. Try running in situations where there isn't enough RAM to edit a document, and make sure it handles them. What happens if you get a memory-low message, and you try to save? If you can't save, the user will be annoyed. What happens when you try to print?

Run your program on a locked disk, and try to save files on the locked disk. The errors you get back should be handled in a nice way, giving the user some feedback. This will often find assumptions in the code, like, "I'm sure it will always be run from a hard disk."

To see if you handle disk-full errors in a nice way, be sure to try a disk that has varying amounts of free space left. Here again, if you've only ever tested on a big, old, empty hard disk, it may shock you to find out that your users are running on a double-floppy-disk Macintosh SE and aren't too happy that disk-full errors crash the program. A particularly annoying common error is saving over a file on the disk. Some programs will delete the old file first and then try to save. If a disk-full error occurs, the old copy of the data has been deleted, leaving the user in a precarious state. Don't force a user to switch disks, but allow the opportunity.

Especially with the advent of System 7, you should see how your program handles the volume permissions of AppleShare. Since any Macintosh can now be an AppleShare server, you can definitely expect to see permission errors added to the list of possible disk errors. Try saving files into folders you don't have permission to access, and see if the program handles the error properly.

ERRORS CAUGHT

Inappropriate error handling, unnecessary crashes, lack of robustness, and general unfriendliness.

CODE SAMPLE

```
i := 0;
REPEAT
   i := i + 1;
   WITH pb DO
   BEGIN
      ioNamePtr := NIL;
      ioVRefnum := 0;
      ioDirID := 0;
      ioFDirIndex := i;
   END;
   err := PBGetCatInfo (@pb, False);
UNTIL err <> noErr;
```

This sample is trying to enumerate all files and directories inside a particular directory by calling PBGetCatInfo until it gets an error. (Note that this sample does one very important thing: initializing the ioNamePtr field to NIL to keep it from returning a string at some random place in memory.) The problem with this loop is that it assumes that any error it finds is the loop termination case. For an AppleShare

volume, you may get something as simple as a permission error for a directory you don't have access to. This is probably not the end of the entire search, but the code will bail out. This bug would be found by trying the program with an AppleShare volume. The appropriate end case would be to look for the exact error of fnfErr instead or, better, to add the permErr to the conditional.

MULTIPLE CONFIGURATION TESTS

This technique goes beyond merely finding the crash-and-burn bugs to help ensure that the program will run in situations that weren't originally expected. Just fixing crash-and-burn bugs is for amateurs. Professional software developers want their programs to be as bug-free as possible. As a step toward this higher level of quality, testing in multiple configurations can give you more confidence that you haven't made faulty assumptions about the system. The idea is to try the program on a number of machines in different configurations, looking for combinations that cause unexpected results.

AVAILABILITY

Multiple configuration tests should use the Macintosh Plus as the low-end machine to be sure that the program runs on 68000-based machines and on ones that have a lot of trap patches. Some of the code the system supports is not available, like Color QuickDraw. If you use anything like that, you will crash with an unimplemented trap number error, ID=12. The Macintosh Plus is a good target for performance testing as well, since it's the slowest machine you might expect to run on. Its small screen can also point out problems that your users might see in the user interface. For example, some programs use up so much menu bar space that they run off the edge of the screen. That might not be noticed until you run the program on a machine with a small screen. If your program specifically doesn't support low-end machines, you should still put in a test for them and warn the user. Crashing on a low-end machine is unacceptable, especially when all you needed was a simple check.

Naturally, the multiple configurations include a Macintosh II-class machine to be sure that assumptions about memory are caught. Because most development is done on Macintosh II computers, this case will likely be handled as part of the initial testing. It's virtually certain that your program will be used on a Macintosh II by some users.

Using multiple monitors on a single system can point out some window- or screen-related assumptions. The current version of the old 512 x 342 fixed-size bug is the assumption that the MainGDevice is the only monitor in the system. Testing with multiple monitors will point out that although sometimes the main device is black and white, there's a color device in the system. Should your users have to change the main device and reboot just to run your program in color?

By testing the program within a color environment, even if it doesn't use color, you'll find any assumptions about how color might be used or the way bitmaps look. It's a rare (albeit lame) program that gets to choose the exact Macintosh it should run on.

Try the program under Virtual Memory to see if there are built-in assumptions regarding memory.

Use the program under both System 6 and 7. If the program requires System 7, but a user runs it under System 6, it should put up an alert and definitely not crash. For the short term, it's obvious that you cannot assume all users will have either one system or the other. The number of fundamental differences between the systems is sufficiently large that the only way to gain confidence that the program will behave properly is to run it under both systems. Some bugs that were never caught under System 6 may now show up under System 7. The bugs may even be in your code, with implicit assumptions about how some Toolbox call works.

Doing a set of functionality tests on these various types of systems will ensure that you can handle the most common variations of a Macintosh. Tests of this form will give you a better feeling for the limits of your program and the situations it can handle gracefully. There's usually no drawback to getting a user's-eye view of your program.

There is a tool called Virtual User (APDA #M0987LL/B) that can help a lot with these kinds of tests. It allows you to script user interactions so that they can be replayed over and over, and it can execute scripts on other machines remotely, over AppleTalk. So, for instance, you could write a script that puts your program through its paces, and then automatically execute that script simultaneously on lots of differently configured Macintosh systems.

ERRORS CAUGHT

As discussed above, this technique attempts to flush out any assumptions your code makes about the environment it's running in: color capabilities, screen size, speed, system software version, and so on.

CODE SAMPLE

```
void Hoohah(void)
{
   long localArray[2500];

   . . .
}
```

Naturally, this little array is stack hungry and will consume 10K of stack. On a Macintosh II machine, this is OK, as the default stack is 24K. On the Macintosh Plus,

THIRD-PARTY COMPATIBILITY TEST LAB

Apple maintains a Third-Party Compatibility Test Lab for the use of Apple Associates and Partners. The Lab features many preconfigured domestic and international systems, extensive networking capabilities, support from staff engineers, and so on. If you're an Apple Associate or Partner, and you'd like to make a test-session appointment or get more information, contact Carol Lockwood at (408)974-5065 or AppleLink LOCKWOOD1. Or you can write to Apple Third-Party Test Lab, Apple Computer, Inc., 20525 Mariani Avenue M/S 35-BD, Cupertino, CA 95014.•

the stack is only 8K, so when you write into this array you will be writing over the heap, most likely causing a problem. This type of easy-to-code bug may not be caught until testing on a different machine. Merely because the code doesn't crash on your machine doesn't mean it's correct.

ASSERTS

Asserts are added debugging code that you put in to alert you whenever a situation is false or wrong. They're used to flag unexpected or "can't happen" situations that your code could run into. Asserts are used only during development and testing; they'll be compiled out of the final code to avoid a speed hit.

AVAILABILITY

You could write a function called ASSERT that takes a result code and drops into the debugger if the result is false—or, better yet, writes text to a debugging window. In MPW, you can use __FILE__ and __LINE__ directives to keep track of the location in the source code. Another thing to check for is bogus parameters to calls, sort of like Discipline. Basically, you want to check any old thing that will help you ensure consistency and accuracy in your code, the more the merrier, as long as the asserts don't "fire" all the time. Fix the bugs pointed out by an assert, or toughen up the assert, but don't turn it off. If you just can't stand writing code to check every possible error, temporarily put in asserts for the ones that will "never" happen. If an assert goes off, you'd better add some error-handling code.

The following sample code shows one way to implement ASSERT.

```
#if DEBUG
#define ASSERT(what)  do { if(!(what)) dbgAssert(__FILE__,__LINE__); } while(0)
#else
#define ASSERT(what)  ((void)0)
#endif

void dbgAssert(const char* filename, int line)
{
   char msg[256];

   sprintf(msg, "Assertion failed # %s: %d", filename, line);
   debugstr((Str255)msg);
}
```

In this example, ASSERT is defined by a C macro. If DEBUG is true, the macro expands to a block of code that checks the argument passed to ASSERT. If the argument is false, the macro calls the function dbgAssert, passing it the filename and line number on which the ASSERT occurs. If DEBUG is false, the macro ASSERT

expands to nothing. Making the definition of ASSERT dependent on a DEBUG flag simplifies the task of compiling ASSERTs out of final code.

ERRORS CAUGHT

This technique catches all sorts of errors, depending, of course, on how you implement it. Logic errors, unanticipated end cases that show up in actual use, and situations that the code is not expecting are some of the possibilities.

CODE SAMPLE

```
numResources = Count1Resources('PICT');
for(i=1; i<=numResources; i++) {
   theResource = Get1IndResource('PICT', i);
   ASSERT(theResource != nil);
   RmveResource(theResource);
}
```

The problem here is that the code doesn't account for the fact that Get1IndResource always starts at the beginning of the available resources. So the first time through, we get the resource with index 1, and we remove it. The next time through, we ask for resource 2, but since we removed the resource at the front of the list, we get what used to be resource 3; we've skipped one. The upshot is that only half the resources are removed, and then Get1IndResource fails. This is a great example of a "never fail" situation failing. The ASSERT will catch this one nicely; otherwise, you might not know about it for a long time. The solution is to always ask for the first resource.

TRACE

Trace is a compiler option that causes a subroutine call to be inserted at the beginning and end of each of your functions. You have to implement the two routines (%__BP and %__EP), and then the compiler inserts a JSR %__BP just after the LINK instruction and a JSR %__EP just before UNLK. This gives you a hook into every procedure that's compiled, which can be extremely useful. Like asserts, trace is debugging code and will be compiled out of the final version.

AVAILABILITY

Trace is available in all the MPW compilers and in THINK Pascal. THINK C's profiler can be configured and used in the same sort of way.

ERRORS CAUGHT

By being able to watch every call in your program as it's made, you can more easily spot inefficiencies in your segmentation and your call chain: If two often-called routines live in different segments, under low-memory situations you may be swapping code to disk constantly. If you're redrawing your window 12 times during

RELATED READING

Debugging Macintosh Software with MacsBug by Konstantin Othmer and Jim Straus (Addison-Wesley, 1991) and *How to Write Macintosh Software* by Scott Knaster (Hayden Books, 1988).•

A WORD TO THE WISE FROM FRED

What we've described in this article are a number of tools for doing Macintosh software development. Some of you are about to say, "Oh, those sound really great, but I don't have time to use them—I'm about to ship," or whatever. I'd like to tell you a story that a man of sound advice, Jim Reekes, told me: A young boy walked into a room and saw a man pushing a nail into the wall with his finger. The boy asked him, "Hey, mister, why don't you go next door and get a hammer?" The man replied, "I don't have time." So the boy went next door, got a hammer, and came back. The man was still pushing the nail into the wall with his finger. So the boy hit the man in the head with the hammer, killed him, and took the nail.

an update event, you could probably snug things up a little and gain some performance. You can watch the stack depth change, monitor memory usage and free space, and so on. Think up specific flow-of-control questions to ask and then tailor your routines to answer them. Expect to generate far more data than you can look at. Really get to know your program. Go wild.

CODE SAMPLE

```
PROCEDURE HooHah
VAR
   localArray: ARRAY[1..2500] OF LongInt;
BEGIN
   . . .
END; {HooHah}
```

Once again, we're building a stack that's too big for a Macintosh Plus. The stack sniffer will catch it eventually, but since VBL tasks don't run very often, you may be far away by then. Trace could watch for it at each JSR and catch it immediately.

USEFUL COMBINATIONS

All these techniques are powerful by themselves, but they're even better when used in combination. Use them as early and as often as you can. Some of them are a bit of trouble, but that smidgen of extra work is paid back many times over in the time saved by not having to track down the stupid bugs. Use them throughout development, right up to the end. Many bugs show up through interactions that only begin near the end of the process. Diligent use of these techniques is guaranteed to find many of the easy bugs, so you can spend your time finding the hard ones, which is much more interesting and worthwhile.

OK, now armed to the teeth with useful techniques, you're ready to stomp bugs. You know what to look for and how to flush them out. But you know what? Debugging is *still* hard.

THANKS TO OUR TECHNICAL REVIEWERS
Jim Friedlander, Pete Helme, Jim Reekes•

MACINTOSH HYBRID APPLICATIONS FOR A/UX

Apple's A/UX operating system is unique among UNIX systems in that it merges the Macintosh user interface and application environment with the multitasking UNIX operating system. Developers can take advantage of this combination by creating a class of applications called hybrids. This article describes the techniques necessary to create Macintosh hybrid applications and demonstrates some of the benefits of these applications.

JOHN MORLEY

The UNIX® operating system began some 20 years ago as a personal project undertaken by a couple of engineers at AT&T Bell Laboratories. For a number of technical and business reasons, UNIX emerged as the leading software platform for a phenomenon called Open Systems. Although this buzzword is batted around in many different and confusing contexts, it basically refers to systems that adhere to multivendor industry standards, thus protecting their owner's investment in software, training, and so on.

At Apple we recognized the growing importance of the UNIX system in many segments of the marketplace, particularly for government, higher education, and large corporate customers. We also understood that the UNIX system's principal weakness was its lack of ease of use at both the system and application level. By grafting the Macintosh user interface onto a full-featured UNIX operating system, and supporting the bulk of popular Macintosh application software as well, we hoped to meet the requirements of the Open Systems marketplace and retain all the joys of working on a Macintosh.

Release 2.0 of the A/UX operating system was the realization of this effort. When using a Macintosh running A/UX, you can treat it purely as a Macintosh or dive into whatever level of sophistication with the UNIX system your expertise and/or bravado allow.

JOHN MORLEY is the manager for development tools in Apple's A/UX engineering group. John has been hacking code for so long (20 years) that he remembers the "good old days" when the mark of a good programmer was being able to sight-read punched paper tape. When asked about the future of software engineering, he is quick to praise the virtues of object-oriented programming and his work on the design of a new language called "Add 1 to COBOL giving Object COBOL." In his spare time John loves to ride his Harley-Davidson motorcycle, plan trips to the Hawaiian island of Kauai, and visit with the fish underwater. John's dream is to work at an Apple engineering facility on Kauai so that the blurry line dividing work and play will finally be dissolved altogether.•

For the developer, A/UX opens up some new possibilities due to the presence of both the UNIX system and Macintosh programming paradigms. Macintosh developers can use A/UX as a gateway from their Macintosh application into the world of UNIX system services. UNIX system developers can use A/UX to deliver UNIX system applications that incorporate the benefits of the Macintosh user interface.

HYBRID APPLICATIONS

As the name implies, a hybrid application combines two distinct programming models within a single application program. In the case of the A/UX operating system, the two available programming models are the Macintosh Toolbox interface and the UNIX system call interface.

In addition to the two programming models present in A/UX, there are two distinct executable file formats: the UNIX executable file format known as Common Object File Format (COFF) System V.2, and the Macintosh executable file format known as Object Module Format (OMF).

The term *Macintosh hybrid application* refers to an application that's represented in Macintosh OMF, primarily uses the Macintosh Toolbox interface, but also accesses the A/UX operating system via the UNIX system call interface.

Alternatively, the term *UNIX hybrid application* refers to an application that's represented in COFF, primarily uses the UNIX system call interface, but also accesses the functions provided by the A/UX Macintosh Toolbox.

A/UX MACINTOSH TOOLBOX

The Macintosh Toolbox as it is supported under A/UX is documented in *Inside Macintosh* Volumes I-V and in *A/UX Toolbox: Macintosh ROM Interface*. The interface mechanism that's used to access the Macintosh Toolbox is the set of A-line trap instructions reserved for this purpose in the Motorola 680x0 architecture. The high-level languages supporting Macintosh programming contain features that allow the programmer to use traditional procedure call notation to access the Macintosh Toolbox. The compiler then translates those procedure calls into the actual A-line trap instructions to access the Toolbox.

A/UX SYSTEM CALLS

The UNIX system call interface is documented in the *A/UX Programmer's Reference*, Section 2. The interface mechanism that's used to access the UNIX system calls is a CPU trap instruction that causes a context switch between the application program, which runs in user mode, and the UNIX system kernel, which runs in supervisor mode. The A/UX C runtime library contains procedures to access each of the UNIX system calls supported by A/UX.

Macintosh applications running on A/UX may also access the UNIX system calls. An MPW library (libaux_sys.o) that contains procedures for each UNIX system call, analogous to the ones in the A/UX C runtime library, is included on the *Developer CD Series* disc for this issue. By calling routines from this library a Macintosh application becomes a Macintosh hybrid application with access to the capabilities provided by the UNIX system.

WHY CREATE MACINTOSH HYBRID APPLICATIONS?

There are several reasons why you might want to create a Macintosh hybrid application. Here are some examples:

- to create a Macintosh style front-end interface for an existing character-based UNIX system application
- to access UNIX system networking from a Macintosh application
- to execute UNIX system applications and utilities from a Macintosh application

The class of applications that act as front ends to existing UNIX system programs is of particular interest. The UNIX operating system has been around for two decades and a large body of software exists that can be ported easily from one UNIX system to another. The problem with these applications is that they were designed to work with character-oriented display devices.

Most people who are familiar with the Macintosh user interface are reluctant to sacrifice the ease of use that applications designed for the Macintosh provide. One way to "dress up" these older UNIX system applications is to provide a Macintosh-style user interface via an application that acts as a front end to the existing character-based application. While not as elegant a solution as redesigning the application with the new user interface in mind, the front-end approach can usually be implemented in less time and at less expense.

MULTITASKING AND THE MACINTOSH

A developer creating a Macintosh hybrid application needs some understanding of Macintosh multitasking and how it's implemented by A/UX. If not properly designed, a Macintosh hybrid application can easily cause the Macintosh Toolbox environment within A/UX to become deadlocked. Following the guidelines given here can keep the number of catastrophic failures during development to a minimum.

The Macintosh was designed to be a personal computer. This resulted in emphasis on the interaction between a single user and the computer while performing a single task. With the advent of MultiFinder the Macintosh became capable of switching between two or more active applications as well as performing some limited processing in the background while the user interacts with any application.

The MPW library libaux_sys.o is also on the *A/UX Developer's Tools* set of CD-ROM discs (APDA #B0596LL/A).•

In Macintosh System 7, MultiFinder is integrated into the system as the Process Manager. This article refers to MultiFinder, since A/UX is currently based on Macintosh System 6 software.•

To avoid major incompatibilities with the existing base of application software, MultiFinder was cleverly designed to implement multitasking on top of the existing Macintosh programming model. This style of multitasking is called *cooperative multitasking*. The name conveys the requirement that applications must provide the system with a cue indicating when it's reasonable to interrupt them.

The UNIX operating system, on the other hand, was designed to control minicomputers that normally support many users at once. These computers require the operating system to preemptively schedule tasks for execution using a well-defined scheduling algorithm. A/UX fully implements this style of *preemptive multitasking* for all UNIX processes.

To implement the MultiFinder method of cooperative multitasking within the preemptive multitasking model of the UNIX system, a special thread of control is defined for all processes that access the A/UX Macintosh Toolbox. The A/UX kernel associates one and only one process at a time with the *token of control* for the Macintosh Toolbox. The token of control is passed in the same way that applications are activated under MultiFinder.

THE PERILS OF MULTITASKING

An unsuspecting programmer creating a Macintosh hybrid application can easily be tripped up by lack of knowledge about the multitasking environment. Consider the following program:

```
#include <StdIO.h>
main()
{
   char buf[100];
   int len;

   write(1,"Type Something\n",15);
   len = read(0,buf,100);
   write(1,"You Typed: ",11);
   write(1,buf,len);
   write(1,"\n",1);
}
```

This rather primitive piece of code can be compiled with MPW C and linked to produce an MPW tool. When run, it writes a prompt to the active MPW window and waits for keyboard input terminated by the Enter key. The program then echoes the input to the window and terminates. During the time that the program is waiting for keyboard input, you can switch MultiFinder layers by clicking in a different application window or choosing from the Apple menu or MultiFinder application icon in the menu bar.

This same program can be compiled and linked with the A/UX C compiler (cc or c89) to produce a native COFF application. When run within a CommandShell window it exhibits the same behavior as when compiled with MPW C, including the ability to switch MultiFinder layers while waiting for input from the keyboard.

The program can be modified so that when compiled with MPW it becomes a Macintosh hybrid application. (See "Compiling and Linking Macintosh Hybrid Applications" for some useful tips.) This is done by substituting calls to the A/UX system call routines in place of the standard MPW C runtime routines, as follows:

```
#include <StdIO.h>
#include <LibAUX.h>
#include </:usr:include:fcntl.h>
main()
{
   char buf[100];
   int len, fd;

   fd = auxopen("/dev/ttyC1",O_RDWR);
   (void) auxwrite(fd,"Type Something\r",15);
   len = auxread(fd,buf,100);
   (void) auxwrite(fd,"You Typed: ",11);
   (void) auxwrite(fd,buf,len);
   (void) auxwrite(fd,"\r",1);
   (void) auxclose(fd);
}
```

The program now opens the device associated with the window CommandShell 1 and performs the I/O to that window. However, this program contains a serious flaw—the call to auxread will result in a deadlock situation, because as yet there is no data available to be read from the file descriptor associated with the CommandShell window, and the A/UX read system call blocks when data is not available. As a result, the entire MultiFinder environment running on A/UX is suspended. This makes it impossible to switch to CommandShell 1 and enter data via the keyboard.

In this example it's not too difficult to solve the problem of blocking. The A/UX system call interface allows you to perform I/O that's nonblocking, otherwise referred to as *asynchronous I/O*. You can use the A/UX system call fcntl to change the blocking status of a UNIX system file descriptor. Here's how to modify the previous example so that the deadlock situation is avoided:

```
#include <StdIO.h>
#include <CursorCtl.h>
#include <LibAUX.h>
#include </:usr:include:fcntl.h>
```


Blocking is the suspension of a process pending completion of some external event—for example, data becoming available.•

COMPILING AND LINKING MACINTOSH HYBRID APPLICATIONS

There are a few things you should know in order to compile and link a Macintosh hybrid application:

- Use the include file LibAUX.h to define the system calls and their prototypes (especially if you're using C++). For example:

  ```
  #include <LibAUX.h>
  ```

- You may need to include A/UX system header files for some of the system calls. These must be specified using the complete pathname in the Macintosh file system format. You must include LibAUX.h before including any A/UX system header files. For example:

  ```
  #include <LibAUX.h>
  #include </:usr:include:fcntl.h>
  ```

  For information on when to include header files refer to *A/UX Programmer's Reference,* Section 2, and *A/UX Development Tools,* Chapter 2.

- The meanings of the special characters \n and \r are reversed between MPW C and A/UX C. In general, use \r within strings that are passed to A/UX system calls.

- It's a good idea to have the application test to see that A/UX—not the Macintosh operating system—is running. To do this, call function AUXisRunning, which returns a nonzero value if A/UX is running or a zero value if it's not.

- You must link with the library libaux_sys.o to access the system call routines. The default filename for this library is

  ```
  {MPW}Libraries:AUX System Calls: libaux_sys.o
  ```

  This library name should be included in addition to any other library names and options you usually include with your Link command.

```
main()
{
   char buf[100];
   int len, fd, flags;

   /* Open the device associated with CommandShell 1's window.*/
   fd = auxopen("/dev/ttyC1",O_RDWR);
   /* Get the current flags for this file descriptor.*/
   flags = auxfcntl(fd,F_GETFL,0);
   /* Add the O_NDELAY flag to the flags that are already set.*/
   (void) auxfcntl(fd,F_SETFL,flags | O_NDELAY);
   (void)auxwrite(fd,"Type Something\r",15);
   while ( (len = auxread(fd,buf,100)) < 1)
      SpinCursor(1);
   (void)auxwrite(fd,"You Typed: ",11);
   (void)auxwrite(fd,buf,len);
   (void)auxwrite(fd,"\r",1);
   /* Reset the flags to their original state.*/
   (void)auxfcntl(fd,F_SETFL,flags);
   (void)auxclose(fd);
}
```

The A/UX system call fcntl is used first to get the file descriptor flags associated with the CommandShell window and then to set the O_NDELAY bit in the flags word. The O_NDELAY bit determines whether reads from the file descriptor will block if data is not available. With this bit set, when data is not available the value returned by the read system call is 0. The call to the MPW library routine SpinCursor creates idle time for the layer switch to occur. The last call to auxfcntl resets the flags to their original state.

If you want to try this example hybrid application, open a CommandShell window under A/UX, type "sleep 1000" in the window, and then run the example from the MPW shell.

HELPFUL UTILITY FUNCTIONS

For many types of potential Macintosh hybrid applications, particularly the front-end variety, the only UNIX system functionality necessary is the ability to execute a UNIX process and communicate with it. Toward this end I've included in the system call library several utility routines that are at a higher level than the basic system calls. A description of these utility routines follows.

AUXFORK_PIPE

The auxfork_pipe function executes several system calls to create a new UNIX process. In UNIX system parlance, the new process is the child and the process that created it is the parent. The definition for this function is

```
Handle auxfork_pipe(int toparent, int tochild, void (*childtask)(),
                    void *childarg);
```

The parameters toparent and tochild are flags that indicate whether or not to establish a communication pipe in either direction between the parent and child processes. A zero value signifies that no pipe should be created and a nonzero value signifies that a pipe should be created.

The parameter childtask is a function pointer used to identify a function to be called by the child process when the child process is first created. The parameter childarg is a generic pointer passed to the function pointed to by childtask so that you can vary the behavior of that function. Typically, the function pointed to by childtask executes one of the variants of the exec system call and uses the childarg pointer to identify the filename of the program to be executed.

The value returned by this function is either a handle to a structure that holds some global information about the child process or a null pointer if the call was unsuccessful. The definition for this structure is as follows:

For details on the read system call see *A/UX Programmer's Reference,* Section 2.•

```
struct childinfo {
   /* file descriptor for parent->child communication pipe */
   int tochild;
   /* file descriptor for child->parent communication pipe */
   int toparent;
   /* process ID of the child process */
   int pid;
};
```

The file descriptor for the toparent communication pipe has the O_NDELAY bit set in its flags word so that reading from this file descriptor won't cause a block when data is not available.

AUXCLEANUP_FORK_PIPE

An additional utility function is used to clean up after the child process terminates—the auxcleanup_fork_pipe function. Its definition is

```
int auxcleanup_fork_pipe(Handle globals);
```

It takes one parameter, which is the handle returned previously by the auxfork_pipe function. You must be sure that the child process has terminated or is about to terminate before calling auxcleanup_fork_pipe. If you're not sure that the child process will terminate, you can call auxkill to send the child process a termination signal.

AUXFGETS

The auxfgets function uses the read system call to read a string of characters from an open UNIX system file descriptor. The definition for this function is

```
char *auxfgets(char *buf, int count, int file, int timeout);
```

The buf parameter is a pointer to space in which to store the characters read. The count parameter specifies the maximum number of characters to read. The file parameter is the UNIX system file descriptor from which to read. (The auxfgets function described here differs from the standard fgets function in that it uses a file descriptor rather than a stream pointer.) The timeout parameter indicates the maximum number of times to retry the read system call when data is not available.

This function reads characters until one of the following conditions occurs:

- A newline character (0x10) is read.
- The number of characters specified in the count is reached (reserving room for a null character to mark the end of the string).

- The retry count specified in the timeout parameter is reached without any new data being available to read. If the value of timeout is 0, auxfgets retries indefinitely.

The newline character, if any, is stored at the end of the character string. In any case, a null character is appended after the last character stored to mark the end of the string.

AUXSYSTEM

The auxsystem function works much like the UNIX library routine named *system*. To use this function in a Macintosh hybrid application, the application must either be linked as an MPW tool or linked with the Simple Input/Output Window (SIOW) package, which implements standard I/O streams for Macintosh applications. The definition for this function is

```
int auxsystem(char *command);
```

The parameter is a pointer to a character string that contains a valid UNIX system shell command (Bourne shell syntax). This function executes the given command and redirects any output that the command produces on the standard output or standard error streams to the SIOW standard output and standard error streams.

The MPW tool Unixcmd included on the *Developer CD Series* disc is an example of a tool that uses the auxsystem function. This tool executes the UNIX command given on the command line for Unixcmd. The standard output and standard error streams produced by the UNIX command are sent to the MPW window from which Unixcmd was executed, or they can be individually redirected using the MPW shell's redirection syntax. The Unixcmd tool also sets the MPW variable {Status} to the exit status of the UNIX command, so that MPW scripts can test the exit status.

THE UNIX MAIL READER

To demonstrate some of the techniques used to create Macintosh hybrid applications, we'll look at an example front-end application for the Berkeley UNIX system mail reading program, mailx. The application is implemented using a HyperCard stack with HyperTalk® scripts that access HyperCard XFCNs.

The interface for the mail reader consists of two cards in a HyperCard stack. The first card is called *headers* and is used to display a list of header lines identifying the available mail messages. The second card is called *message* and is used to display the content of a selected message.

The UNIX Mail Reader example is provided solely to illustrate the technical issues involved in creating an A/UX Macintosh hybrid application. It is *not* an example of good user interface design, since it was written by a UNIX hacker (yours truly) with

little knowledge of Macintosh user interface guidelines (a little knowledge is a dangerous thing).

HYPERCARD XFCNS

A HyperCard XFCN is a code segment that the HyperCard application calls to perform a function that can't be accomplished with the standard HyperCard commands. By providing HyperCard with XFCNs to access UNIX system calls, you can create Macintosh hybrid applications that are implemented by HyperTalk scripts. Five different XFCNs are used by the UNIX Mail Reader to create a HyperCard front end to the UNIX mail reading program. The XFCNs used are as follows:

- forkpipexfcn, which calls the auxfork_pipe utility function
- fgetsxfcn, which calls the auxfgets utility function
- fgetfxfcn, which makes multiple calls to the auxfgets utility function
- writexfcn, which calls the auxwrite utility function
- cleanupxfcn, which calls the auxcleanup_fork_pipe utility function

The source code for these XFCNs is included on the *Developer CD Series* disc to serve as a model for other XFCNs you may create.

THE ENTRY SCRIPT

When the UNIX Mail Reader stack is opened, the HyperTalk script associated with the headers card is executed.

```
on openStack
   global global_handle, linecount
   put empty into cd field one
   put empty into global_handle
   put forkpipexfcn("/usr/bin/mailx") into global_handle
   - - A real application would give an error message here.
   if global_handle is empty then go to home
   - - Call fgetsxfcn to read first line of mailx output.
   put fgetsxfcn(global_handle,2500) into buf
   - - Check if any mail is available.
   if word 1 of buf is "No" then
      - - Inform user there's no mail.
      put cleanupxfcn(global_handle) into status
      put empty into global_handle
      Beep 1
      answer "Sorry, no mail"
      go to home
```


For more information about HyperTalk and XCMDs refer to the *Apple HyperCard Script Language Guide: The HyperTalk Language.*•

```
   else
      - - Inform user there's mail.
      play "mail.sound"
      - - Discard 2nd line of mailx output.
      put fgetsxfcn(global_handle,2500) into buf
      - - Read available mail headers.
      repeat with linecount = 1 to 9999
         put fgetsxfcn(global_handle,250) into buf
         - - Check if done.
         if length(buf) = 0 then exit repeat
         put buf into line linecount of cd field one
      end repeat
      - - Calculate number of messages.
      subtract 1 from linecount
   end if
end openStack
```

After some initialization, the XFCN forkpipexfcn is called to start execution of the Berkeley UNIX system mail reader located in the file /usr/ucb/mailx. Then, the XFCN fgetsxfcn is called to read the first line of output from the mailx program into the HyperTalk variable buf.

Notice that the second parameter to fgetsxfcn is the value 2500. This parameter is the timeout count described in the definition of auxfgets. The value 2500 was derived by observing the longest time it normally takes for the mail program to begin execution and produce the first line of output.

The script tests the string that was just read to see if it's the special message that indicates no mail messages. If it is, the script notifies the user and exits.

If the first message is other than the no-mail message, the script reads the header for each message and places it sequentially in the message headers field on the headers card. The message headers begin with the third line of output from the mailx program, so the script reads and discards the second line of output from the mailx program. After the final message is read, the global HyperTalk variable linecount is set to the total number of messages. The final message header is identified by checking to see if the last fgetsxfcn call returned no data, indicating a timeout.

Figure 1 shows the headers card of the UNIX Mail Reader example.

THE EXIT SCRIPT

When the user clicks the Home button, the UNIX Mail Reader stack exits and the following script associated with the headers card executes:

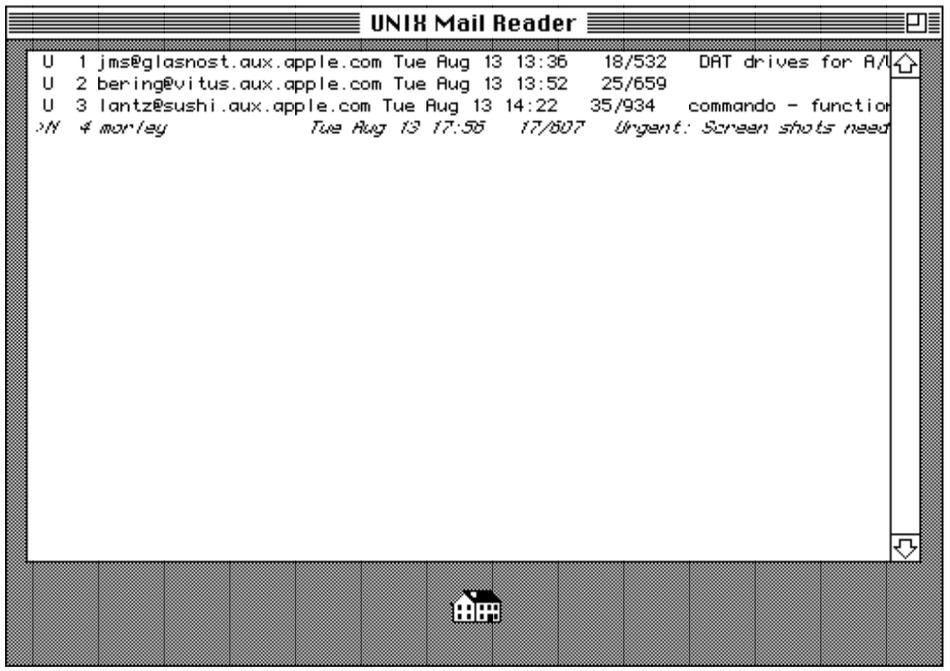


Figure 1
Headers Card With Sample Mail Headers

```
on closeStack
   global global_handle
   if global_handle is not empty then
      answer "Update Message Queue?" with "Yes" or "No"
      if It is "No" then
         put writexfcn(global_handle,("x" & lineFeed)) into writecount
      else
         play "empty trash (flush)"
         put writexfcn(global_handle,("q" & lineFeed)) into writecount
      end if
      put cleanupxfcn(global_handle) into status
      put empty into global_handle
   end if
   put empty into cd field one
   play "bye.sound"
end closeStack
```

This script asks if the user wants to update the message queue, which permanently deletes any messages marked for deletion. Depending on the user's response, the script sends either the exit command (indicated by the letter *x)* or the quit command (indicated by the letter *q)* to the mailx program to terminate the session. The commands are sent to the mailx program by writing to the communication pipe with the XFCN writexfcn. The special character lineFeed is appended to the command to simulate the user pressing the Return key.

The script then calls cleanupxfcn to perform the termination processing. This is safe now, since the mailx program will be terminating as a result of the exit or quit command just sent.

THE SELECTION SCRIPT

When the user clicks one of the message headers displayed in the first card of the stack, the following script is executed to display the selected message in the second card of the stack. Figure 2 shows a message card with a sample message.

```
on mouseUp
   global global_handle, linecount, vline
   put (item 2 of the clickLoc) + (the scroll of cd field one) into vline
   divide vline by the textHeight of cd field one
   put trunc(vline+.6) into vline
   if vline <= linecount then
      select line vline of cd field one
      if the textStyle of the selectedLine is italic then
         beep 1
      else
         put writexfcn(global_handle,(vline & lineFeed)) into writecount
         play "ZoomUp"
         go to card 2
         put fgetfxfcn(global_handle,250) into cd field msgx
      end if
   else
      beep 1
   end if
   select empty
end mouseUp
```

This script computes the line number of the selected message, checks to see that it's a valid message number, and then sends this value to the mailx program, causing that message to be displayed. The call to fgetfxfcn reads multiple lines of output into a field with one XFCN call. This is much faster than calling fgetsxfcn several times and inserting each line into the field.

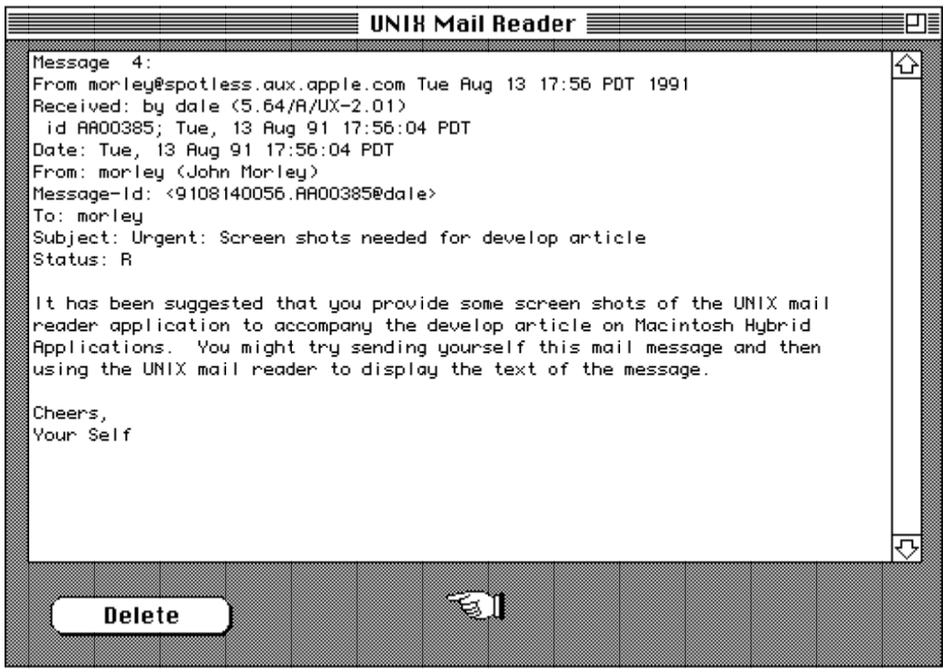


Figure 2
Message Card With a Sample Message

THE DELETE BUTTON SCRIPT

Users viewing the content of a message in the second card of the stack have the option of marking the message for deletion by clicking the Delete button. That causes the following script to be executed:

```
on mouseUp
   global global_handle, vline
   play "Cash Register"
   put writexfcn(global_handle,("d" & lineFeed)) into writecount
   put empty into cd field msgx
   go to card 1
   select line vline of cd field one
   set textStyle of the selectedLine to italic
end mouseUp
```

This script sends the delete command (the letter *d)* to the mailx program, clears the message field on the second card, and then changes the text style of the header for

that message to italic. This is an indication to the user that the message has been marked for deletion. The text style is checked in the selection script to prevent access to a deleted message.

PARTING THOUGHTS

I hope this article has given you some insight into the possible uses of programming Macintosh hybrid applications for A/UX, as well as some helpful techniques for doing this on your own. Although HyperCard was used to quickly implement the UNIX Mail Reader, the same techniques apply to using UNIX system calls from a Macintosh application written without HyperCard.

The *A/UX Developer's Tools* set of CD-ROM discs (APDA #B0596LL/A) contains more tools for dealing with both UNIX and Macintosh hybrid applications on A/UX. Developers interested in exploring this programming technique in depth may want to acquire that product to supplement the library and examples from this article.

REFERENCES

- *A/UX Programmer's Reference,* Section 2, Apple Computer, 1990.
- *A/UX Development Tools,* Chapter 2, Apple Computer, 1991.
- *A/UX Toolbox: Macintosh ROM Interface,* Apple Computer, 1990.
- *Apple HyperCard Script Language Guide: The HyperTalk Language,* Addison-Wesley, 1988.
- *Inside Macintosh,* Volumes I-V, Addison-Wesley, 1986.

THANKS TO OUR TECHNICAL REVIEWERS

Kent Sandvik, Joe Sokol, John Sovereign, Kristin Webster•

THE VETERAN NEOPHYTE

DON'T FENCE ME IN

DAVE JOHNSON

So here I am in Tucson, Arizona, standing in a dry streambed reeking of sunscreen with a notebook computer contraption in my hands and a walkie-talkie hooked on my pants and a squawking headset barely staying on my head, watching people tying white cotton string from one tree to another. Every time I move my head the headset slips forward a little more, but my hands are completely filled with this device, a fairly delicate thing due to the wire and duct tape and Velcro attachments, and I can't find any place safe to put it down. So I'm trying not to move my head, and I'm trying to look like an intelligent and purposeful person at the same time. It's hard.

I had written a little piece of the software that's running on the notebook computer, which is a GRiDPAD (pen-based, small, DOS). There are three others like it scattered around the canyon I'm in. Each has a tiny wireless modem and a battery stuck to the back, and each is running a sort of collaborative spreadsheet: data entered in one is quickly picked up by all the others. I'll call this contraption The Device from now on, to avoid funny capitalizations.

This event is an experiment in wireless communication and in collaboration at a distance. The main participants are a bunch of elementary school kids and their science teachers. They're here to study the canyon, and are split into four groups, each of which is being deployed to a different spot to take environmental measurements (pH, temperature, and so on) and to count species. Each group has one of The Devices, so once we're set up everybody will be able to see everybody else's data all the time. Each group also has a walkie-talkie, so they'll be able to talk to one another. We've just arrived for the very first field trials.

The string people have finished, neatly delineating the area that is our group's responsibility. The kids energetically begin fooling with thermometers and vials and yardsticks and—everyone's favorite—sling psychrometers. I finally find a rock flat enough to put The Device on, just as the headset lets go and slips over my eyes, pulling the earplug out of my ear. Gathering together the shredded remains of my dignity, I disentangle the headset from my ears and sunglasses, put it back on my head, and reinsert the little earplug in my ear. Who designed this thing? It just won't stay put. Maybe my head is a weird shape, or maybe this headset isn't meant to be used with sunglasses on. I take off the headset, take off my sunglasses, put the headset back on, and it's just as bad. Must be my head.

There are some very large issues that are touched on by this experiment, directly or indirectly: the implications of wireless communication, the nature of collaboration, and the nature of communication itself. I'll take a closer look at these issues, with the assumption up front that the goal is to provide technology that can enable—and hopefully enhance—collaboration at a distance.

First of all, what does wirelessness mean exactly? At face value, not much: it means that the wires are gone. The information flow hasn't changed; it's just using a different medium to flow in (or on, or around, or whatever). One person involved in this project compared the advent of wireless communication to taking down the fence around a herd of captive buffalo. Will they burst from their former confinement, joyously kicking their heels (I'm not sure if buffalo *have* heels, but you get the idea) and searching out new limits to conquer? Or will they not even notice, so used to the way things were that they can't conceive of anything else? That would be like some large, unimaginative corporation's vision of wirelessness: the

DAVE JOHNSON was born in southern California, but moved elsewhere as a small boy, so he's never even been on a surfboard and doesn't say "totally" unless he means it. He did most of his growing up in a suburb of Chicago (and consequently has strong opinions about pizza and snow shovels), but was dragged kicking and screaming back to southern California at the age of seventeen. As soon as he could, he left for college at Humboldt State University, which was as far as he could get from San Bernardino and still be in the California school system. He and his wife Lisa now make their home in San Francisco with their two dogs and two cats (which is, incidentally, the legal limit on animals in a San Francisco household). They're currently looking for a big house elsewhere.•

wires vanish, which saves a little money and hassle setting up the office, but nothing else changes. Clearly an approach like this is doesn't take advantage of the situation.

Wireless communication adds a whole new degree of freedom to human communication, that of space. It makes communication position-independent, it unsticks people geographically. This is potentially a far-reaching freedom if high-bandwidth wireless communication becomes pervasive. Many existing institutions—schools, businesses, and so on—that largely evolved from the need to have a central physical place are now freed up, just like the buffalo, and have lots of new territory to explore. Maybe there are other ways of doing things without people's bodies in attendance that we haven't thought of yet, since we haven't had the opportunity. But will people take advantage of this new freedom? Or are we too cozy, too entrenched in our old familiar ways to transcend them?

Periodically pushing my headset back onto my head, I look around: chaos reigns, but it's a good, busy, productive sort of chaos. The kids are measuring things, counting plants and birds and spiders, clambering around in the brush, chattering incessantly on their walkie-talkies, and entering data into The Device; the video crew—ostensibly documenting the event but mostly invoking Heisenberg's Uncertainty Principle—are running around poking their cameras and microphones at people; the adults are watching carefully, coordinating when necessary, commenting on the action over their own walkie-talkie channel.

One thing that surprises me about how the kids are relating to The Device is that they pretty much take it for granted. I guess I expected them either to ignore it or to be amazed by it, but they seem to just take it in stride, and use it when it's appropriate. It occurs to me that that's probably the best reaction we could hope for, to see it for what it is—a tool—and use it accordingly. And yes, by golly, the kids are collaborating a little. It's hard to tell, though, whether they'd collaborate on their own. They know that's what the adults are looking for, and every school kid learns early on to do what they think their teachers want.

What The Device provides very nicely is a kind of shared data space. It's as if each group has a magic looking glass that lets them all see the same thing, despite being physically separated. What The Device *doesn't* provide is a communication channel that lets them discuss the data with each other. The walkie-talkies provide that channel, although in a stilted manner. To collaborate effectively, you really need both kinds of communication: the communication of data (the shared thing you're collaborating on) and the communication of instructions (the conversation *about* the thing you're collaborating on).

One really interesting point (hotly debated in the post-study debriefing) is the importance of including voice capabilities in new communication/collaboration hardware. There is no question that voice, or more generally language, is the primary medium we use to communicate. The debate pivots on whether voice should be provided de facto in the technology or whether there is some new, as-yet-undiscovered mode of communication that technology can provide us that would render voice communication unnecessary, or at least optional. Pretty heady stuff, no? We're talking about a quantum leap here, a revolutionary change away from the familiar. Using voice *only* because it's the way we do it now might hinder or prevent our moving forward into the grand and glorious communication revolution.

On the other hand, why should we not provide voice? Voice compression technology is advancing quickly because of cellular demand. Good-quality voice can be transmitted at 4800 bits per second now, and soon it will be 2400. Those bandwidths are easy and getting easier and cheaper all the time. Full-duplex real-time voice transmission (in other words, a conversation) also provides a huge degree of familiarity to people, and it really adds to the feeling of being connected. Isn't that the goal? To be apart but not to feel apart? Voice alone provides lots of that "in the same space" kind of feeling.

The ideas and issues discussed in this column did not arise in a vacuum. I'd like to acknowledge the others who inspired me to think about these things and provided the grist for the mill. In particular, thanks are due to Wayne Grant and Rick Borovoy, who dreamed up this project in the first place and were kind enough to let me participate and observe. Also, I want to thank everyone who attended the post-study debriefing meeting: Tyde Richards, Kathy Ringstaff, Brian Reilly, and Rifaat "Rick" Dayem. It was at that meeting that I really began to see the implications and issues surrounding wireless communications and communication technology in general.•

My thoughts seem to keep circling back to one central question: should we use this radical new wireless technology to adopt (and then attempt to improve upon) the way people communicate and collaborate now, or should we throw out all the rules and go for something really new? If you're interested in selling products today, probably you want something like the former. If you're a wild-eyed visionary intent on changing the world, you'll tend toward the latter. And in reality, let's face it, you always end up with something in between.

In my humble and inexpert opinion, we ought to build from familiarity. I don't really have any facts or studies to back me up, just personal observations and a strong feeling that building from existing modes of communication is the most effective way to get what we want. And I don't think that this method necessarily precludes radical advances. Look at the Macintosh. Few people would argue against the fact that it was a radical leap from any other machine. But the very thing that made it radical was its familiarity, its *humanness*. Humanness in a machine is extremely powerful, and extremely attractive to people.

Maybe I just have a soft spot for humans, being one myself, but I tend to give them a lot of credit. The advent of wireless communication does remove a fence, but it is a fence around people, not buffalo. People explore, it's one of the things they're best at. They won't stay huddled in the center of the corral for long. And I also believe that people already know how to collaborate. If you and I are standing next to each other looking at a piece of paper that we're working on together, we won't have trouble proceeding. So if we *can* provide communication tools that are truly transparent, I think that collaboration will fall out automatically. Now don't get me wrong. I know that there's plenty of room for improvement in the way people work together, but I think the communication tools need to be available before we can make much progress in collaboration. And yeah, I really think we should provide voice, if we can. If there *are* better ways to communicate, I suspect it will be quite a while until we find them, and in the meantime voice is the best we've got and really does make people feel connected.

It will of course take a while for all these things to come to fruition, but we are tantalizingly close. I think the best way to proceed is to build what we can right now, and get it out into the world as quickly as possible. Then, to learn how to make it better, we should watch very closely what people do with it. This way the tools will evolve as natural extensions of the people who use them, which seems like a good goal to me. Maybe if the headset makers had followed this approach . . .

MACINTOSH Q & A

Q *When my application is running, it relies on the MultiFinder's "puppet strings" (which choose Open from the application's File menu and suppress the SFGetFile dialog) to open a document that was double-clicked in the Finder. Why doesn't this work under System 7? The high-level event-aware bit in my 'SIZE' resource is clear.*

A System 7 will not pull puppet strings for an application that makes use of the System 7 Standard File routines, such as StandardGetFile and CustomGetFile, nor will it pull them if the application's high-level event-aware bit is set.

If you update an older application to take advantage of any System 7 features, be sure to also add support for the 'odoc' and other required Apple events. Sample code showing how to support the required Apple events is available on the System 7 Golden Master CD.

Q *Why does Gestalt tell me I have Color QuickDraw features on a non-Color QuickDraw machine?*

A The gestaltQuickdrawFeatures ('qdrw') selector, used for determining your system's Color QuickDraw features, has a bug that causes it to tell you incorrectly that noncolor machines have color. The fix is quite simple: Gestalt has another selector, gestaltQuickdrawVersion ('qd '), which simply returns the QuickDraw version number. This version number is < gestalt8BitQD for classic QuickDraw and >= gestalt8BitQD for Color QuickDraw (see *Inside Macintosh* Volume VI, page 3-39, for more information). The trick is to ask Gestalt for the QuickDraw version first; once you've determined that you have Color QuickDraw, the 'qdrw' selector is OK to use to find out specifics.

Q *What do we return to the Apple event handler if we get an application error while processing a standard event, Edition Manager event, or custom Apple event for commands and queries? Probably not errAENotHandled, since that means we didn't handle the event, which is different from trying to handle it and failing. Would it be errAEFail? What if we want to return more specific error information? Do we define our own errors, or try to use Apple's errors such as memFullErr or parmErr?*

A You pass back errAENotHandled, because it's true, and because some simple applications will not be able to handle anything more than that. What you can also do, and what most commercial applications will do (particularly applications that want to be scripting savvy), is add errn and errs parameters to the reply record for that event (as shown on page 6-49 of *Inside Macintosh* Volume VI). You can be as descriptive as you like in the text—the more the better, in fact, since this text will be seen at the user level usually. The errn value you pass back can be the system error number; then the sending program may be able to recover and try again.

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, Sonya Andreae, Mark Baumwell, Mike Bell, Jim "Im" Beninghaus, Rich Collyer, Neil Day, Tim Dierks, Marcie "MG" Griffin, C.K. Haun, Pete Helme, Dave Hersey, Dennis Hescox, Jim Luther, Joseph Maurer, Jim Mensch, Guillermo Ortiz, Craig Prouse, Dave Radcliffe, Greg Robbins, Jack Robson, Kent Sandvik, Paul Snively, Bryan "Stearno" Stearns, Forrest Tanaka, Vincent Tapia, John Wang, and Scott "Zz" Zimmerman for the material in this Q & A column.

Q *According to the QuickTime Movie Toolbox documentation, "The Movie Toolbox maintains a set of global variables for every application using it." How much global memory is required? Our application is shy on global data space.*

A The information maintained is not kept with the application's global variables. The handle created by the EnterMovies call is stored in the system heap, not in the application heap. You don't have to worry about how much space to allocate in your application. This initialization does not affect your A5 world either.

EnterMovies initializes everything, including setting up the necessary data space and creating a handle to it. When you're done, be sure to make a call to ExitMovies to clean up the QuickTime data.

If an application makes multiple calls to EnterMovies, a different set of "globals," or data area, is set up for each call. A call to ExitMovies should be made before exiting the area that made the call to EnterMovies. For example, an application that uses QuickTime will call EnterMovies and set up the QuickTime world. Then an external may be called upon that wants to use QuickTime. This external would have to make a call to EnterMovies to set up another QuickTime world for its use. Before leaving, the external should call ExitMovies to clean up after itself. The application can then continue to use QuickTime with the original world it set up.

Q *Why does the longword at location $0 get changed to 0x40810000 at every trap?*

A In System 7, the Process Manager slams a benign value into location $0 to help protect against bus errors when an application inadvertently dereferences a NIL pointer. (There's no bus-error checking on writes to ROM, so the "benign value" is usually ROMBase+$10000.)

If you're debugging, you want the opposite effect: you want these inadvertent accesses to "cause" bus errors. If you put a different value in location $0 before the Process Manager starts up (that is, from MacsBug or TMON initialization, or from an INIT like EvenBetterBusError), it will force that value instead. For more information, see the "Macintosh Debugging" article in this issue.

Q *I'm filling a large buffer with one SCSIRead call. What happens if the Macintosh runs under System 7 with virtual memory (VM) and parts of my buffer are swapped out?*

A Parts of your buffer must *not* be swapped out. Before calling SCSIGet, you must ensure that all code and buffers accessed while the SCSI bus is busy are held in physical memory. If there isn't enough real memory to allocate a full buffer, the application must request smaller blocks (if possible) from the SCSI

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.•

device, because it's not possible to swap in and out any buffer space during a single I/O operation. Page faults are not serviced while SCSI I/O is in progress. If SCSI I/O is performed at device driver-level Read or Write calls, VM holds your buffer for you. Otherwise, you are responsible for doing this yourself. If there is insufficient physical memory for VM to hold your buffers for you, the Read or Write call fails with an error result.

In general, I/O buffer space used by drivers *must* be held in real memory for the duration of the I/O operation. This is especially true for SCSI I/O because VM uses SCSI to swap virtual memory in and out, and encountering another page fault would cause a bus error. Device Manager-level I/O handles this automatically, by holding down the appropriate memory when the driver is entered through a Read or Write call. The Device Manager does not take care of this for you when the driver is entered through a Control or Status call, however. If the SCSIRead call is made from within a device driver as a result of a PBRead, no special action is necessary. Any other type of code must be very careful not to cause page faults between SCSIGet and SCSIComplete. This requires holding or placing in the system heap any code or data structures referenced during this time.

Q *Is there anything special that a Macintosh hard disk or a removable cartridge driver must do to be fully compatible with System 7?*

A One important thing you should be aware of regarding removable cartridges is that a cartridge can't be removable if VM is to use it for a backing store. Some removables allow you to fix this with a SCSI command to prohibit ejection and, just as important, the drive must be marked nonejectable in the drive queue.

Here are a couple of suggestions: Read Macintosh Technical Note #285, "Coping with VM and Memory Mappings." Also, take a look at the Memory Management chapter of *Inside Macintosh* Volume VI and the Virtual Memory paper (Goodies:VM Goodies:VM Paper) on the System 7 Golden Master CD.

Q *What does the "!" mean when I use the MacsBug Heap Zone (HZ) command? It appears in front of one of the zone names listed, or just after the address if the zone doesn't have a name.*

A MacsBug's HZ command does a quick-and-dirty heap check, and if it thinks something is wrong with a heap, it puts the exclamation point after the address range of the heap. If you select the heap flagged with a "!" with the Heap Exchange (HX) command and then use the regular Heap Check (HC) command, MacsBug tells you what it thinks is wrong with that heap.

Q *I want to display only visible files and folders in a Standard File dialog, but I can't find a way to filter out invisible folders—specifically the 000Move&Rename folder. The FileFilter routine filters only files, not folders. If I put in a nonzero TypeList, invisible folders seem to be removed, but I want to open all types of files, just not invisible files or folders. Any suggestions?*

A This is, in fact, impossible under System 6 using general methods. The problem is that passing -1 as numTypes means not only to display all items, but to display invisible items. A file filter can be used to remove the invisible files but cannot affect invisible folders. The only current way to do this is to use CustomGetFile under System 7, as described in the Standard File Package chapter of *Inside Macintosh* Volume VI. This provides a filter that allows you to filter both files and folders. This will give you the right functionality, but will work only under System 7. We recommend that you use this method under System 7, and a more standard SFGetFile when running under earlier systems.

Q *How can I obtain the volume reference information in my DlgHook function for a file selected by the user before SFPPutFile or SFPGetFile has completed the reply record?*

A On exit, SFPGetFile and SFPPutFile generate a working directory reference number in the vRefNum field of the reply record. This is not available to you from within the operation of a DlgHook function. WDRefNums are provided to allow compatibility with older, pre-HFS functions that took vRefNum values of integer size with the older flat file system.

We suggest that, unless you plan to support the flat file system of 64K ROM Macintosh systems, you move your file system interfaces to the HFS interfaces documented in the File Manager sections of *Inside Macintosh* Volumes IV and V (or to the equivalent high-level calls as documented in Macintosh Technical Note #218, "New High-Level File Manager Calls"). If you're using the HFS calls, low-memory globals SFSaveDisk and CurDirStore contain, respectively, the negative of the "real" volume reference number for the current volume and the HFS ID of the directory that Standard File is displaying. You then have all the information you need to create, open, rename, or delete files from within the SFPGetFile and SFPPutFile DlgHook functions. If a user is accessing an MFS volume on an HFS system, these calls are designed tohandle file access transparently.

Moving your file system interfaces to the HFS-level conventions has a side benefit of being closer to the System 7 file system specifications. If you look at the new high-level file system calls in *Inside Macintosh* Volume VI, you'll recognize much of the HFS information embedded in the new data structures.

If your file system interfaces depend on MFS-style vRefNums, or WDRefNums in the HFS nomenclature, you can use the HFS functions PBOpenWD, PBCloseWD, and PBGetWDInfo to open, close, and obtain volume reference numbers and directory IDs. This is particularly important if, for instance, you're using the THINK C ANSI file I/O functions, which rely on SetVol to operate correctly.

Complete information on the HFS-level calls that will be most useful in Standard File customization is contained in the File Manager chapters of *Inside Macintosh* Volumes IV and V, and in Macintosh Technical Notes #66, 77, 102, 140, 179, 190, and 218. For C users, Macintosh Technical Note #246 summarizes a list of the difficulties with mixing C file I/O with Macintosh file I/O. Macintosh Technical Notes #47 and 80 discuss a few points of Standard File customization from the point of view of HFS.

Q *Why does GetGWorldPixMap (when called on a Macintosh II, IIcx, or IIx running system software version 6.0.5 or 6.0.7 with 32-Bit QuickDraw 1.2) return a combination of the device field (two bytes) and the first two bytes of the portPixMap field? Is this a bug?*

A Your analysis of GetGWorldPixMap is exactly right: It doesn't work correctly in system software version 6.0.5 and 6.0.7 with 32-Bit QuickDraw 1.2. It returns a value that's two bytes before the value it's supposed to return.

The solution is to use GWorldPtr->portPixMap instead of GetGWorldPixMap. It's safe to do this, but you should use GetGWorldPixMap under System 7. Not only is the bug fixed there, but dereferencing the port is dangerous under System 7 because it may not be CGrafPort. Use Gestalt with the gestaltQuickdrawVersion selector to determine whether you can use GetGWorldPixMap. If Gestalt returns a value from gestalt8BitQD ($0100) through gestalt32BitQD12 ($0220), then GetGWorldPixMap either doesn't exist or is the buggy version. If it returns gestalt32BitQD13 ($0230) or higher, then GetGWorldPixMap does exist and works correctly. Interestingly, GetGWorldPixMap can be called on a black-and-white QuickDraw machine under System 7. It returns a handle to a structure which should be treated as a BitMap structure, though there are some undocumented fields after the normal BitMap fields. To tell whether GetGWorldPixMap is available on a black-and-white QuickDraw machine, you must check the system software version by calling Gestalt with the gestaltSystemVersion selector. If it returns $0700 or higher, GetGWorldPixMap is available.

Q *DrawText with srcCopy takes six times as long as with srcOr now that my Macintosh is running System 7. Why is this so slow? Is this a bug in System 7?*

A It's true that srcCopy is slower than srcOr when handling text, especially in color mode. This loss in speed occurs because CopyBits is a lot smarter than it used to be. It can handle foreground and background colors a lot better, but that improvement came at the cost of speed. Our recommended method for drawing text is to erase before drawing, and use srcOr to draw, not srcCopy. Alternatively, you could draw colorized text in srcOr mode off screen and then use CopyBits to draw it on the screen in srcCopy mode without colorization.

Q *I'm creating PICTs that are comprised of many lines drawn in srcOr mode. When using the LaserWriter 6.x or 7.x driver with the Color/Grayscale radio button selected, some lines fail to print. Why is this happening?*

A The problem is a bug in the LaserWriter driver. Sometimes, when using a CGrafPort, the driver doesn't reproduce lines drawn in srcOr mode. (A CGrafPort is used when the Color/Grayscale print option is selected; in Black & White print mode, a regular grafPort is used.) A workaround is to use srcCopy instead of srcOr when drawing QuickDraw objects within your PICTs.

Q *Is there any way to determine whether I'm printing to either a color printer or a printer simulating color, such as the LaserWriter set for Color/Grayscale?*

A Check the grafPort returned by your call to PrOpenDoc. If the rowBytes of the grafPort is less than 0, the Printing Manager has returned a color grafPort. You can then make Color QuickDraw calls into this grafPort. LaserWriter driver version 6.0 and later returns a color grafPort from the PrOpenDoc call, if the Color/Grayscale button has been set.

The following code fragment demonstrates this:

```
(* This function determines whether the port passed to it is a *)
(* color port. If so, it returns TRUE.                         *)
FUNCTION IsColorPort(portInQuestion: GrafPtr): BOOLEAN;
BEGIN
  IF portInQuestion^.portBits.rowBytes < 0 THEN
     IsColorPort := TRUE
  ELSE
     IsColorPort := FALSE;
END;
```

A third-party printer driver should return the type of grafPort that represents the abilities of its printer. Therefore, if the printer supports color and/or grayscale, and if Color QuickDraw is present, the application will receive a color grafPort after calling PrOpenDoc. Otherwise, if the Macintosh you're

running on does not support Color QuickDraw, you should return a black-and-white grafPort.

Q *If I use the PostScriptHandle PicComment to send PostScript code to the LaserWriter driver, do I need to open a picture and then draw the picture to the driver, or can I just use the PicComment with no picture open while drawing to the printer's grafPort?*

A You don't need to create a picture with your PicComment in it and draw the picture to the driver. The best method for sending PostScript code to the LaserWriter is to use the PostScriptHandle PicComment documented in Macintosh Technical Note #91, "Optimizing for the LaserWriter—Picture Comments," as shown below.

```
PrOpenPage(...)
{ Send some QuickDraw so that the Printing Manager gets a }
{ chance to define the clipping region. }
PenSize(0,0);
MoveTo(0,0);
LineTo(0,0);
PenSize(1,1);
PicComment(PostScriptBegin, 0, NIL);
{ QuickDraw representation of graphic. }
MoveTo(100, 100);
LineTo(200, 200);
{ PostScript representation of graphic. }
thePSHandle^^ := '100 100 moveto 200 200 lineto stroke';

PicComment(PostScriptHandle, GetHandleSize(thePSHandle),
          thePSHandl);
PicComment(PostScriptEnd, 0, NIL);
PrClosePage(...)
```

The above code prints a line on any type of printer, PostScript or not. The first MoveTo/LineTo combination is required to give the LaserWriter driver a chance to define a clipping region. The LaserWriter driver replaces the grafProcs record in the grafPort returned from PrOpenDoc. In order for the LaserWriter driver to get execution time, you must execute a QuickDraw drawing routine that calls one of the grafProcs. In this case, the MoveTo/LineTo combination calls the StdLine grafProc. When StdLine executes, it notices that the grafPort has been reinitialized, and therefore initializes the clipping region for the port. Until the MoveTo/LineTo combination is executed, the clipping region for the port is set to (0,0,0,0). If PostScript code is sent via the PostScriptHandle PicComment before executing any QuickDraw routines, all PostScript operations will be clipped to (0,0,0,0).

The next thing that's done is to send the PostScriptBegin PicComment. This comment is recognized only by PostScript printer drivers. When the driver receives this comment, it saves the current state of the PostScript device (by executing the PostScript gsave operator), then disables all QuickDraw drawing operations. This way, the QuickDraw representation of the graphic will be ignored by PostScript devices. In the above example, the second MoveTo/LineTo combination is executed only on non-PostScript devices.

The next PicComment is PostScriptHandle, which tells the driver that the data in thePSHandle is to be sent to the device as PostScript code. The driver then passes this code unchanged to the PostScript device for execution. The PostScriptHandle comment is recognized only by PostScript printer drivers.

The last PicComment, PostScriptEnd, tells the driver to restore the previous state of the device (via a PostScript grestore call), and to enable QuickDraw drawing operations.

Since most PicComments are ignored by QuickDraw devices, only the QuickDraw representation is printed. Since PostScriptBegin tells PostScript drivers to ignore QuickDraw operations, only the PostScript representation is printed on PostScript devices. This is a truly device-independent method for providing both PostScript and QuickDraw representations of a document.

Q *How do users install the Macintosh Communications Toolbox (CTB)?*

A The Communications Toolbox consists of two parts: the CTB managers and the CTB tools. The installation procedure for CTB tools is different between System 6 and System 7. Under System 6 the CTB tools are dragged from the *Basic Connectivity Set* disk to the Communications Folder in the System Folder on the hard disk. Under System 7 the CTB tools are dragged from the *Basic Connectivity Set* disk to the Extensions folder in the System Folder on the hard disk. Of course, because of the way System 7 works, CTB tools can simply be dragged to the System Folder and the Finder will automatically move them to the Extensions folder where they belong.

No installation of the CTB managers is required under System 7, since System 7 includes the Communications Toolbox as part of the system. Users running System 6.0.5 should use the Installer and Install script on the *Communications 1* disk to install the CTB managers and other resources onto their hard disks. Users running System 6.0.7 should use the Installer and Install script on the *Network Products Installer* disk, which is part of the System 6.0.7 set users receive with their Macintosh systems. The *Network Products Installer* disk does not contain the CTB managers and other resources, but it prompts the user to insert the *Communications 1* disk during the installation procedure.

The *Basic Connectivity Set* and *Communications 1* disks should be shipped with your CTB-aware product. Contact Apple's Software Licensing group (AppleLink SW.LICENSE) for a licensing agreement to ship the disks.

Q *Can any AppleTalk routines be called at interrupt time? Inside Macintosh says that DDPWrite and DDPRead can't be called from interrupts. If all higher-level AppleTalk protocols are based on DDP, it seems that they all would not work.*

A The AppleTalk routines you can't call at interrupt time are the original AppleTalk Pascal Interfaces listed in *Inside Macintosh* Volume II; these are also known as the "Alternate Interface" AppleTalk routines, or ABPasIntf.

The Alternate Interface routines cannot be called at interrupt time because they allocate the memory structures needed to make the equivalent assembly language AppleTalk call. For example, when the NBPLookup routine is called, it's passed a handle to an ABusRecord. NBPLookup then has to allocate an MPPParamBlock and move the parameters from the ABusRecord into the newly allocated MPPParamBlock. Then NBPLookup makes a LookupName call, passing it the MPPParamBlock. When LookupName completes, NBPLookup must move results into the ABusRecord and release the memory used by the MPPParamBlock. Since memory is allocated and released within the routine, it cannot be called at interrupt time.

With that out of the way, the calls you can make at interrupt time (with some restrictions listed below) are what Apple calls the "Preferred Interface" AppleTalk routines. Most of the Preferred Interface routines are listed on page 562 of *Inside Macintosh* Volume V. There are a few additional calls that were added after the publication of *Inside Macintosh* Volume V; they're documented in the AppleTalk chapter of *Inside Macintosh* Volume VI.

The Preferred Interface AppleTalk routines can be made at interrupt time as long as:

- You make them asynchronously with a completion routine (that is, the asynch parameter must be TRUE and you must provide a pointer to the completion routine in the ioCompletion field of the call's parameter block). Making a call asynchronously and polling ioResult immediately afterward within the *same* interrupt-time code (which is basically the same as making the call synchronously) is *not* the same as using a completion routine.
- They are not listed as routines that may move or purge memory. The Preferred Interface routines do not allocate or dispose of any memory, since they're just high-level ways to make the assembly language AppleTalk calls and are *not* built upon the old Alternate Interface routines.

Q *Why do I get only the left channel of a stereo sound out of my Macintosh IIcx?*

A The only Macintosh models that combine the two stereo channels into one for playback out of the internal speaker are the Macintosh SE/30 and the IIsi. All others use just the left channel. If you would like to check for the machine's ability to do mixing, you can use Gestalt. This is documented in *Inside Macintosh* Volume VI, page 22-70. Bit 1 of the Gestalt selector for sound will tell you whether the machine has stereo mixing on the internal speaker.

Q *Inside Macintosh Volume VI, page 2-22, recommends updating the window positions in a file without changing the last modification date and time on the file. How do I alter a file without automatically changing the timestamp?*

A To modify the contents of a file's data or resource fork without changing the last modified date, get the modified date before performing any save operations on the file and restore it when you're done. You can use the PBHGetFInfo and PBHSetFInfo calls to do this. A short Pascal snippet that modifies the contents of a known file's resource fork without modifying its modification date is included in the Snippets folder on this issue's *Developer CD Series* disc. The code shows how the parameter block is filled in with the file's information at the start of the routine with a PBHGetFInfo call, and the same data is then used without modification to set the file information at the end of the routine with a PBHSetFInfo. *Inside Macintosh* Volume IV, page 150, tells you which fields can be changed with PBHSetFInfo.

Q *Help! I've just added Balloon Help to my application, but I'm having some problems. Whenever a balloon appears, it immediately begins floating away off the top of the screen. What can I do to stop this madness?*

A It appears you failed to heed our warning when it comes to routines that can move balloons. Consult Appendix D of *Inside Macintosh X-Ref*, "Routines That May Pop Balloons or Cause Barometric Disturbances," for a complete listing of these help balloon meteorological nightmares. In addition, be sure to call the new trap HMSetBalloonContents:

```
OSErr HMSetBalloonContents (balloonContents: INTEGER);
CONST { types of balloon contents }
  helium = 0;
  air = 1;
  water = 2;
  whippedCream = 3;
```

with balloonContents set to something greater than helium.

Q *How can I "dim" text in my Apple IIGS application, similar to the way the Toolbox dims menu titles and menu options, and the way that HiliteControl dims a button name?*

A Dimming text is a piece of cake. Here's how the Apple IIGS Menu Manager does it in both 640 and 320 mode. The same thing works great for both.

1. Calculate the bounding box of the text.
2. Fill it with the proper background color for the operation in question.
3. Draw the text.
4. Execute code that looks something like this:

```
            pea dimmed>>16
            pea dimmed
            _SetPenMask         ;Set the pen mask to "dimmed" mask
            pea textRect>>16
            pea textRect        ;Push pointer to text boundary rect
            lda backColor       ;Get text's background color
            and #$00FF          ;Always solid

; Here you need to do something in your code to get the
; appropriate pattern for the text you're drawing. This will be
; one of the 16 patterns in the 512-byte table, starting with 16
; bytes of $00, 16 bytes of $11, and ending with 16 bytes of $FF.
; We leave the routine GetColorPtr for you to code, but our
; example assumes it returns the pointer we need in A (low word)
; and X (high word).

            jsr GetColorPtr     ;(see above)
            phx                 ;high word
            pha                 ;low word
            _FillRect           ;Fill will be dithered
            pea nor_mask>>16    ;Reset drawing mask to normal solid
            pea nor_mask
            _SetPenMask
            rts

textRect    DC.B $00,00,00,00   ;Put your rectangle here
backColor   DC.W $0000          ;Background color of text to dim

dimmed      DC.B $55,$AA,$55,$AA,$55,$AA,$55,$AA
nor_mask    DC.B $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF
```

Yes, it's really this easy!

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 Eric Soldan for the material in this Q & A column.•

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.•

Q *When I call FixFontMenu from my Apple IIGS new desk accessory (NDA), everything works fine, but if the current application has a font menu it stops working. What's wrong?*

A FixFontMenu keeps only one correspondence between menu item IDs and font family numbers—if you call FixFontMenu from an NDA, you blow away the already existing correspondence that the application was using, maintained within the Font Manager. ItemID2FamNum then works only on item IDs corresponding to your NDA's font menu, and these usually are unrelated to the values the application passes from its font menu. Worse, FamNum2ItemID will subsequently return menu item IDs for font family numbers that have nothing to do with the application's menus; depending on how the application operates on the resulting item ID, this could be disastrous.

Fortunately, doing this yourself is fairly easy with the Font Manager's help. CountFamilies tells you how many font families there are, and FindFamily returns the name of each of them. You can manipulate this information into a font menu in a fairly straightforward way, using standard Menu Manager calls. While you're at it, you can also use FindFontStats to figure out what point sizes and styles are available for each font family, so you can indicate this in your Size and Style menus. You could even use the information to build your own font-choosing dialog, much as HyperCard IIGS does. Remember that your NDA should run in either 320 or 640 mode, so don't make the dialog box too wide.

Q *When using an Apple IIGS run queue item, can I use the second reserved field to find out when the item was last executed? I assume this value is the tickCount. Currently, I just get the current tickCount and subtract the last tickCount. Using this field could save me one tool call, and when doing animation via a RunQ, every extra tick counts.*

A No, you should not use undocumented fields in the run queue header because they could change with future software releases. However, there is a fast way to get the current tick count. Pass refNum $0005 to the GetAddr function in the Miscellaneous Tools once each time your program runs, and it tells you the location of the tick counter. Since the tick count changes during heartbeat interrupts, be sure to disable interrupts around your direct accesses to the tick counter (PHP, SEI, access tick count, PLP).

If your application makes certain tool calls frequently, it may be worthwhile to short-circuit the tool dispatcher and transfer control to the Toolbox function directly (if the tool takes a long time to execute anyway, it isn't worth it). You can get a function's address and work area pointer by calling GetFuncPtr and GetWAP in the Tool Locator. When the function gets control, the Y and A registers must contain the tool set's work area pointer, the X register must

contain the function number, and there must be *two* RTL addresses on the stack.

Q *Does the Apple IIe Card for the Macintosh LC have a technical reference manual?*

A There's no separate technical reference manual. Use the *Apple IIe Technical Reference* (Addison-Wesley), together with Apple IIe Technical Note #10, "The Apple IIe Card for the Macintosh LC."

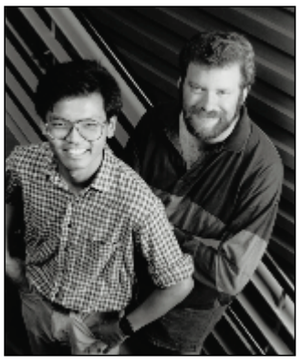
Q *What's the proper method of saving the Apple IIGS Super Hi-Res (SHR) screens? If my application both uses shadowing and is fast-port aware, the restored screen colors are garbaged. Can I simply use HandToPtr with ptr representing the screen addresses, or will this mess up the scan-line control byte (SCB) restoration since these are read-modify-write locations?*

A The shadowed screen's SCBs may not be correct, so by saving and restoring them you're causing random data to be restored into the standard SCBs. Your best bet, if shadowing is on, is to turn shadowing off, restore the bank $01 screen, including its SCBs and color tables, turn shadowing on, and restore the $E1 screen and contents. If you don't want to double-restore the $E1 screen and $01 screen, you should turn shadowing off, restore the bank $01 color tables/SCBs, turn shadowing on, and restore the $01 screen. But, since these screens are never guaranteed to be the same when you save, you should always restore both screens separately.

QuickDraw never touches the shadowed screen SCBs, so if the fastPort flag is set you can ignore the restoring of the bank $01 SCBs/color tables, since the application promised not to interfere with them. But since this won't save very much time, you probably shouldn't worry about it.

Q *I am using an Apple IIGS utility to generate resources for my application, and I noticed that some of the resource IDs generated are in the range $07FF0000 to $07FFFFFF, which is reserved for the system software's use. What's happening?*

A Your utility is calling UniqueResourceID with an IDRange of $FFFF, to request any unused resource ID. A bug in system software version 5.0.x allows UniqueResourceID to accidentally return a system-range resource ID *if* any system-range resources of that type are already present. This will be fixed in System 6. In the meantime, utilities can use UniqueResourceID with IDRange values other than $FFFF, and you should watch your resource IDs carefully to avoid using system-range resource IDs.



BE OUR GUEST

GWORLDS AND NUBUS MEMORY

FORREST TANAKA AND PAUL SNIVELY

In Developer Technical Support, we're asked from time to time how to make a GWorld so that its pixel image uses memory on a NuBus™ card rather than memory in the application's heap. The idea is to create a GWorld, put the address of the card into the GWorld, use QuickDraw to draw into the GWorld, which effectively draws into the NuBus card's memory, and display the resulting image on the screen. Doing this in a way that works well with the 8•24 GC version of QuickDraw and with whatever QuickDraw brews up in the future isn't possible without breaking a few guidelines. We'll talk about the reasons for this and what you can do instead. For the more cavalier among you, we'll also talk about the least offensive method of coercing a GWorld so that it uses memory on your NuBus card.

NewGWorld allocates off-screen buffers simply by using the same Memory Manager calls that you can make. To allocate the memory, NewGWorld simply calls NewHandle to allocate the buffer in your application's heap unless you have the useTempMem bit set, in which case it allocates the buffer in temporary memory. It then tries to move the buffer as high in your heap as possible by calling MoveHHi. That's really all there is to it. The GWorld's pixMap, GDevice, and CGrafPort are allocated similarly—they're all allocated in your heap using regular Memory Manager calls with no special options, patches, or other nefarious tricks.

None of this changes when you have the 8•24 GC software active—all memory is still allocated out of your application's heap. Once you start drawing into the GWorld, though, the GC software can copy the parts of the GWorld to the 8•24 GC memory. The GWorld and its parts still occupy your heap's memory though, regardless of whether it's cached on the 8•24 GC card.

If you have a NuBus card with gobs of memory, NewGWorld can't take advantage of it because the Memory Manager calls that it uses can't allocate memory on NuBus memory cards. There are no options to NewGWorld or any other GWorld calls that let you say, "There's lots of memory over on this NuBus card, all for you." While GWorlds are absolutely fantastic for creating off-screen drawing environments for most of the usual kinds of situations, they're just not appropriate if you want complete control over where or how the parts of a GWorld are allocated.

QuickDraw is the only thing that's supposed to know how GWorlds are constructed. We know that they're CGrafPorts and we can get their pixMap, GDevice, and off-screen buffer, but we shouldn't make any assumptions about how they were allocated and where they are. For example, we know that the off-screen buffer is allocated as a handle now, but that won't necessarily be the case in the future. There's no guaranteed way to tell which way it was allocated, or even if NewGWorld uses the Memory Manager to allocate it at all (which it always does currently, of course). Even the GWorld's CGrafPort is allocated as a handle that just happens to be always locked. If you try to dispose of a GWorld in which you've modified the baseAddr, you'll need DisposeGWorld to make sure everything is deallocated properly, but its behavior is undefined when it tries to deallocate the off-screen buffer.

So if you want to use the memory on your NuBus memory card and feel comfortable that you're not relying on something that could change, you're going

FORREST TANAKA has been in Developer Technical Support just shy of two years after a stint with unemployment and trying to get a job at Apple. Before that, he got a BSCS while writing Macintosh device drivers for scanners and writing utility software for a PBX. Now he's working with anything that makes images appear on the Macintosh's screen while avoiding anything that makes images appear on paper. Whenever he's not working, eating, sleeping, watching TV, reading, or watching a movie, he's out riding his bike and wondering whether he should shave his legs.•

PAUL SNIVELY, formerly of Apple's DTS group, came to Apple from ICOM Simulations, Inc., the land of the TMON debugger. He wrote the *TMON 2.8 User's Guide* and has written for *MacTutor* magazine. His interests include natural-language processing, knowledge representation, adventure-game programming, horror and suspense, hiking, camping, spelunking, and other things better left unsaid.•

to have to create your own off-screen drawing environment by creating an off-screen pixMap, a color table if your off-screen drawing environment uses indexed colors, a GDevice, and a CGrafPort. The April 1989 edition of Macintosh Technical Note #120, "Drawing Into an Off-Screen Pixel Map," covers creating your own off-screen pixMap, CGrafPort, and color table, but it requires you to have the same depth and the equivalent color table that the screen has, so it just steals a screen's GDevice. We think it's always a good idea to create your own GDevice when you draw off screen. If you use a screen's GDevice for drawing off screen, you have to depend on that GDevice's depth and color table. By creating your own GDevice, your off-screen drawing environment can use any depth and color table you want at any time and still be insulated from whatever changes the user makes with the Monitors control panel.

To create your own GDevice, it's better not to use NewGDevice because it always creates the GDevice in the system heap; it's better to keep your data structures in your own heap so that they don't get orphaned if your application quits unexpectedly and that precious system heap space is preserved. Here's what you should set each of your GDevice's fields to be:

gdRefNum	Your GDevice has no driver, so just set this to 0.
gdID	It doesn't matter what you set this to; you might as well set it to 0.
gdType	Set to 2 if your off-screen pixMap uses direct colors (16 or 32 bits per pixel) or 0 if it uses a color table (1 through 8 bits per pixel).
gdITable	Allocate a small (maybe just 2-byte) handle for this field. After you're done setting up this GDevice and your off-screen pixMap, color table (if any), and CGrafPort, set this GDevice as the current GDevice by calling SetGDevice, and then call MakeITable, passing it nil for both the color table and inverse table parameters, and 0 for the preferred inverse table resolution.
gdResPref	We reckon that more than 99.9% of all inverse tables out there have a resolution of 4. Unless you have some reason not to, we'd recommend the same here.
gdSearchProc	Set to nil. Use AddSearch if you want to use a SearchProc.
gdCompProc	Set to nil. Use AddComp if you want to use a CompProc.
gdFlags	Set to 0 initially, and then use SetDeviceAttribute after you've set up the rest of this GDevice.
gdPMap	Set to be a handle to your off-screen pixMap.
gdRefCon	Set to whatever you want.
gdNextGD	Set to nil.
gdRect	Set to be equal to your off-screen pixMap's bounds.
gdMode	Set to -1. Why? We're not sure. This is intended for GDevices with drivers anyway.
gdCCBytes	Set to 0.
gdCCDepth	Set to 0.
gdCCXData	Set to 0.
gdCCXMask	Set to 0.
gdReserved	Set to 0.

For gdFlags, you should use SetDeviceAttribute to set the noDriver bit. You should also set the gDevType bit to 1 if you're using two bits per pixel or more, but it can be left at 0 if you're using only one bit per pixel.

The other big difference from the technique shown in Technical Note #120 is that the off-screen pixel image shouldn't be allocated. Instead, just point the baseAddr field of your off-screen pixMap at your NuBus card's

For information about inverse tables, see pages 137 through 139 in the Color Manager chapter of *Inside Macintosh* Volume V.•

memory. You should also set the pmVersion field of your off-screen pixMap to be the constant baseAddr32 (equal to 4). That tells Color QuickDraw to use 32-bit addressing mode to access your off-screen buffer, and that's a requirement if your off-screen pixel image is located on a NuBus card.

When you want to draw into your off-screen pixMap, save the current port with a call to GetPort and the current GDevice with a call to GetGDevice. Then set the current port to the off-screen CGrafPort with a call to SetPort, and set the current GDevice to the off-screen GDevice with a call to SetGDevice. Now all QuickDraw commands are drawn off screen and the resulting images are in your NuBus card's memory. To switch back to drawing on screen, set the current port and GDevice back to the port and GDevice that you saved earlier. Easy!

Even with all this, there might still be a reason to use GWorlds to draw into a NuBus memory card. You might just want some quick and dirty way to get an off-screen drawing environment that uses your NuBus memory card and don't care whether it works with future system software releases or not. We'll talk about that next and also discuss the issues that you have to be careful about when you do this.

First, create a GWorld using NewGWorld as usual. If you want to, pass it a color table, or you can just pass it nil if you want it to make the default color table. For the GWorld flags, make sure you pass only the keepLocal flag. This makes sure that all the pieces of the GWorld are kept in your own heap rather than being cached into the 8•24 GC card, even when you draw into it. That way, you avoid running into any conceivable conflicts with GC QuickDraw over where the GWorld really is. There's no way to tell NewGWorld not to allocate the pixel image, so you might want to make the bounds rectangle small and then make it bigger later so that your heap isn't hit up for a lot of memory that you don't even want. Don't pass it an empty rectangle because NewGWorld just gives you a paramErr in that case. Call GetGWorldDevice to get a handle to your GWorld's GDevice and save it for later.

Now it's time to have the new GWorld use your NuBus card's memory. The baseAddr of your GWorld's pixMap is allocated as a handle, and it has to be thrown out. Call GetPixBaseAddr with a handle to your GWorld's pixMap to get a pointer to the pixel image that NewGWorld allocated for you. Call RecoverHandle with that pointer to get a handle to the pixel image, and then call DisposHandle to get rid of it. Now put the address of your NuBus board into the baseAddr of your GWorld's pixMap. Then set the pmVersion field of your GWorld's pixMap to the constant baseAddr32. That tells Color QuickDraw that the baseAddr of the pixMap is a 32-bit address and so it should switch to 32-bit addressing mode whenever it draws into your GWorld.

If you passed NewGWorld a rectangle that's smaller than you actually want, you can now set it to the real size. Set the bounds rectangle of your GWorld's pixMap and the portRect rectangle of your GWorld's CGrafPort to the rectangle that you really wanted. Also, set the visRgn of the CGrafPort and the gdRect field of your GWorld's GDevice to that same rectangle. Your GWorld is ready for use!

Now the bad news. Many of the GWorld routines assume that the baseAddr field is either a real handle or a copy of the handle's master pointer. Because the pointer in the baseAddr field isn't a master pointer, those routines can crash when they expect one. Setting the pmVersion field doesn't help in most cases; these routines just assume that the GWorld's pixel image was allocated by NewGWorld, which is a reasonable assumption. What this implies is that you can no longer call many of the GWorld routines to maintain your GWorld without a risk of crashing. When you call SetGWorld for your GWorld, you should pass it the GWorld's GDevice instead of nil (that's why we recommended that you save the GWorld's GDevice after calling NewGWorld). For safety's sake, don't call any of the following:

LockPixels
UnlockPixels
AllowPurgePixels
NoPurgePixels
GetPixelsState
SetPixelsState
UpdateGWorld
GetGWorldDevice

You can call DisposeGWorld because it won't get hung up trying to deallocate the pixel image on your NuBus card; setting your pmVersion to baseAddr32 makes this possible. Of course, since all these GWorld routines are off limits, almost all the benefits of having a GWorld at all are gone as well.

Another piece of bad news is that this doesn't take advantage of the speed benefits of using GWorlds with an 8•24 GC card. Most of the speed benefit of using GWorlds with GC QuickDraw is that the GWorld's pixel image is allocated on the 8•24 GC card itself, and so the image data doesn't have to take the time to move across NuBus. If your GWorld draws into a NuBus memory card, the image data has to be moved across NuBus, and so that speed benefit is gone.

The last bit of bad news is that even if you follow all of this, you're still not guaranteed that it will still work in future system software or future video card releases. As we said earlier, this should only be done if you don't care whether it works on future system software releases or not. The description above breaks a lot of rules: don't assume that the pixel image is allocated as a handle; don't set the baseAddr of a GWorld; don't change the dimensions of a GWorld without UpdateGWorld; and don't set the pmVersion field of a GWorld.

You have your choices when you want to use QuickDraw to draw off screen into the memory of a NuBus video card. You can be safe for future compatibility by creating your own off-screen drawing environment from scratch, or you can modify a GWorld so that it uses your NuBus card's memory at the risk of breaking on future systems and at the cost of losing most of the benefits of GWorlds. If you choose the first method and you have no existing routines to create off-screen drawing environments, it's worth it to take a look at Skippy White's Famous High-Level Off-Screen Map Routines in DTS Sample Code #15 on the *Developer CD Series* disc. You can see these routines in action in DTS Sample Code #16. These routines are GWorld-like to some extent, except this time you have the great benefit of source code!

REFERENCES

- "About 32-Bit Addressing," Konstantin Othmer, *develop* Issue 6, Spring 1991, pp. 36-37.
- "Deaccelerated _CopyBits & 8•24 GC QuickDraw," Guillermo Ortiz, Macintosh Technical Note #289, January 1991.
- "Drawing Into an Off-Screen Pixel Map," Jim Friedlander, Rick Blair, and Rich Collyer, Macintosh Technical Note #120, April 1989.
- *Inside Macintosh* Volume VI, Graphics Devices Manager chapter, Addison-Wesley, 1991.
- *Inside Macintosh* Volume V, Color Manager chapter, Addison-Wesley, 1988.

Thanks to Guillermo Ortiz for reviewing this column.•

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).•

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