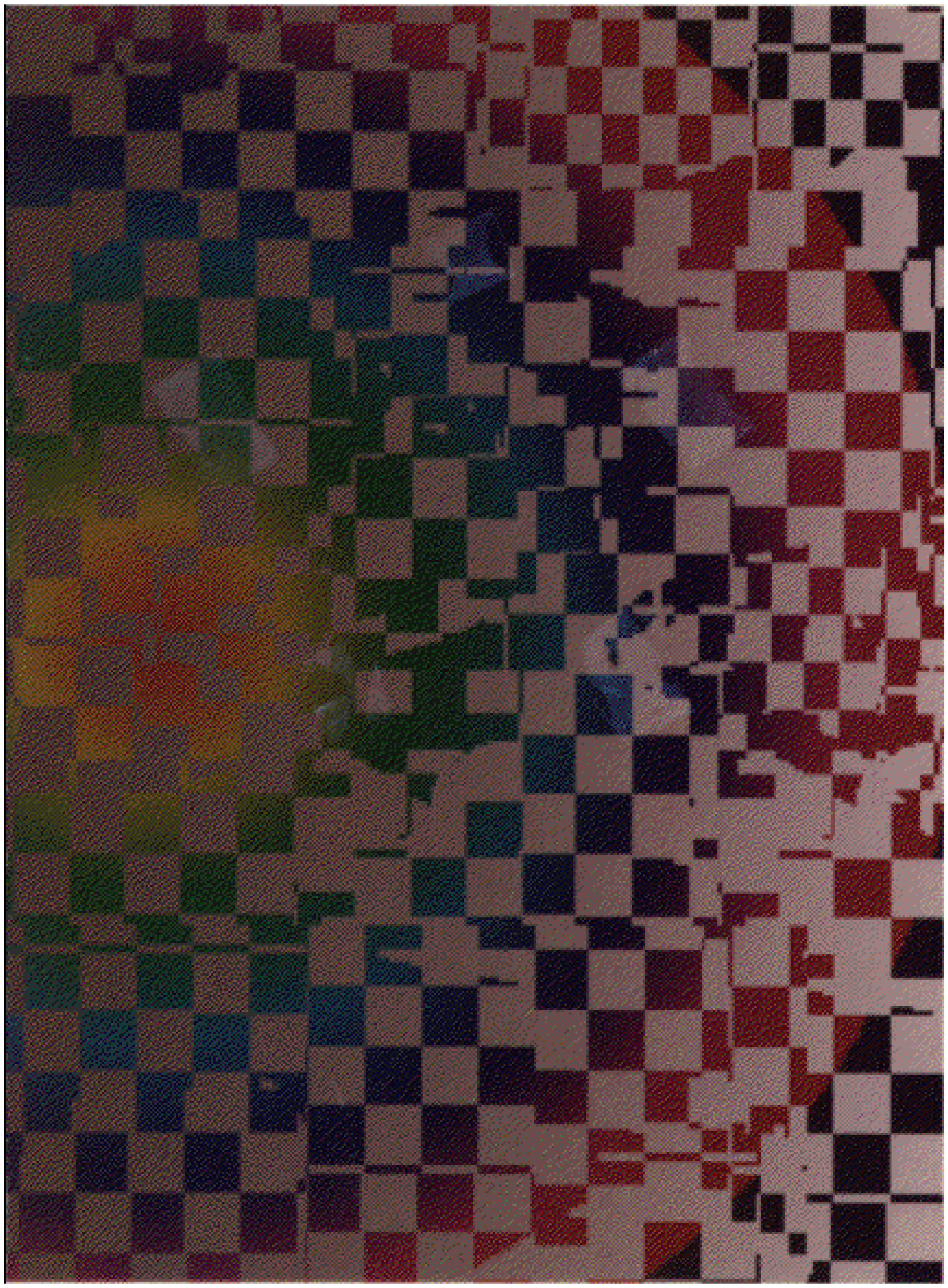
develop

The Apple Technical Journal



COLOR SUITE:

REALISTIC COLOR FOR REAL-WORLD APPLICATIONS

ALL ABOUT THE PALETTE MANAGER

BRAVING ONSCREEN GWORLDS

ALSO:

THE PERILS OF POSTSCRIPT

COMPATIBILITY: RULES OF THE ROAD

DEBUGGING DECLARATION ROMS

APPLE II DEVELOPMENT DYNAMO

Premier Issue January 1990

RESOURCES

Apple provides a wealth of information, products, and services to assist developers. The Apple Programmers and Developers Association (APDA) provides access to development tools to anyone who wants them. Qualified commercial and non-commercial developers may gain access to additional information and services through the Apple Partners and Associates programs. These programs are administered by Apple's Developer Programs organization. In addition to automatically becoming APDA subscribers, Partners and Associates also have access to the information and services provided by Apple's Developer Services organization.

APDA

APDA is an Apple-operated worldwide mail-order distribution service for developers. Serving as the sole source for non-retail development products created by Apple, APDA also offers an extensive selection of key third-party development tools, languages, and technical books.

All Apple computer users worldwide are eligible to participate in APDA. In addition to commercial developers, APDA's customers include in-house corporate developers, university professors and students, value-added resellers, and hobbyists. You do not need to be an Apple Partner or Associate to participate in APDA. **Contact** APDA at (800) 282-2732 (U.S.), or (408) 562-3910, or 20525 Mariani Ave., M/S 33G, Cupertino, CA 95014.

Apple Associates Program

The Apple Associates Program is a program designed to assist noncommercial developers by providing them with technical information and resources on a regular basis.

Apple Partners Program

The Apple Partners Program is Apple's major developer support program. This program provides access—for U.S.-based developers—to marketing and technical information, to answers to development questions, and to the annual Worldwide Developers' Conference. Apple Partners can also purchase, directly from Apple at special prices, a limited number of Apple systems for development purposes. **Contact** Developer Programs at (408) 974-4897; or 20525 Mariani Ave, M/S 75-2C, Cupertino, CA, 95014 for information or an application kit for either the Associates or the Partners program. Non-U.S. developers should contact the Apple office in their country for information about the developer programs they offer.

Apple Developer University

Apple Developer University is committed to teaching you skills that will help you create superior software products—whether you've just started programming in the Macintosh environment or whether you're an expert. Our hands-on teaching approach offers you the most direct and responsive means of acquiring up-to-date development skills. And you can be confident that the programming languages, tools, and platforms taught at Apple Developer University are chosen because they are integral to Apple's direction for the future. **Contact** the registrar at (408) 974-6215 to reserve your place or request a current catalog. You can also AppleLink Developer University at DEVUNIV.

CONTENTS

E D I T O R I A L

Editor in chief's clothing *Louella Pizzuti*

Developmental Editor *Carol Westberg*

Editorial Assistant *Susan Marsland*

Code Tester *Charlie Springer*

Manager, Dev. Press *David Krathwohl*

A R T & P R O D U C T I O N

Designer *Joss Parsey*

Technical Illustrator *Cleo Huggins*

Production *Creative Communications Center*

Printer *Craftsman Press*

Circulation Management *Dee Kiamy*

Now that you're holding the first issue of develop, Apple's quarterly technical journal, you may want to know how and why it got to you. As a company, Apple tries to support third-party developers as much as possible: in addition to the tools and system software our engineers write to make your lives easier, we've got an entire department (the Apple Developer Group) of more than 300 people whose jobs consist of trying to help you be successful. We do that by lobbying for software and hardware changes that will make your development job easier and more productive, and by publishing marketing and technical information. Enter develop.

This journal is intended to lead you into other reference materials like *Inside Macintosh* and the *Apple IIGS Reference Manual*; it doesn't try to replace or reword these books, it complements them and helps you figure out which sections you may need to study more carefully. Each article includes the author's photo and a biography. This should help you understand the minds behind the madness (as well as help you spot these folks at trade shows).

I like to think of develop as very heavily commented sample code. The text that surrounds the code explains and clarifies what the code does and why. We want you to understand and to use the code freely, so we've included a CD that contains the entire journal as well as applications built from the code in the articles. This should make it easier for you to find what you're looking for, and for you to copy the code you'd like to use. Each quarter we'll include the past issues and code, so if you've got one develop CD, you've got them all (as long as it's the most recent one).

Because we want the code to demonstrate what the text explains, you won't see the latest breaking news here. What you will see is code that solves real problems in ways that we'll strive to keep compatible in the future. All of the questions and answers (and many of the articles) come from the Apple Developer Technical Support group, so you know they're more than just theoretical exercises. They're questions or problems that real people have struggled with; hopefully publishing them here will help you solve your own problems before you lose too much sleep.

TECHNICAL REVIEWERS

Pete Alexander
Brian Bechtel
Rick Blair
Rob Dearborn
Matt Deatherage
Chris Derossi
Tim Enwell
Dave Fung
Naresh Gupta
John Harvey
Dave McGary
Jean-Charles Mourey
Andy Shebanow
Jim Straus
Larry Rosenstein
Dave Weiner
Gregg Williams
Jon Zap

SPIRIT GUIDES

Margery Cantor
Lou Tomafsky

Just as the programming problems you face are many and varied, so too are the articles in this issue. Our color suite includes Bruce Leak talking about the changes he made to 32-Bit QuickDraw, Dave van Brink discussing the new and improved Palette Manager, and Guillermo Ortiz saving you time (and compatibility headaches) by explaining the new offscreen calls. Our other articles run the gamut from Dave Radcliffe's compilation of compatibility strategies, to Eric Soldan describing his new 8-bit development system, to Mark Baumwell explaining exactly how to build and debug a declaration ROM, and finally to Scott "Zz" Zimmerman giving the scoop on how to mix PostScript and QuickDraw for a happy tomorrow.

If you're a Certified developer, Partner or Associate, you'll be getting develop every quarter as a part of your developer package. If you are not in one of the above programs, you can subscribe to develop using the envelope and order form in the back of the journal.

If you like develop, I'd love to hear about it, and if you don't like it, I'd like to change it. So let me know what you think. What you like, what you hate, and what you'd like to see more of. I'll listen and respond, and together we can make develop evolve into the journal we've all been waiting for.

[illegible]

Louella Pizzuti

Editor

COMMENTS

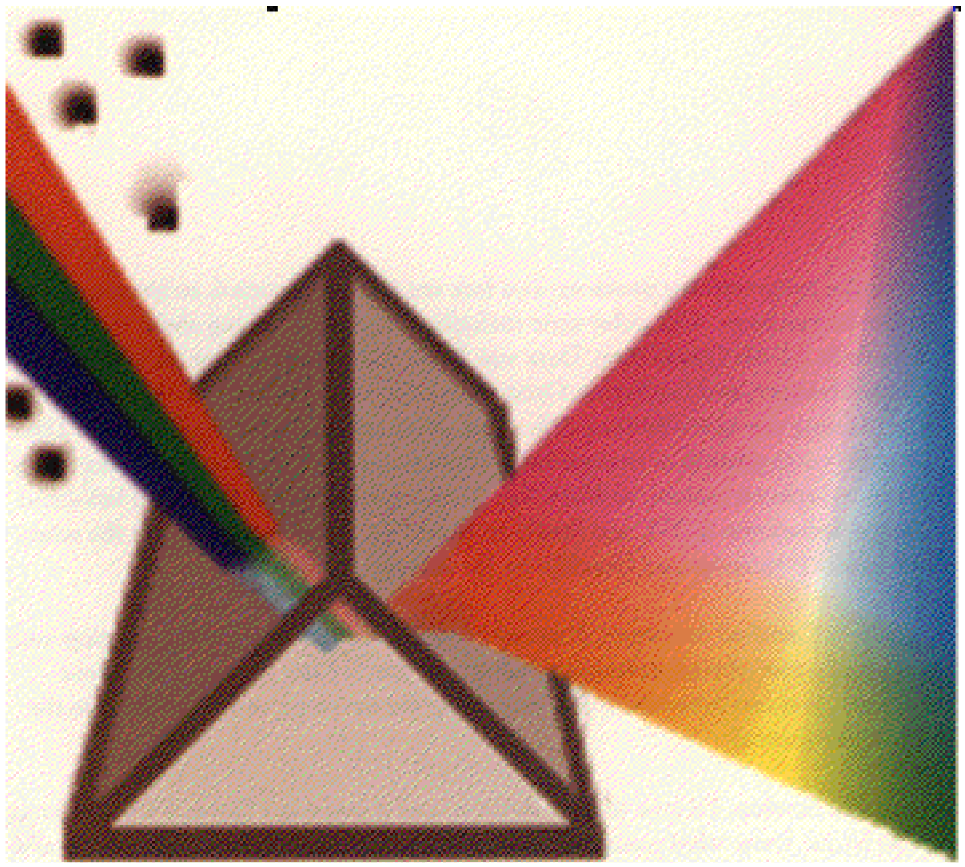
Send all letters and comments about develop directly to the editor:
Louella Pizzuti
20525 Mariani Ave M/S 75-3B
Cupertino, CA 95014   U.S.A.
AppleLink: Pizzuti1

SUBSCRIPTION INFO

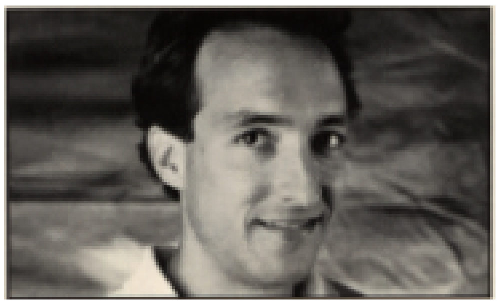
Send all subscription enquiries to:

develop
Apple Computer, Inc.
P.O. box 3721
Escondido, CA 92025 U.S.A.

REALISTIC COLOR FOR REAL-WORLD APPLICATIONS



32-Bit QuickDraw, the new extension to the Macintosh graphics system software, enables the manipulation of 16- and 32-bit color data. This article gives details on the new offscreen pixMap support, the improved Palette Manager, color to grayscale conversion through luminosity mapping, and advanced dithering from 16 and 32 bits per pixel to lower bit depths.



BRUCE LEAK, our local pixel dealer, is one of our famous convertible-driving, shorts-wearing maniacal-laughing system software engineers. He has a MSEE from Stanford University and trained for this job at Sandcastles and Microsoft. Now he spends his nights and weekends changing how you see the world on your Macintosh (32-Bit QuickDraw is his

On the Macintosh, Apple's graphics software (QD) lets a programmer work with a high-level graphics model that is independent of the physical display device. This gives Apple the option to take advantage of new features and technologies without requiring application developers to rewrite their code.

32-Bit QuickDraw extends QuickDraw to encompass the full range of displayable color. Although today's monitors are typically capable of showing a full spectrum of color or grays, it is the video card and system software of a computer that control the number of colors or shades of gray on the screen. A 1-bit video card allows 2 colors or shades (typically black and white), 2 bits gives 4, 4 bits gives 16, 8 bits gives 256, and 24 bits gives over 16 million colors.

Color QuickDraw always supported the description of nearly unlimited colors (2^{48}) but constrained images and screens to any 256 of the expressible colors. 32-Bit QuickDraw significantly increases the standard number of colors available for an application while maintaining speed and affordability.

32-Bit QuickDraw—which carries 24 bits of color information—consists of three files:

- a General `cdev` that works more effectively with the system, whether the other pieces are installed or not
- a Monitors `cdev` extended for 32-bit addressed video cards .
- a 32-Bit QuickDraw Init file, which contains the new QuickDraw software

NEW FUNCTIONALITY IN 32-BIT QUICKDRAW

This is the laundry list of 32-Bit QuickDraw features. They are listed in relative order of importance, but some may be more important to you, depending on whether you've thought about them before.

Support for 32-bit-per-pixel graphics The pixel data has 8 bits each in red, green, and blue and an 8-bit alpha channel. Only 24 of the 32 bits per pixel are used by QuickDraw. 24 bits per pixel—over 16 million colors—gives you pretty much the maximum number of colors the eye can distinguish. When the display mode of the graphics system supports this much color, you don't have to restrict the application or user to a particular set or palette.

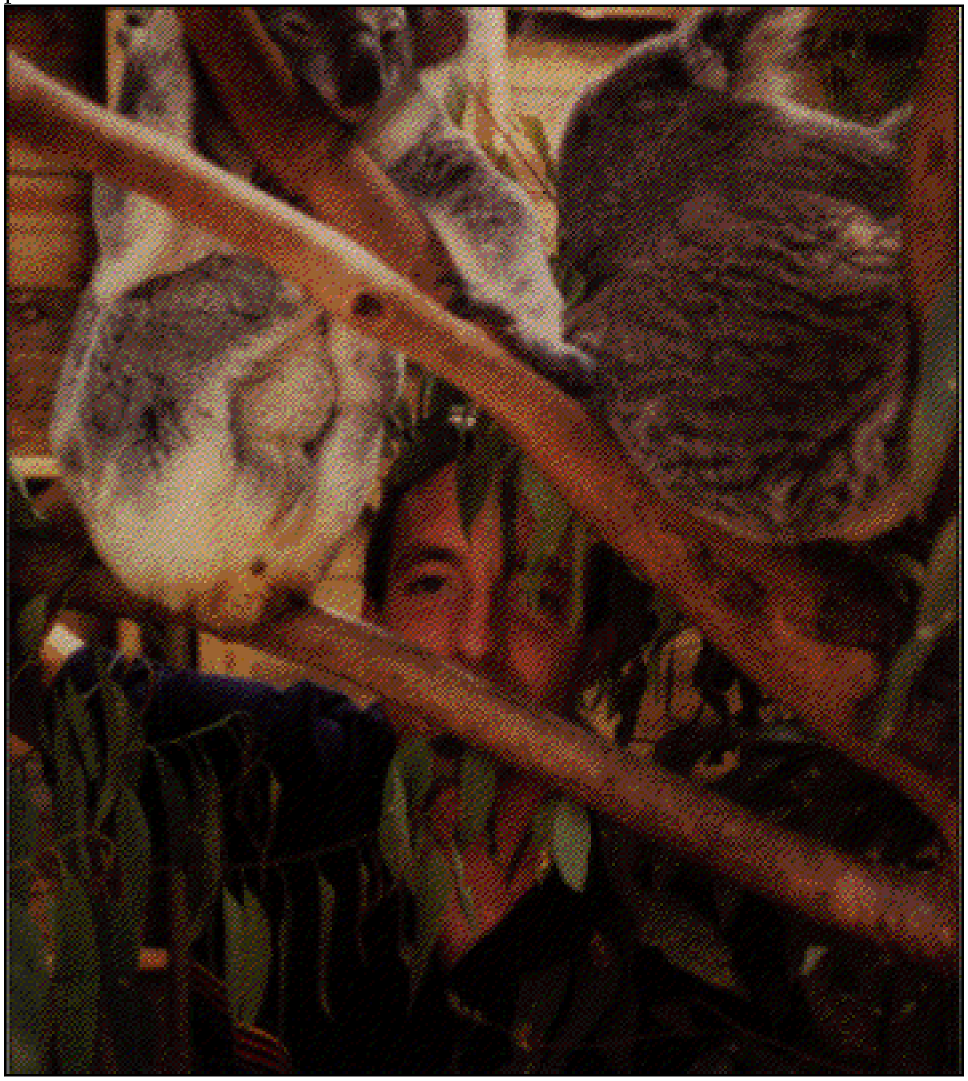
Support for 16-bit-per-pixel graphics (1 alpha-5-5-5) The pixel data has 5 bits each in red, green, and blue and a 1-bit alpha channel. 16 bits per pixel—or over 32,000 colors—is considered by many to be sufficient for high-quality graphics. Because 16 bits per pixel requires less memory than 32 bits per pixel, displays can use less expensive hardware.

specialty), and his days on the soccer field. If you have any questions (about anything) feel free tot call him tat home. tHe's tat t(408) 767-1739. •

At the time of this writing, 32-Bit QuickDraw was installable as a set of files that could be dragged into the latest system folder. 32-Bit QuickDraw will be integrated into Apple's System 7.0. •

Dithering of 32- and 16-bit images to lower color resolutions—1, 2, 4, and 8 Color dithering is similar to creating a color that looks like orange by placing red and yellow pixels close together. If the display is limited to a certain number of colors, the Macintosh can take images rich in color information and produce a close match at lower color resolutions, giving the appearance of more color by combining the limited number of colors in an effective manner.

Gray-level representation with luminosity In grayscale mode, QuickDraw will map a color to its nearest luminance value or lightness. This produces superior images, even with 16 gray levels. It's hard to imagine how good this can be without seeing it, so we'll show you just how great it is in these next two pictures:



32-Bit QuickDraw takes an additional 100K of RAM, so we recommend you use it with at least two MB of memory. 32-Bit QuickDraw also requires Color QuickDraw, which is present on the SE/30 or Macintosh II family computers. •

An improved Palette Manager The Palette Manager now allows application developers to have more control over their color environment by improving multiple monitors support, supporting a highlight color as one of four colors in 2-bit mode, setting up a true grayscale look-up table on monochrome devices, and restoring the color environment when an application quits. The new Palette Manager also directly supports color table animation of pixel images in a device-independent manner (see “All About the Palette Manager” in this issue for details).

New offscreen support These new routines can isolate developers from device dependencies by extending offscreen pixMap support. Developers typically use offscreen support to buffer drawing for fast, seamless updates to the screen. Offscreen imaging is also convenient for custom rendering algorithms that assume a particular frame buffer configuration—since requiring a particular video display mode is not a user-friendly solution

Improved graphics performance in all bit modes 32-Bit QuickDraw provides performance improvements in region-clipped pattern fills and bit blits, such as updating the desktop pattern or resizing a window.

Improved rescaling of images to smaller sizes When 32-bit-per-pixel images are resized to smaller sizes, pixels are combined using an averaging technique to yield recognizable thumbnail-sized images—an important improvement for the postage stamp market.

New routine to get regions from bitmaps Getting a bitmap from a region is easy—just paint the region. The new routine `BitmapToRgn` provides the inverse transformation.

Alpha channel movement 32-Bit QuickDraw supports routines that use up to 24 bits of color. The additional 8 bits of information are typically used by application developers as an alpha channel or transparency mask. QuickDraw now moves the extra 8 bits around without adding functionality. Note that alpha channel memory may not actually be present on the video card.

The new Monitors cdev The new `cdev` improves the Monitors interface and adds advanced features. Now developers can easily modify the `cdev` through a standard programming interface.

Support for color PostScript printing Apple's new LaserWriter 6.0 driver now offers a standard mechanism for printing the high-quality color that 32-Bit QuickDraw provides.

Color Picker changes Most of the changes to the Color Picker are related to the user interface. In the former version of the Color Picker, if the `where` parameter was set to (0, 0), then the dialog was centered on the main device. For obvious reasons, this method did not work well if the main device was not a color device. In the new Color Picker, if you want the dialog centered on the best device—that is, the device with the highest bit depth, regardless of color support—then you must pass (–1, –1) as the `where` parameter.

Compression for file formats 32-bit and 16-bit data are normally compressed when they are in PICT files so that storage on disk takes significantly less space than the original data.

Support for very large frame buffers Video cards that require more than 1 MB of memory, such as a 1280 x 1024 x 8-bit deep card, now work in a standard way with the system.

Changes in rowbyte restrictions 32-Bit QuickDraw relaxes the restriction that rowbytes be less than `$2000.` However, the new limit of `$3FFE` may still be a problem with large pixel maps at 32 bits per pixel. Early releases of the 32-Bit QuickDraw documentation incorrectly reported that rowbytes had a limit of `$8000`.

Not to mention, as you always expected... 32-Bit QuickDraw has implications for several familiar aspects of working with the Macintosh. PICT, the Macintosh graphics resource and file format, has been extended to support 16-bit and 32-bit data. The transfer modes that allow QuickDraw to blend different bitmaps and combine their colors interactively in different ways have been extended so that they work at, and between, all bit depths. Because images of any depth can be displayed at any depth, you have ultimate flexibility to respond to the requirements of the task at hand. Colors are mapped to an appropriate match when increasing or decreasing color depth of the display.

WHAT YOU CAN DO WITH IT

To begin with, you can't use 32-Bit QuickDraw features if you don't know they're installed. To check for 32-Bit QuickDraw, your application should ensure that Color QuickDraw is present on the machine by calling `SysEnvirons`. Then, since 32-Bit QuickDraw internally uses trap number `$AB03,` check to see if this trap is available by comparing its trap address with that of the standard unimplemented trap `$A89F`. If the two are the same, trap `$AB03` is unavailable, and 32-Bit QuickDraw is not present.

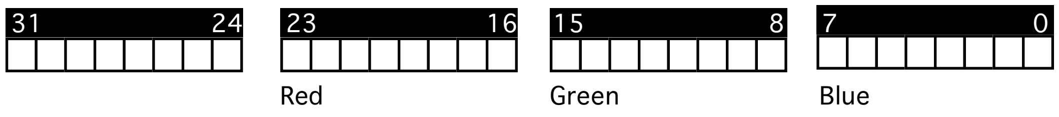
You can use the following MPW C code fragment to test for Color QuickDraw and 32-Bit QuickDraw. If either test fails, tell the user.

```
#define QD32Trap         0xAB03
#define UnImplTrap       0xA89F
#define False 0
#define True 1
PutUpInformativeMessage()
{
    printf("\n 32-Bit QuickDraw is not implemented.");
}
```

```
QD32Exists()
{
    short error;
    Boolean result = False;    /*  Assume not there  */
    SysEnvRec theWorld;

    error = SysEnvirons (2, &theWorld);

    if (theWorld.hasColorQD)
    result = (NGetTrapAddress (QD32Trap, ToolTrap) !=
           NGetTrapAddress (UnImplTrap, ToolTrap));

    return result;
}

main()
{
    Boolean QD32IsImplemented;

    QD32IsImplemented = QD32Exists();
    if (!QD32IsImplemented)
        PutUpInformativeMessage();
}
```

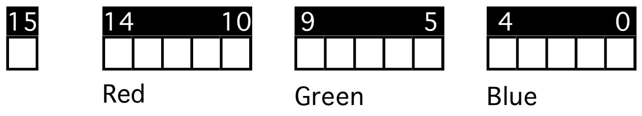
Direct pixMaps 32-Bit QuickDraw supports two new pixel formats, corresponding to 32-bit pixels and 16-bit pixels. In each case, the pixel's color is specified by the pixel value directly. The pixel value is not an index into a color look-up table. In a `pixMap`, this is specified by setting the `pixelType` field to `RGBDirect = 16.`

Before 32-Bit QuickDraw, when each pixel was a single value representing an index into a color table, the cmpCount field was always equal to 1. With `RGBDirect` pixels, each pixel contains three components, one each for the intensities of red, green, and blue; therefore, `cmpCount` should be set to 3.



```
pixelType     =     16;    {RGBDirect}
pixelSize     =     32;    {Must be a power of 2}
cmpCount      =     3;     {Three components: Red, Green, Blue}
cmpSize       =     8;     {8 bits for each component}
pmVersion     =     0;     {Must be set for future compatibility}
pmTable                    {Handle to color table with 1 entry}
```

```
        ctSeed =       24;     {CmpCount * CmpSize}
        ctFlags        =       0;      {No special flags}
        ctSize =       0;      {Zero based count of entries}
        ctTable        =               {Space for one color spec}
```



```
pixelType      =       16;     {RGBDirect}
pixelSize      =       16;     {Must be a power of 2}
cmpCount       =       3;      {Red, Green, Blue
cmpSize        =       5;      {5 bits for each component}
pmVersion      =       0;      {Must be set for future compatibility}
pmTable                        {Handle to color table with 1 entry}
        ctSeed =       15;     {CmpCount * CmpSize}
        ctFlags        =       0;      {No special flags}
        ctSize =       0;      {Zero based count of entries}
```

Since the Window Manager, as well as many applications, examines the ctSeed of screen `pixMaps`, the `pmTable` field for a direct device pixel map should always contain a valid handle with a color table header. For consistency, the ctSeed should be equal to `cmpCount * cmpSize`—although you can have the seed equal a unique value. If the seed value is the same across devices, then window moves will copy an image with `CopyBits`; otherwise, the Window Manager will generate an update event. The `ctFlags` and `ctSize` values should be set to zero.

PixPats In addition to the standard 8 x 8 foreground/background QuickDraw pattern filling we have all come to know and love, Color QuickDraw supports drawing objects with tiled pixel images or `pixPats` (short for pixel patterns). Although `pixPats` actually contain a `pixMap` that is capable of describing a pixel image of any size and depth, only bit depths 1 through 8 and dimensions that are a power of two are supported. Since `pixPats` contain color information, the `grafPort's` foreground and background colors are ignored when drawing with pixel patterns. Future versions of Color QuickDraw will support 16- and 32-bit pixel patterns.

Color QuickDraw expands a `pixPat` to the depth of the target device before drawing. Make sure that enough memory is available for the expanded pattern. The current 32-Bit QuickDraw limits the volume (area times depth) of an expanded `pixPat` to less than 64K. Hence a 128 x 128 `pixPat` (16K @ 8 bits deep) will work at 8 bits per pixel but draw incorrectly when rendered at 32 bits per pixel. A 64 x 128 `pixPat` (32K @ 32 bits deep) will work at any depth.

Another type of `pixPat`, the RGB pattern, is completely described by a single RGB Color. When drawing an object with an RGB pattern, Color QuickDraw computes an ordered dither matrix that most accurately represents the RGB Color on each screen it intersects. Since dithering is not performed on 32-bit-per-pixel devices, there is never any penalty for using `makeRGBPat` to approximate a desired color.

Drawing in color When an application requests the drawing of one of the 2^{48} expressible colors, Color QuickDraw finds the closest or best color available on `TheGDevice`. In general, this is done by the Color Manager routine `Color2Index()`, more appropriately called `Color2Pixel`. On indexed display devices, `Color2Index()` uses the device's inverse table to find the best match. For direct devices, `Color2Index()` truncates each color component to the cmpSize of the device. When the destination device's color table only contains shades of gray, `Color2Index()` matches the luminance of the requested color to the closest gray on the device. Luminance mapping gives superior-looking images on grayscale displays.

While most of Color QuickDraw will find the best color regardless of inverse table resolution, `CopyBits` from a direct `pixMap` to an indexed device cannot distinguish between two colors that do not differ in the high `itabRes` bits of any component. Fortunately, the default `itabRes` is four and the standard 8-bit color table does not contain any colors that are not differentiated by the high four bits of each component. Future versions of 32-Bit QuickDraw may be able to find these "hidden colors" (for more information on hidden colors, see *Inside Macintosh*, volume V, page 138).

32-bit addressing In 24-bit addressing mode, there are 16 MB of address space, 6 of which are reserved for NuBus slots at only 1 MB per slot. Since most 16- and 32-bit-per-pixel video cards, as well as some very large 8-bit ones, require more than the 1 MB of address space per slot, the CPU must access these cards in 32-bit addressing mode. 32-Bit QuickDraw performs all drawing operations in 32-bit addressing mode. The base address of a screen is assumed to be a valid 32-bit address, while all other pixMap base addresses are treated as 24-bit addresses. Currently, 32-Bit QuickDraw cannot determine whether a nonscreen base address is a valid 32-bit address. When setting up a `pixMap`, make sure to initialize the `pmVersion` field to zero and use `StripAddress` on dereferenced handles installed as bitmap or `pixMap` base addresses.

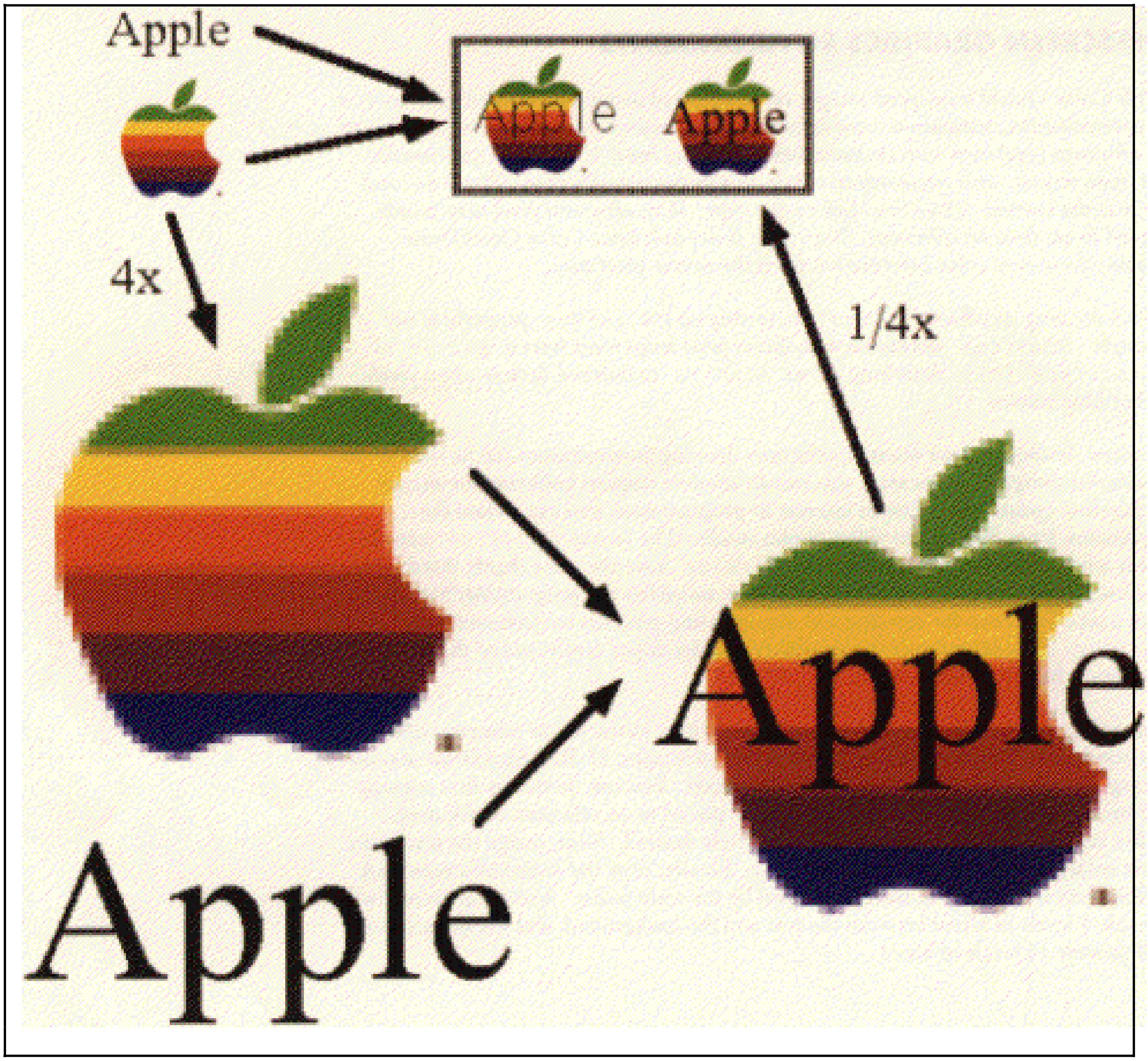
OFFSCREEN GRAPHICS ENVIRONMENTS

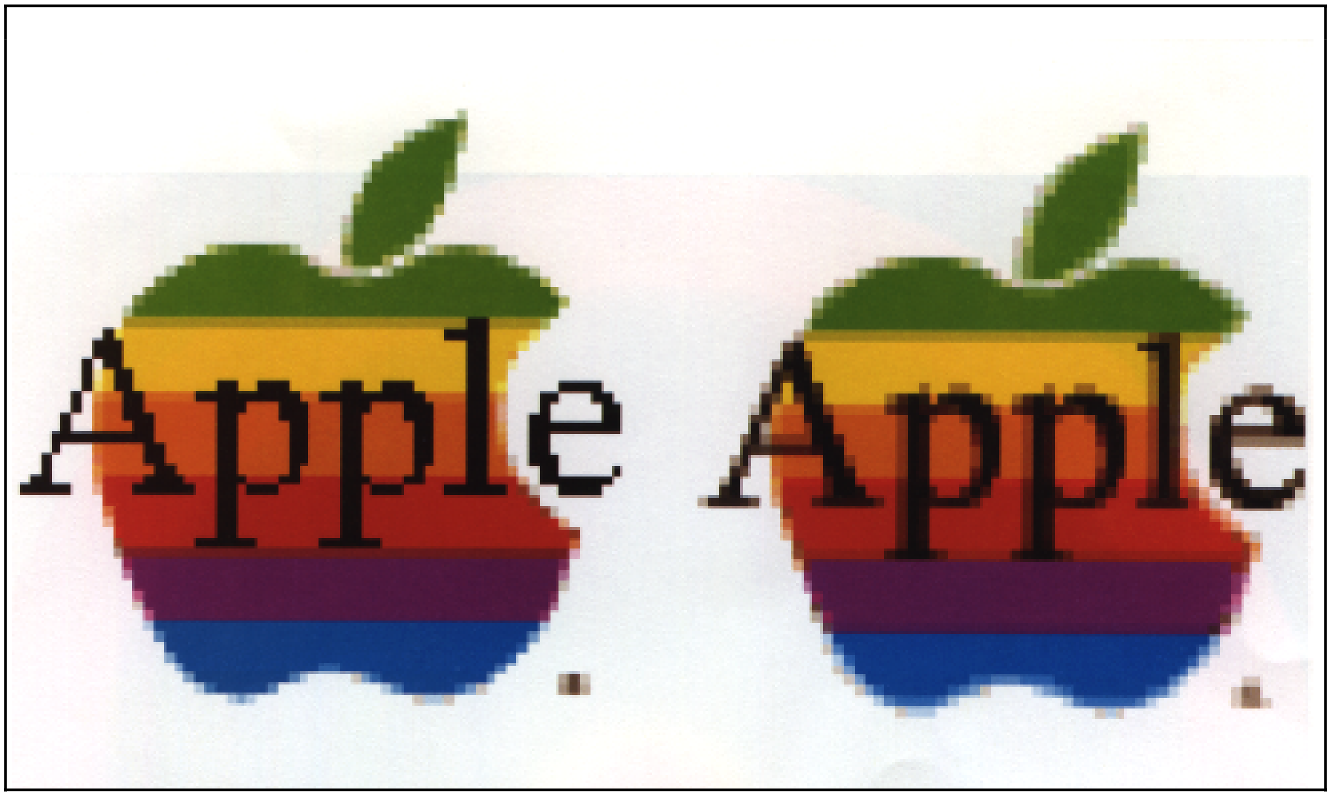
With Color QuickDraw, pixel images are transferred using data from `TheGDevice` to determine the destination color information. Consequently, whenever copying to an offscreen pixel map with characteristics differing from `TheGDevice`—usually the main screen—it is necessary to create an appropriate offscreen GDevice and set it as the current `GDevice` before the copy. If an offscreen pixel map is only copied from, then no offscreen `GDevice` is needed, since Color QuickDraw obtains the source color information from the source pixel map.

When creating an offscreen `GDevice`, setting up the gdPMap properly is not enough. `GDevices` associated with direct pixel maps must have a gdType of `directType (=2).` Attaching direct `pixMaps` to indexed devices often yields rather blue results.

As many developers have learned, offscreen drawing environments can be used to do wonderful things. For instance, you can do window content buffering for snappy flicker-free updates, or, of more interest to programmers, you can isolate the application from the current video display mode. The fastest `CopyBits` transfers occur when the source and destination `pixMaps` have the same depth, color table, and long word alignment. 32-Bit QuickDraw simplifies the programmer's model with a set of routines for creating and manipulating graphics environments or `GWorlds`. To ensure future compatibility and developer sanity, use of the new routines is highly recommended.

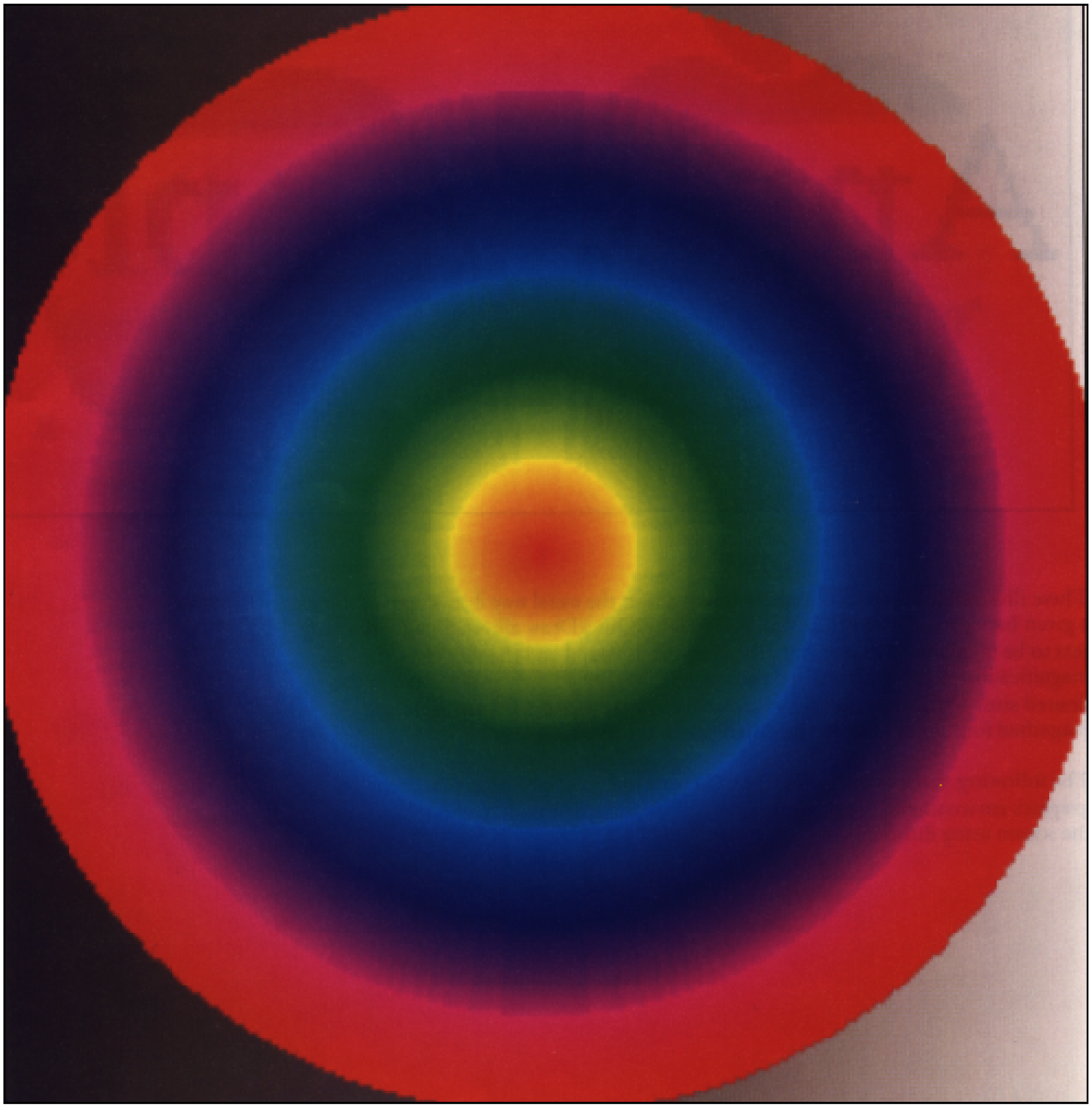
Using the new routines, text can be antialiasedwith a clever use of offscreen drawing environments. When shrinking 32-bit-per-pixel images, 32-Bit QuickDraw uses an averaging technique that can yield antialiased text. You can do this by first copying the background image where the text is to be placed to an offscreen 32-bit deep buffer that is 2 or 4 times bigger than ultimately desired. Next, image the text at the same enlargement into the offscreen buffer. Finally, copy the entire offscreen back to the desired destination, shrinking down by the scale factor. A scale factor of 2 will provide 4 levels of blend between the text and the background, and a scale factor of 4 will provide 16 levels of blend.


Apple
4x
1/4x
Apple
Apple
Apple
Apple
Apple



These illustrations show two ways of drawing text; the usual way draws text on top of a given background at the final size. Now 32-Bit QuickDraw allows for anti-aliasing text to be obtained by first drawing both the background and the text magnified (4x magnification in the example) and then using CopyBits to display the result at the desired size. The illustration above shows the contents of the framed rectangle magnified four times to show the difference in the resulting text

The following sample code in THINK C creates a 32-bit-per-pixel offscreen graphics environment, draws an exciting bull's eye image into it, and displays it on the screen using dithering.



```
/*************************************************************
*
*   Better Bull’s eye
*   Code fragments for creating and drawing to a 32-bit-per-pixel
*   offscreen graphics world.
*                                       Written in THINK C.
*                                       DVB 8-8-89
*
*************************************************************/
static Rect dOffBounds = {0,0,256,256};
static GWorldPtr gMyOffG;
MakeMyOffscreen()
  /*
  * Create a 32-bit offscreen GWorld with a gray ramp
  * and some stylish concentric circles.
  */
{
    GDHandle oldGD;
    GWorldPtr oldGW;
    HSVColor hsvc;
    RGBColor rgbc;
    long x;
    Rect r;
    GetGWorld(&oldGW,&oldGD);                          /*  Save the current graphics state */
    if NewGWorld(&gMyOffG,32,&dOffBounds,nil,nil,0)
    /*  Was it successful?  */
        PutUpErrorMessageAndExit();
        /*  Just bail; Could try a smaller one  */
    LockPixels(gMyOffG->portPixMap);
    /*  Must lock ’em before drawing there  */
    SetGWorld(gMyOffG,nil);
    /*  Start drawing here  */
    for(x = 0; x<dOffBounds.right; x++)
    /*  Do a gray ramp from left to right  */
        {
        rgbc.red = rgbc.green = rgbc.blue = x * 65535 / (dOffBounds.right - 1);
        RGBForeColor(&rgbc);
        MoveTo(x,0);
        LineTo(x,dOffBounds.bottom);
        }
r = dOffBounds;
    /*  Copy the full bounds rectangle  */
    hsvc.value = 65535;
    /*  Value and Saturation at full:  */
    hsvc.saturation = 65535;
    /*  We'll use bright colors only  */
    for (x = dOffBounds.right/2; x; x--)
    /*  Draw a series of concentric ovals  */
        {
        hsvc.hue = x * 131070/(dOffBounds.right - 1);
        /*  Step the hue as we get smaller  */
        HSV2RGB(&hsvc,&rgbc);
        /*  Get an RGB color  */
        RGBForeColor(&rgbc);
        /*  Set that as the foreground color  */
        FrameOval(&r);
        /*  Draw the oval  */
        InsetRect(&r,1,1);
        /*  Step down to the next oval  */
        }
    SetGWorld(oldGW,oldGD);
    /*  Go back to old graphics state  */
    UnLockPixels(gMyOffG->portPixMap);
    /*  Let 'em float around for a while  */
    }
  /*
  * Update the current grafport (presumably a window)
  * with the contents of the gMyOffG GWorld.
  */
UpdateMyWindow()
    {
    LockPixels(gMyOffG->portPixMap);
    /*  Must lock 'em before drawing to it  */
    /*  Fit it to the window with dithering  */
    CopyBits(&gMyOffG->portPixMap, &thePort->portBits, &dOffBounds,
    &thePort->portRect, ditherCopy,  0);
    UnLockPixels(gMyOffG->portPixMap);
    /*  Let 'em float around for a while  */
    }
```

ADVANTAGES OF BITMAPTOREGION

`BitmapToRegion` lets you use most of QuickDraw's region-oriented calls on bitmaps by converting the bitmaps into regions. Though this call was previously available through Software Licensing, `BitmapToRegion` was added to the 32-Bit QuickDraw package in the interest of seeing it more widely used. This call is particularly good for converting bitmaps to regions when you need to clip to a bitmap or drag its outline. One application is using a bitmap to mask a color image and apply a transfer mode. This allows you to call the more powerful `CopyBits` with a region clip instead of `CopyMask` with a bitmap clip. A `CopyMask` operation, for example, would not be recorded into a picture and does not support transfer modes.

An example of a call done with `CopyMask` would be:

```
CopyMask (srcPixMap, maskBitmap, destPixMap, srcRect,
maskRect, destRect)
```

Instead, you could use:

```
BitmapToRegion (maskRegion,maskBitmap)

/*  Region must have been created previously with NewRgn  */
CopyBits (srcPixMap, destPixMap,srcRect,destRect,mode,maskRegion)
```

Another use for `BitmapToRegion` would be in creating a patterned paint bucket fill for a bitmap.

```
BitmapToRegion (maskRegion,myBitmap);
PenPat (myPattern);  /*  or PenPixPat (mypixPat)  */
PaintRegion(maskRegion);
```

Alternatively, to change the color of a bitmap, you could use:

```
BitmapToRegion (maskRegion,myBitmap);
RGBForeColor (mycolor);
PaintRegion(maskRegion);
```

You could drag the outline of a bitmap around by calling:

```
BitmapToRegion (maskRegion,mybitmap);
DragGrayRegion (maskRegion, startPt, etc...);
```

Finally, you could test a mouse point, or whatever, for intersection with a bitmap with:

```
BitmapToRegion (maskRegion,mybitmap);
PtInRgn (pt,maskRegion);
```

THE 72 DPI PIXMAP BARRIER

Actually, there never was a 72 dpi `pixMap` barrier. Rather, the proper usage of `pixMap` resolution has not been well described. In the past, applications have accepted `pixMaps` of a given number of rows and columns and assumed that they were generated on 72 dpi devices. These `pixMaps` were then copied around at a 72 dpi resolution and printed out at a 72 dpi resolution, leaving the impression that QuickDraw could not handle `pixMaps` of different densities. The advent of frame grabbers and scanners renders this method of `pixMap` handling obsolete. Now, many `pixMaps` have a higher resolution than 72 dpi. In fact, a user expects such a `pixMap` to display an approximation of the information on a 72 dpi display, but print on a higher-resolution device to the best of its ability.

When recording pictures that contain `pixMaps`, make sure to set the `hRes` and `vRes` fields of the pixMap record to the native resolution of the image. When importing pictures, obtain `pixMap` information from the `StdBits` bottleneck procedure, not by imaging the picture into its `picFrame`.

CUSTOM COLOR SEARCH PROCEDURES

The following 32-Bit QuickDraw lore describes the May '89 release and is subject to change in future 32-Bit QuickDraw versions.

When pixel images are transferred to a different depth, the destination color information is obtained from `TheGDevice`. Custom Color Search Procedures, or Search Procs for short, associated with a `GDevice` provide a mechanism for customizing QuickDraw's color matching algorithms. When the source image is 32 or 16 bits per pixel, 32-Bit QuickDraw calls the Search Proc associated with the destination `GDevice` for each source pixel. Since 32-Bit QuickDraw always accesses direct pixel maps in 32-bit addressing mode, and since direct `pixMap` image translation is performed at draw time, don't be surprised if your custom Color Search Procedure gets called in 32-bit addressing mode. Color Search Procedures should call `StripAddress` on dereferenced handles and `SwapMMUMode` if toolbox access such as `Color2Index` is required (for more information on 32-bit addressing and `SwapMMUMode`, see *Inside Macintosh*, volume V, page 592).

Also, for direct pixel source images, the application's global pointer in register A5 may not be valid on entry to custom Color Search Procedures. If your Color Search Procedure accesses global data structures referenced from register A5 (including thePort), it must first save and later restore the A5 contents (see Technical Notes #180 and #208).

In the current version of 32-Bit QuickDraw, custom Color Search Procedures are ignored when transferring 32-bit or 16-bit images to a device of identical depth—this is subject to change in future releases. Since dithering techniques need to accurately maintain colorspace distance, 32-Bit QuickDraw refuses to dither direct `pixMaps` when a custom Search Procedure is present. A printer-driver developer should not rely on this functionality. To prevent a picture from dithering, intercept the `StdBits` bottleneck routine, and remap the transfer mode to `srcCopy`.

CopyBits error codes QuickDraw uses stack space for work buffers. For complex operations such as depth conversion, dithering, or image resizing, stack space may not be enough. In such situations, Color QuickDraw simply skips the operation. In this case, 32-Bit QuickDraw will request temporary memory from MultiFinder. If that is still not enough, or if MultiFinder is not present, 32-Bit QuickDraw returns

```
   QDErr = -149 /*  Insufficient stack  */
```

This value can be obtained by calling `QDError(),` which will reset `QDErr` to zero. One recourse for the application is to divide the operation—for example, divide the image into left and right halves—and try again.

Region creation error codes While recording drawing operations into an open region, it is possible that the resulting region description will overflow the current 64K limit. Should this happen, 32-Bit QuickDraw will return

```
   #define rgnOverflowErr -147  /*  Region overflow  */
```

Since the resulting region is potentially corrupt, `closeRgn` will return an empty region if it detects `QDErr` has been set to -147.

32-Bit QuickDraw's offscreen bitmaps, luminosity mapping, and advanced dithering automatically do work that would have required sophisticated and lengthy code from a developer. The new Palette Manager unloads much of the color management burden from your application. When all this is combined with the ability to display such broad ranges of color, we can see that 32-Bit QuickDraw represents a tremendous programming effort.

32-Bit QuickDraw may be redistributed by developers that have a standard licensing agreement with Apple. Information on the agreement is available from the Software Licensing group (AppleLink: SW.LICENSE) at Apple. Further documentation on 32-Bit QuickDraw is available from APDA. 32-Bit QuickDraw has been made available to customers through all Apple dealers, licensed user groups, and licensed bulletin boards, and will be incorporated in System 7.0. •

ALL ABOUT THE PALETTE MANAGER

One of the goals of QuickDraw is to isolate the application from the specific graphic display devices it is running on. The Palette Manager lets multiple applications share screen space and color allocation in a fair and orderly fashion.

The Palette Manager has been enhanced in 32-Bit QuickDraw to support new color usages. When applications use the Palette Manager to establish and maintain a specific color environment, the Palette Manager juggles numerous factors in honoring a request. It must consider, for example, the limits of the available display hardware and the presence of other applications requesting color environments. On a multiple-screen system, the Palette Manager will keep track of the colors for each screen. Also, the Palette Manager provides an extra level of indirection in drawing colors, which serves as a color naming or numbering system.

UNDERSTANDING THE PALETTE MANAGER

A palette is a data structure attached to a window using the `SetPalette` system call or other means. The palette, which contains a list of colors, each with a usage value and a tolerance value, lets the system know what colors that window needs. When colors are changed, the Palette Manager makes sure that windows, the menu bar, and the desktop are redrawn as needed with the new colors.

The Palette Manager calls should be used any time you might be tempted to call the Color Manager routines `SetEntries` or `RestoreEntries`. These calls modify the color environment directly, without letting the system decide which colors would be best to change. Also, they operate on a single screen. `SetEntries` or `RestoreEntries` should only be called by programs that have no intention of sharing the screen—programs like lava-lamp screen-savers and programs that will never, ever run under MultiFinder. Most commercial software does not fall in this category and absolutely should not call `SetEntries` and `RestoreEntries`. Use the Palette Manager to modify the color environment to get a better application with less work.

DAVID VAN BRINK, graphics software engineer, graduated from the University of California-Irvine in 1986 with a BSICS (you figure it out!) degree. After working in southern California for MegaGraphics, he moved north to work for Apple. Despite the fact that sleeping on the job is his professed favorite part of working here, he does seem to find time to write some awesome software. When he's not

Suppose we are writing a program to display PICTs on a four or more bit-depth screen. The built-in color table for four bits ('clut' ID 4, in ROM resources) contains a smattering of different colors. If the PICT we wish to display contains only shades of red, we'd want to have as many shades of red as possible (14 on a four-bit screen) in the screen's 'clut'. There are actually 16 different colors available, but 2 of them, black and white, are never changed. This guarantees that menus, windows, and other such things that are always black on the Macintosh will be visible in their intended colors.

To get some shades of red on the screen, we create a palette with 14 entries, each with a different shade of red in it. We set the usage of each entry to `pmTolerant`. When the palette's window is activated, the Palette Manager will look for shades of red that are within a certain range, or within tolerance, of each palette entry. If an index in the screen's 'clut' is already within range of one of the entries, then the Palette Manager will use that index. If not, the Palette Manager will steal an index in the order specified by its color arbitration rules and change it to the requested color.

Here is how we generate the sample palette:

```
   FUNCTION Make14RedPalette: PaletteHandle;
   VAR
      i:              LONGINT;
      ph:             PaletteHandle;
      c:              RGBColor;

   BEGIN
      ph := NewPalette(14,NIL,pmTolerant,4000);
(* should check for NIL result *)
      c.green := 0;
      c.blue := 0;
      FOR i:=0 TO 13
      DO
      BEGIN
(* range red component from 1/14 to 14/14 *)
(* i is a longint, and so can safely be multiplied by 65535 *)
                     c.red := (i+1)*65535 DIV 14;
                     SetEntryColor(ph,i,c);
      END;
      Make14RedPalette := ph;
   END;
```

sleeping his days away at Apple, he dabbles in computer-aided music and writes his own music sequencing software. Other activities he enjoys: recreating Philip K. Dick scenarios with friends; watching *Star Trek* while eating Kraft® Macaroni And Cheese; and encouraging his parrot named "12" to teach his rat "X" to talk. And always remember....•

Color Arbitration Color look-up table ('clut') displays, like the Macintosh II Video Card, have a certain number of colors available at each bit-depth. Different applications, and different documents within an application, use the colors in different ways. For example, a full-color digitized photograph isn't usually meant to be displayed in various tones of brown. If there aren't enough

Next, we might attach it to our window:

```
myPalette := Make14RedPalette;
SetPalette(myWindow,myPalette);
```

Whenever this window is brought to the front, the Palette Manager will attempt to provide 14 shades of red, ranging from RGB(4681,0,0) to RGB(65535,0,0). We might consider RGB(0,0,0), black, which is always available, to be a 15th shade of red. To draw in these colors, we just make the normal Color QuickDraw calls (like `RGBForeColor`), and we'll automatically get the closest shade of red available. If, after setting up this palette, we try to draw in nonred colors, the results will not be pretty. With black, white, and 14 shades of red, the available options for a good match to green, for example, are severely limited.

In the preceding example, we assumed the PICT uses shades of red. This is generally hard to determine, unless we generate the PICT in the first place. Since the PICT is in reds, we might want to load the palette from a resource with `GetNewPalette` rather than compute it in code. Or, we might happen to have a color table with the colors we need. In that case, we could have passed it to `NewPalette`, instead of nil in the example, or to `CTab2Palette`, if we already had a palette allocated.

We've just described how a program can ensure that a set of colors is available for a specific graphic display. The well-mannered programmer may be thinking, "Gee, that's swell, but how do I give those colors back when I'm done?" The answer is, don't even try. Any other window that needs colors to look good will have its own palette attached, and therefore get the colors it needs. Also, each time a program quits, the Palette Manager restores the color environment to a well-balanced state in terms of color distribution.

SELECTING THE RIGHT COLOR SET

Different types of screens often require different color sets to best display the same image. Grayscale screens default to having a range of gray tones from black to white, which is an excellent range for drawing most images. A grayscale screen should usually be left with its default color table.

Here is the sample routine modified to provide 14 shades of red on a four-bit color screen, 254 shades on an eight-bit color screen, and no color requests at all on two-bit or grayscale screens:

colors to go around, the Palette Manager arbitrates. When a window with a palette comes to the front, the Palette Manager inspects the window's palette and tries to modify the screen's color table to best satisfy the palette. (If a window without a palette comes to the front, no change occurs.) When another window is activated and comes to the front, the Palette Manager inspects its palette and modifies the colors, taking colors used by previous windows last. •

```
   FUNCTION MakeRedPalette: PaletteHandle;
   VAR
      i:     LONGINT;
      ph:    PaletteHandle;
      c:     RGBColor;

   BEGIN
      ph := NewPalette(14+254,NIL,0,0);
(* should check for NIL result *)
      c.green := 0;
      c.blue := 0;
(* Make fourteen reds that are inhibited on all *)
(* screens except four-bit color *)
      FOR i:=0 TO 13
      DO
      BEGIN
(* range red component from 1/14 to 14/14 *)
(* i is a longint, and so can safely be multiplied by 65535 *)
         c.red := (i+1)*65535 DIV 14;
         SetEntryColor(ph,i,c);
         SetEntryUsage(ph,i,pmTolerant+pmInhibitC2+

      pmInhibitG2+pmInhibitG4+

      pmInhibitC8+pmInhibitG8,4000);
      END;
(* Make 254 reds that are inhibited on all *)
(* screens except eight-bit color *)
      FOR i:=0 TO 253
      DO
      BEGIN
(* range red component from 1/254 to 254/254 *)
(* i is a longint, and so can safely be multiplied by 65535 *)
         c.red := (i+1)*65535 DIV 254;
         SetEntryColor(ph,14+i,c);
         SetEntryUsage(ph,14+i,pmTolerant+pmInhibitC2+

   pmInhibitG2+pmInhibitG4+

   pmInhibitC4+pmInhibitG8,0);
      END;
      MakeRedPalette := ph;
   END;
```

MORE WAYS OF REQUESTING COLOR

So far, we've seen how to request a set of colors from the Palette Manager. The examples use colors with usage combinations of `pmTolerant`, and various inhibit bits. Other ways to request colors are explained in *Inside Macintosh*, volume V, page 154. In addition, combinations of `pmExplicit` and `pmTolerant` or `pmAnimated` are now supported. Here are some examples of different usages.

```
SetEntryUsage(ph,1,.pmCourteous;,0);
```

The color is courteous, activating the palette will never cause a change in a screen's color table. This is useful only for "naming" the color, in this case to 1. `PmForeColor(1)` and `PmBackColor(1)` are two ways to use this color.

```
SetEntryUsage(ph,2,.i.pmTolerant;,10000);
```

The color is tolerant, activating the palette will ensure that some index in the screen's color table is within 10000 in each RGB component from palette entry 2.

```
SetEntryUsage(ph,3,.i.pmAnimated;,0);
```

The color is animated, activating the palette will reserve an index from the screen's color table to be owned by this palette (if the screen is indexed). If the color is changed with AnimateEntry then any previous drawing done with that entry will change.

```
SetEntryUsage(ph,4,.i.pmExplicit;,0);
```

The color is explicit, any drawing done with this entry will draw in device index 4, because this entry is the 4th color in the palette. This is mostly useful for monitoring the color environment.

```
SetEntryUsage(ph,5,pmExplicit+pmTolerant,0);
```

The color is both explicit and tolerant. Activating the palette will ensure that device index 5 exactly matches the palette's color 5 (because the tolerance here is 0).

```
SetEntryUsage(ph,6,pmExplicit+pmAnimated,0);
```

The color is both explicit and animated. Activating the palette will reserve screen index 6 for this palette's use. The color may be modified with `AnimateEntry` and `AnimatePalette`.

```
SetEntryUsage(ph,7,pmAnimated+pmInhibitC8+pmInhibitG8,0);
```

The color is animated on any screen other than an 8-bit color or an 8-bit gray-scale device. On those devices, the color behaves as a courteous color.

DRAWING WITH PALETTE COLORS

After the palette has been set up, there are several ways to draw with the colors in the palette. In addition to `PmForeColor` and `PmBackColor`, a `pixMap` or `pixPat` color table may be specified to point to palette entries. To do this, set bit 14 in the `ctFlags` field of the color table (`ctFlags` is called `transindex` in older equate files). Then set the desired palette entry numbers in the value field of each `colorSpec`. The color table is then assumed to be sequential, as device tables are (`colorSpec` 0 refers to pixel value 0 in the `pixMap` or `pixPat`; color value 1 refers to pixel value 1, and so on).

This code retrieves a copy of the desktop pattern (system resource `'ppat'` 16) and modifies its fields to refer to sequential palette entries.

```
myPP: PixPatHandle;
myCT: CTabHandle;
myPP := GetPixPat(16);
(* Gets the system color desktop pattern     *)
myCT := myPP^^.patMap^^.pmTable
   FOR j := 0 TO myCT^^.ctSize
   (* Set .value field equal to position for each element *)
   DO
      myCT^^.ctTable[j].value := j;
   myCT^^.ctFlags := BitOr(myCT^^.ctFlags,$4000);
   (* .ctFlags aka .transindex *)
```

Drawing the unmodified `ppat` 16 would produce it exactly as it appears on the desktop. After the modification, drawing with `myPP` would produce the same pattern with the colors replaced by palette colors.

One use for this might be to draw a `pixMap` with all animated colors, and then let the user adjust color, brightness, and contrast with slider controls. The color changes would be performed with `AnimatePalette` calls.

ACCESSING NEW COLOR LOOK-UP TABLES

Several new 'clut's have been defined for 32-Bit QuickDraw and may be accessed with the `GetCTable` routine. As before, 'clut' IDs 1, 2, 4, and 8 are the standard color tables for those depths. A gray ramp for each depth can be requested using the depth plus 32. 'Clut' IDs for gray ramps with depths 1,2,4, and 8 are 33, 34, 36, and 40 respectively.

'clut's 2 and 4 are modified to include the highlight color and may be accessed as the depth plus 64.

'clut's may not exist as actual resources; the GetCTable routine may synthesize them when they are requested. If there is a 'clut' resource with the specified ID, however, GetCTable will load that resource and return a detached handle to it. So to dispose of the handle, you should call DisposCTable rather than release resource. •

BRAVING OFFSCREEN WORLDS

No one disputes the benefits of using offscreen environments to prepare and keep the contents of windows. The only problem is that creating such environments—from a simple pixMap to a more complicated GDevice—can be rather difficult. 32-Bit QuickDraw includes a set of calls that allows an application to create offscreen worlds easily and effectively. This article describes those calls and provides details on how to use them.

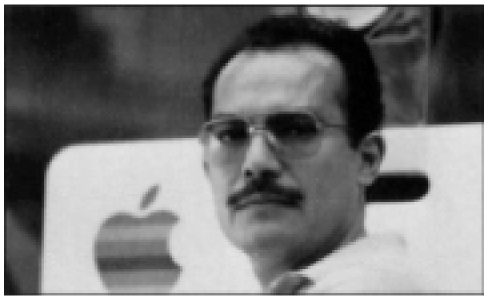
Until now, creating and maintaining offscreen devices or ports has been complicated and confusing at best. As part of 32-Bit QuickDraw, Apple's engineering team has included a set of calls that makes creating and maintaining offscreen devices and ports a real breeze. Using the offscreen graphics environment, QuickDraw can maintain the data alignment necessary to improve the performance of `CopyBits` when you use it to display onscreen the contents of the offscreen buffer.

Also, applications using the offscreen world support from QuickDraw are more likely to benefit from future enhancements to QuickDraw than programs doing their own offscreen management. This can save you a lot of time down the road.

The offscreen world offers a few more benefits:

- The system takes care of all the messy details involved in creating the offscreen `GDevice`. You don't need to bother asking "What flags should I set?" or "What is the `refNum` for an offscreen device?"
- You can tailor the offscreen port according to your specifications rather than being restricted to `screenBits.bounds`. Because the `visRgn` comes set to the right dimensions, you won't need to change anything.
- The `pixMap` associated with the offscreen world comes back from the call complete and ready to use. You won't have to lose sleep wondering if you set the correct value for `rowBytes` or if your `baseAddress` pointer has less memory than it should because the compiler didn't do the multiplication using `Longs`.

Think of the `GWorld` structure as an extension to the `CGrafPort` structure, containing the port information along with the device data and some extra state



GUILLERMO ORTIZ graduated from college with a BSEE, but this event took place so long ago and so far away that he gave up trying to remember when and where. He is truly the Apple veteran among the bunch of authors in this issue, having successfully emerged as a pretty nice guy from six years at Apple--four as an Apple II Pascal "expert" and two working on Color QuickDraw, the Palette Manager, and the

information. In most cases, a `GWorldPtr` can be used interchangeably with a `CGrafPtr`, which makes converting applications quite easy. At this point, however, Apple is keeping the structure of an offscreen graphics world private to allow for further expansion.

For instance, in the following section of Developer Technical Support's sample program FracApp, the new calls were implemented without changing the document's data structure at all. This example illustrates the difference the new set of calls can make when creating offscreen environments.

A LOOK AT THE NEW CALLS

Let's use a section of the sample program FracApp to illustrate the difference this new set of calls can make when creating offscreen environments. The procedure `TFracAppDocument.BuildOffWorld` creates a new device and its accompanying structures for each document. Here is the original code, with comments shortened, followed by the equivalent code using the new calls:

```
PROCEDURE  TFracAppDocument.BuildOffWorld (sizeOfDoc: Rect);
VAR   oldPerm:        Boolean;
      dummy:          Boolean;
      docW, docH:     LongInt;
      fi:             FailInfo;
      currDevice:     GDHandle;
      currPort:       GrafPtr;
      Erry:           OSErr;

   PROCEDURE DeathBuildOff (error: OSErr; message: LONGINT);
   {Error handler}

   BEGIN
      oldPerm := PermAllocation (oldPerm);
      { Set memory back to previous. }

      SetGDevice (currDevice);
      { Set device back to main, just in case. }
      SetPort (currPort);
   END;
```

like. He attributes surviving the 32-bit QuickDraw project to being in good shape from running and playing tennis with good friends. His curriculum vitae is entitled, "My Life As An Elvis Impersonator." Although he profusely denies ever having written that line, there's a trail of sequins leading to his office. 'Fess up, Guillermo. •

All Apple developers (Associates and Partners) received FracApp with their monthly mailings. It is also available on develop, the CD, and through APDA on the DTS sample code disks.•

```
BEGIN
   currDevice := GetGDevice;        { save current for error handling. }
   GetPort(currPort);

   oldPerm := PermAllocation (TRUE);

   CatchFailures(fi, DeathBuildOff);         { any failures, must be cleaned up. }

   { Let's set up the size of the rectangle we are using for the document. }
   docW := sizeOfDoc.right - sizeOfDoc.left;
   docH := sizeOfDoc.bottom - sizeOfDoc.top;


   { Now try to set up the offscreen bitMap (color). }
   fBigBuff := NewPtr (docW * docH);
   FailMemError;                { couldn't get it we die. }

   { OK, now we get wacko.  We need to create our own gDevice, }

   fDrawingDevice := NewGDevice (0, -1);     { -1 means unphysical device.  }
   FailNIL (fDrawingDevice);                 { If we failed, error out. }

   { Now init all the fields in the gDevice Record, since it comes uninitialized. }
   HLock ( Handle(fDrawingDevice) );
   WITH  fDrawingDevice^^  DO  BEGIN
      gdId := 0;     { no ID for search & complement procs }
      gdType := clutType;                    { color table type fer sure. }

      DisposCTable (gdPMap^^.pmTable);{ kill the stub that is there. }
      gdPMap^^.pmTable := gOurColors;{ make a copy of our global color table. }
      Erry := HandToHand (Handle(gdPMap^^.pmTable));
      FailOSErr (Erry);          { if not possible, blow out. }

      { build a new iTable for this device }
      MakeITable (gdPMap^^.pmTable, gdITable, 3);
      FailOSErr (QDError);{ no memory, we can leave here. }
      gdResPref := 3;{ preferred resolution in table. }
      gdSearchProc := NIL; { no search proc. }
      gdCompProc := NIL;         { no complement proc. }
      { Set the gdFlags }
      gdFlags := 2**0 + 2**10 + 2**14 + 2**15;  { set each bit we need. }

      { Now set up the fields in the offscreen PixMap }
      gdPMap^^.baseAddr := fBigBuff;         { The base address is our buffer. }
      gdPMap^^.bounds := sizeOfDoc;          { bounding rectangle to our device. }

      gdPMap^^.rowBytes := docW + $8000;
      gdPMap^^.pixelSize := 8;
      gdPMap^^.cmpCount := 1;
      gdPMap^^.cmpSize := 8;

      gdRect := sizeOfDoc;{ the bounding rectangle for gDevice, too. }
   END;                          { With fDrawingDevice }

   HUnLock ( Handle(fDrawingDevice) );

   { Yow, that was rough.}
   SetGDevice (fDrawingDevice);

   fDrawingPort := CGrafPtr( NewPtr (SizeOf (CGrafPort)) ); { addr CPort record. }
   FailNil (fDrawingPort);       { didn't get it, means we die. }

   { Now the world is created }
   dummy := PermAllocation (FALSE);

   OpenCPort (fDrawingPort);     { make a new port offscreen. }
   FailNoReserve;                { Make reserve, die if we can't }

   { QuickDraw is most obnoxious about making a port that is bigger than the screen,
   so we need to modify the visRgn to make it as big as our full page document }
   RectRgn(fDrawingPort^.visRgn, sizeOfDoc);

   fDrawingPort^.portRect := sizeOfDoc;

   { OK, we have a nice new color port that is offscreen. }
   Success (fi);
```

```
  oldPerm := PermAllocation (oldPerm);

  { Now we have the offscreen PixMap, we need to initialize it to white. }
  SetPort (GrafPtr(fDrawingPort));
  EraseRect (sizeOfDoc);                    { clear the bits. }

  SetGDevice (currDevice);
  SetPort (currPort);
END;                   { BuildOffWorld }
```

Now let's see what the equivalent code looks like when we use the calls provided by 32-Bit QuickDraw and its offscreen support:

```
PROCEDURE TFracAppDocument.BuildOffWorld(sizeOfDoc:RECT);
VAR  oldPerm         :Boolean;
     fi              :FailInfo;
     currDev         :GDHandle;
     currPort        :CGrafPtr;
     erry            :QDErr;

  PROCEDURE DeathBuildOff (error: OSErr; message:LONGINT);

  BEGIN
     oldPerm := PermAllocation(oldPerm);
     SetGWorld (currPort,  currDev);
  END;

BEGIN  (*myBuildOffWorld*)
  GetGWorld(currPort,  currDev);
  CatchFailures(fi, DeathDocument);
  Erry := NewGWorld(fDrawingPort, 8, sizeOfDoc, gOurColors, NIL, GWorldFlags(0));
  FailOSErr(Erry);

  SetGWorld (fDrawingPort,  NIL);

  IF ( NOT LockPixels(fDrawingPort^.portPixMap) ) THEN
  FailOSErr(QDError);

     EraseRect(FDrawingPort^.portRect);

  UnlockPixels (fDrawingPort^.portPixMap);

  SetGWorld (currPort,  currDev);
END {myBuildOffWorld};
```

CALLS YOU CAN'T DO WITHOUT

The new routines simplify the code and help prevent the typical errors you make having to initialize all those special fields. It only makes sense to make your work easier if you can. Here are the calls in the order they appear in the sample code:.

```
PROCEDURE GetGWorld(VAR port:CGrafPtr; VAR gdh:GDHandle)
```

This call takes the place of two standard calls, `GetPort` and `GetGDevice`. It saves the current settings for later restoration and works for offscreen worlds as well as old-style ports..

- `port is set to the current port, which can be a GrafPtr, a CGrafPtr, or a GWorldPtr.`

- `gdh` returns the current device..

```
FUNCTION  NewGWorld(VAR offscreenWorld: GWorldPtr; pixelDepth:
          INTEGER; boundsRect: Rect; cTable:CTabHandle;
          aGDevice: GDHandle; flags: GWorldFlags): QDErr;
```

This is the call that does the work. If the function returns `noErr`, then `offscreenWorld` contains a pointer to the newly created offscreen environment. If the function does not return `noErr`, then something didn't work. Most likely the Memory Manager couldn't allocate enough memory for all the structures. In that case, your program has to decide what to do next, such as draw to the window's port, sacrificing speed, features, or both.

Possible error returns other than `noErr` are `cDepthErr` (no such depth is possible), `paramErr` (illegal parameter), plus any Memory Manager or QuickDraw errors.

- `pixelDepth` must be 1, 2, 4, 8, 16, or 32 bits per pixel. With one exception, other values will make the call return `cDepthErr`. `pixelDepth` can be 0, as described later in this article.

- `boundsRect` is used to calculate the offscreen `pixMap's` size and coordinate system. It is also used to set `portRect, pixMap` bounds, `gdRect`, and the port's `visRgn`.

- When it is provided, `cTable` is copied and the copy is used for the offscreen `pixMap`. If `cTable` is `nil`, then the system uses the default table for the desired depth.

NewGWorld(**offscreenGWorld, pixelDepth, boundsRect, cTable, aGDevice, flags);**

On entry **offscreenGWorld** should point to nothing
On exit **offscreenGWorld** is your new GWorld

Flags can be one of ...

0	pixels not purgeable, create a new device
pixPurge	pixels are purgeable, check LockPixel's return value
NoNewDevice	**pixelDepth** and **cTable** are from **aGDevice**
pixPurge and NoNewDevice	

If **pixelDepth** is 0 ...	optimize for CopyBits speed
boundsRect holds a global rectangle	corresponding to a window
cTable is ignored	uses cTable from boundsRect's deepest screen device
aGDevice is ignored	

If **pixelDepth** is 1,2,4,8,16, or 32 — optimize for offscreen data depth

```
    boundsRect is a rectangle in local coordinates
    if cTable is nil
      NewGWorld uses default color table for given pixelDepth
    else
      NewGWorld attaches a copy of your cTable to your new GWorld
    if NoNewDevice flag is passed (see flags)
      aGDevice is used to create your new GWorld (aGDevice should not be a screen device)
    else
      aGDevice is ignored
```

J.Z.

- When `aGDevice` is not `nil` and the `noNewDevice` flag is set, no new offscreen device is created. This is the case when you want to create an offscreen port but do not need the offscreen device. In our example, `aGDevice` is always `nil` because we want to keep a separate environment for each document being drawn, which requires a unique color table and inverse table, implying the need for individual devices.

  This is useful when, for example, an application has several offscreen worlds with similar characteristics, such as depth or color table. It is possible to allocate only one `GWorld` with an offscreen device and to use that offscreen device as a `GDevice` for all the other `GWorlds`. Be sure to avoid using one of the screen devices as a `GDevice`. If the user changes the characteristics of a screen device via the Control Panel, your offscreen world will become invalid.

Note that `aGDevice` is used to create the offscreen graphics world only when `noNewDevice` is set and its `pixelDepth` is not 0. In any other instance, `aGDevice` should be set to `nil`.

If `pixelDepth` is 0, `boundsRect` is used to find the deepest device that intersects the rectangle. Taken in global screen coordinates, the depth, color table, and inverse table of this device are used to set up the offscreen environment.

- The `flags` parameter gives some control to the application for the creation of the new offscreen graphics environment. Currently the possible values are a combination of `pixPurge` and `noNewDevice`.

  If `pixPurge` is set, the offscreen buffer is created as a purgeable block. The effects of `noNewDevice` are the same, as discussed earlier.

```
FUNCTION GetGWorldDevice(offscreenWorld: GWorldPtr):GDHandle;
```

This call returns the `GDevice` associated with `offscreenWorld`, normally the offscreen device created with `NewGWorld`. If `noNewDevice` was used, however, the device returned will be the device passed to `NewGWorld` or `UpdateGWorld`.

```
PROCEDURE SetGWorld (port: CGrafPtr; gdh: GDHandle);
```

This call replaces `SetPort` and `SetGDevice`. If `port` is a `GrafPtr` or a `CGrafPtr`, then the current port is set to `port` and the current device is set to `gdh`. If `port` is a `GWorldPtr`, then the current port is set to `port` and the current device is set to the device attached to the offscreen graphics world..

As a rule of thumb, when you use the offscreen support provided with 32-Bit QuickDraw, you should use `GetGWorld` instead of `GetPort` and `GetGDevice`, and `SetGWorld` instead of `SetPort` and `SetGDevice`. Both calls are safe to use within the old-style environment and ensure proper behavior in the new.

WHERE IS THE CATCH?

When using the offscreen world environment, you have to make sure that the `pixMap` is actually available and locked when you draw to or from an offscreen graphics world. This is why you must bracket any drawing action with `LockPixels` and `UnlockPixels`.

In our example, to anchor the offscreen `pixMap`, we need to call `LockPixels` just before calling `EraseRect`. When we are done, a call to `UnlockPixels` releases the buffer. These two calls are the only extra work offscreen support in 32-Bit QuickDraw demands.

```
FUNCTION LockPixels (pm: PixMapHandle):Boolean;
```

Call this function prior to any drawing operation to or from the offscreen environment. A false value returned means the offscreen `pixMap` has been purged. A `LockPixels` result of false tells the application to either recreate the offscreen `pixMap`, using `UpdateGWorld` in the process, or draw directly to the target window.

```
PROCEDURE UnlockPixels (pm: PixMapHandle);
```

When you finish drawing, you should call `UnlockPixels`. Just remember that all that is *locked* should be *unlocked*. Otherwise, strange things may happen..

ONE MORE CALL YOU'LL NEED

```
FUNCTION  UpdateGWorld (VAR offscreenGWorld: GWorldPtr;
          pixelDepth: INTEGER; boundsRect: Rect; cTable:
          CTabHandle; aGDevice: GDHandle; flags: GWorldFlags):
          GWorldFlags;
```

This call reconstructs the offscreen environment according to the new `boundsRect`, `cTable`, and `pixelDepth`. These parameters work in large part the same way they do in `NewGWorld`.

When `pixelDepth` is 0, the device list is parsed again to find the deepest device intersecting `boundsRect` taken in global coordinates. If `aGDevice` is not `nil`, then `pixelDepth` and `cTable` are ignored and the fields from `aGDevice` are used offscreen. If the offscreen buffer has been purged, `UpdateGWorld` allocates a new one.

When necessary, the application can simply call `UpdateGWorld` to change the offscreen environment without having to recreate the offscreen image from scratch. This is the case, for example, when the user selects a different depth or the color table is modified somehow.

Keeping the offscreen world parallel to the conditions used to display the images to the user guarantees that when `CopyBits` is called to update a window, the operation will be performed at the highest speed.

The controlling parameter `flags` can take the following values:.

- `[]` I know what I am doing—don't update the pixels for me.
- `[clipPix]` If `boundsRect` is smaller than the current `boundsRect`, the bottom and right edges pixels are clipped out. If `boundsRect` is larger, the bottom and right edges are filled with the background color.

- `[stretchPix]` If `boundsRect` is smaller, the image is reduced to the new size. If the rectangle is larger, the offscreen image is enlarged to fit the new area.
- `[clipPix, ditherPix]` and `[stretchPix, ditherPix]` These provide error diffusion when necessary in addition to clipping or stretching.

<u>UpdateGWorld</u>(offscreenGWorld, pixelDepth, boundsRect, cTable, aGDevice, flags);

On entry **offscreenGWorld** is your old GWorld
On exit **offscreenGWorld** is your new GWorld — the pointer may or may not be the same

Flags can be one of ...

0	pixels not updated
clipPix	preserves pixels it can, puts background in new areas
stretchPix	stretches pixels to fit from old to new pixmap
clipPix and ditherPix	plus the pixels are dithered if necessary
stretchPix and ditherPix	

! If **aGDevice** is not nil then the **pixelDepth** and **cTable** you supply are overridden by the **pixelDepth** and **cTable** of **aGDevice**.

If **pixelDepth** is 0

boundsRect is a global rectangle	typically corresponds to the associated window
cTable is ignored	uses cTable from boundsRect's deepest screen device
aGDevice is nil	watch out for side effects if not nil!

If **pixelDepth** is 1,2,4,8,16, or 32
boundsRect is a rectangle in local coordinates — if different from old boundsRect then pixmap, etc are updated appropriately
if **cTable** is nil
uses default color table for given pixelDepth and updates pixmap if different
else
new (?) color table used to update your pixmap
aGDevice is nil or the device to determine your cTable and pixel depth — J.Z.

Now if `boundsRect` is new but the same size, `UpdateGWorld` realigns the `pixMap` for best performance. With a new `pixelDepth`, the pixels are scaled to the new depth. If `cTable` is new as well, the pixels are mapped to the new colors.

Even for our engineers, some miracles are just too difficult. If the `pixMap` has been purged, it is reallocated, but the old contents are lost.

When `UpdateGWorld` returns, check its result, which will be of the `GWorldflags` variety. If `gwFlagErr` is set, that means the call was unsuccessful. The offscreen world has not changed, and some correcting action is required. The errors might be `cDepthErr` (no such depth is possible), `paramErr` (illegal parameters), and any

Memory Manager or QuickDraw errors..

If the call was successful, the rest of the flags can be interpreted as follows:

`mapPix`	Color mapping was necessary..
`newDepth`	Depth is new.
`alignPix`	Pixels were realigned for best results..
`newRowBytes`	RowBytes changed.
`reallocPix`	pixMap was reallocated..
`clipPix`	Clipping was used..
`stretchPix`	Stretching was used..
`ditherPix`	Dithering was used.

SOME BONUS CALLS.

```
PROCEDURE DisposeGWorld (offscreenGWorld: GWorldPtr)
```

Make this call only when you are one hundred percent sure you don't need `offscreenGWorld` any more, and you want to release the memory it uses..

```
PROCEDURE AllowPurgePixels (pm: PixMapHandle);
```

This call makes the given `pixMap` purgeable. .

```
PROCEDURE NoPurgePixels (pm: PixMapHandle);
```

This one makes the `pixMap` nonpurgeable. .

```
FUNCTION GetPixelsState (pm: PixMapHandle): GWorldflags;
```

Make this call to find out the condition of the flags `pixelsPurgeable` and `pixelsLocked`. When you want to make temporary changes, you can use this call in conjunction with `SetPixelsState` to save and later restore the state of the flags.

```
PROCEDURE SetPixelsState (pm: PixMapHandle; state:
GWorldFlags);
```

This call sets the state of the `pixelsPurgeable` and `pixelsLocked` flags.

```
FUNCTION GetPixBaseAddr (pm: PixMapHandle): Ptr;
```

Since the offscreen world is yours, you will probably feel the urge to mess with it directly. `GetPixBaseAddr` is the call you need. It returns the 32-bit address of the start of the offscreen buffer associated with the given `pixMap`. The address is valid until any call that moves memory around is made. If you make such a call, you must call `GetPixBaseAddr` again. If the offscreen `pixMap` has been purged, the call returns nil.

```
FUNCTION  NewScreenBuffer (globalRect: Rect; purgeable:
          BOOLEAN; VAR gdh: GDHandle; VAR offscreenPixMap:
          PixMapHandle): QDErr;
```

This call creates a `pixMap` using the color table and depth of the deepest device intersected by `globalRect`. It is useful when the offscreen buffer is used to keep a copy of a portion of a window. Normally, applications don't need to make this call.

If the call is successful, `gdh` is a handle to the deepest device and `offscreen pixMap` points to the new `pixMap`. Note that if the screen device returned in `gdh` is changed by the user in `monitors`, the offscreen `pixMap` becomes invalid.

Errors are `noErr` (everything is okay), `cNoMemErr` (couldn't get all the memory needed), `paramErr` (illegal parameters), and any Memory Manager or QD errors. ;

```
PROCEDURE DisposeScreenBuffer (offscreenPixMap: PixMapHandle);
```

You made it, so call this procedure to dispose of it.

CALLS TO AVOID DISASTER

Even though all the books, Technical Notes, and documents that describe how to program the Macintosh talk about the things you shouldn't do and the data fields you are not supposed to mess with, not everyone can resist temptation. That's why 32-Bit QuickDraw includes a set of calls that allows you to tell the system you have modified some forbidden structure, and it should try to accommodate to your designs. The calls are: .

```
PROCEDURE CTabChanged (ctab: CTabHandle)
```

This call says "Yes, I know I shouldn't mess with a color table directly, but I did it and I want to come clean." Use `SetEntries`, or even better let the Palette Manager maintain the color table for you.

```
PROCEDURE PixPatChanged (ppat: PixPatHandle);
```

Use this call to say "I admit I modified the fields of a `PixPat` directly. Please fix the resulting mess for me." When the modifications include changing the contents of the color table pointed to by `PixPat.patMap^^.pmTable`, you should also call `CTabChanged`.

`PenPixPat` and `BackPixPat` are a better way to install new patterns.

```
PROCEDURE PortChanged (port: GrafPtr);
```

You should not modify any of the port structures directly. But if you cannot control yourself, use this call to let QuickDraw know what you've done.

If you modify either the `pixPat` or the color table associated with the port, you need to call `PixPatChanged` and `CTabChanged`.

```
PROCEDURE GDeviceChanged (gdh: GDHandle);
```

The best practice is to stay away from the fields of any `GDevice`. But if you do change something, make this call so the system can rectify any problems. If you change the data of the color table in the device's `pixMap`, you must also call `CTabChanged`.

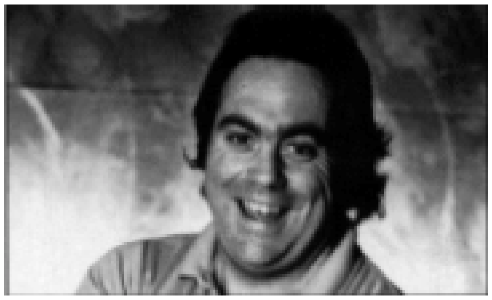
THE PERILS OF POSTSCRIPT

Letting your application rather than the LaserWriter driver convert QuickDraw commands into PostScript is simple in most cases, yet when you use direct PostScript to print documents, subtle interactions between the QuickDraw and PostScript imaging models can cause problems. This article will help you in two important areas: using a font from PostScript while selecting it using QuickDraw and preserving your PostScript state while using QuickDraw to select fonts.

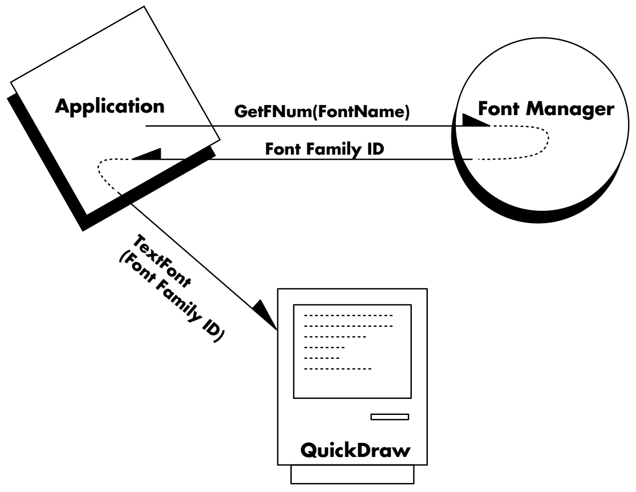
When selecting a PostScript font from QuickDraw, an application first calls `GetFNum` (see *Inside Macintosh*, volume I, page 223 [IM I-223]) to get the Font Family ID for a particular font. It then calls `TextFont` (IM I-171) to actually select it. The name passed to `GetFNum` is the name of the font as seen in the Font menu (for example, Helvetica).

In PostScript, fonts are selected by name using the `findfont` (see *PostScript Language Reference Manual*, page 156 [PLRM 156]) and `setfont` (PLRM 215) operators. If the application attempts to select a font named Helvetica®, however, it will find that this font doesn't exist. This is because the LaserWriter performs a special operation on the font called encoding. Font encoding is the process of mapping missing characters into another font.

For example, a character like ø may not exist in the standard Helvetica font. In order to provide that character, the LaserWriter driver will modify the Helvetica font, inserting a reference to the ø character in the Symbol font. Once this is done, the font is no longer standard Helvetica, so it is renamed. The actual name is something like |_____Helvetica, but this naming convention is not standard and could change in the future.



SCOTT "ZZ" ZIMMERMAN is a DTS printing guru. (He's particularly impressed with the strictly enforced dress code at Apple.) In his spare time he sails, scuba dives for lobsters, and plays the piano, guitar, and saxophone. His doorway is adorned by a melted gummy rat, a good luck charm from his Intel days. At home, atop his monitor is perched a rare Asian black



So if you don't know the font's name, how can you select it? Simple, let QuickDraw do it. When you select a font via `TextFont` and then use it via one of the QuickDraw text drawing routines (such as `DrawChar` or `DrawString` [IM I-172]), the LaserWriter driver handles the complex task of selecting an appropriate font on the PostScript device. This includes downloading and encoding the font if necessary. Using QuickDraw to select the font not only saves you a lot of work, but also improves compatibility. The process of font downloading and character encoding could change in the future, and if your application does it internally, it will have to be revised. If you use QuickDraw to download the font, your application will be immune to changes in the font downloading mechanism.

PICK A FONT, ANY FONT

Now let's look at the code to actually select a font. The following procedure will select a font for any device, QuickDraw or PostScript:

```
PROCEDURE   SetFont;(fontName: Str255; fontSize: INTEGER; fontStyle:
            Style);
VAR
    theFontID:  INTEGER;
    thePenLoc:  Point;
BEGIN
    GetFNum(fontName, theFontID);              (* Get the font ID. *)
    TextFont(theFontID);                       (* Set it *)
    TextSize(fontSize);                        (* Set the size *)
    TextFace(fontStyle);                       (* ...and the style. *)
    GetPen(thePenLoc);         (* Save the current pen position. *)
    DrawChar(' '); (* Draw a space so the font gets downloaded.*)
    MoveTo(thePenLoc.h, thePenLoc.v);          (* Restore original pen *)
                                               (* position. *)
END;
```

scorpion (behind glass, we hope). His other cuddly pets include two geckos and an iguana. •

There are two important things to note in the `SetFont` procedure above. First, the procedure uses the `GetFNum` trap to get the Font ID. This is essential to make sure that you get the correct font. (See Technical Note #191, Font Names for more information.) Second, the `SetFont` procedure calls `DrawChar` to draw a space. This is required to force the font selection on PostScript devices, since the `TextFont` call only changes the `txFont` field of the `GrafPort`. By actually using the font (via `DrawChar`) the LaserWriter driver's `StdText GrafProc` is called, and selects the font on the printer. Subsequent calls to the PostScript `show` (PLRM 222) operator will use this font. Since `DrawChar` will change the pen position, it is saved (via `GetPen` [IM I-169]) and restored (via `MoveTo` [IM I-170]).

ON WITH THE SHOW

Now that we have a font selected, we need to actually draw something with it. For now, as an example, let's say that we want to draw some text with the show operator. We'll send our PostScript using the following procedure. Although convenient for sending PostScript in our example, this method is very inefficient and should not be used in an application. Here's the code:

```
PROCEDURE SendPostScript(theComment: Str255);
VAR
   PSCommand       : Str255;
   CommandHdl      : Handle;
   CRString        : Str255;
   theError        : OSErr;
BEGIN
   CRString := ' ';
   CRString[1] := CHR(13);
   PSCommand := theComment;
   PSCommand := CONCAT(PSCommand, CRString);
   theError := PtrToHand(POINTER(ORD(@PSCommand) + 1),
   CommandHdl,LENGTH(PSCommand));
   if theError <> noErr THEN BEGIN
                  (* Handle the error! *)
   END;
   PicComment(PostScriptHandle,
                  LENGTH(PSCommand), CommandHdl);
   DisposHandle(CommandHdl);
END;
```

The procedure simply takes a string of text, adds a carriage return at the end of it, and converts it into a handle. The handle is then passed to the `PostScriptHandle` picture comment, which actually sends it to the printer. Since this procedure created the handle, the procedure also disposes of it. Again, this is not how a normal application would do it, but it keeps things nice and localized for this example. So now that we can send PostScript, consider the following:

```
SetFont('Helvetica', 14, [bold]);
PicComment(PostScriptBegin, 0, NIL);
      (********************************************)
      (*** QuickDraw representation of graphic. ***)
      (********************************************)
   (* These calls are only executed by QuickDraw *)
   (* (i.e. non-PostScript) devices.         *)
      MoveTo(50, 50);
      DrawString('This is some gray text.');
      PenPat(ltGray);
      MoveTo(100, 100);
      LineTo(300, 300);
      (*********************************************)
      (*** PostScript representation of graphic. ***)
      (*********************************************)
   (* These calls will only be executed by PostScript devices.*)
   SendPostScript('50 50 moveto (This is some gray text.) show');
   SendPostScript('.10 setgray');
   SendPostScript('100 100 moveto 300 300 lineto stroke');
PicComment(PostScriptEnd, 0, NIL);
```

In this fragment, the call to `SetFont` sets the PostScript `currentfont` to be Helvetica. The `PostScriptBegin` comment is used to suppress QuickDraw calls on PostScript devices, and vice versa. When the LaserWriter sees `PostScriptBegin`, it ignores all QuickDraw drawing calls, and just executes picture comments. When a `PostScriptEnd` is received, the LaserWriter will once again interpret QuickDraw calls. The LaserWriter driver will ignore the QuickDraw representation, and begin executing the `SendPostScript` calls. The first one draws a string of text, the second one changes the default gray level of the printer from 100% black to 10% black using the `setgray` (PLRM 216) operator, and the third one draws a diagonal line using the new gray level. Note that the QuickDraw representation for a gray level is handled by using `PenPat` (IM I-170).

SAVE THE POSTSCRIPT STATE

The fragment we just looked at illustrates a good method for sending both QuickDraw and PostScript. It also demonstrates a new problem. When the `PostScriptBegin` comment is sent, the LaserWriter driver performs a PostScript `gsave` (PLRM 166) operation. This saves the current graphics state required for QuickDraw printing. The application can then do what it needs to the state without having to worry about side effects on the QuickDraw environment. When the LaserWriter driver receives a `PostScriptEnd` comment, it performs a `grestore` (PLRM 165) operation to restore the QuickDraw state. Normally this is exactly what you would want. But there are cases when an application may want to execute some QuickDraw commands without losing the PostScript state is has setup.

For example, the above code fragment set the gray level of the printer to 10%. At the time we did the `PostScriptEnd` comment, the gray level was restored to 100%. If we then want to change the font size, and redraw the text, we would have to resend the setgray operator. It would look like this:

```
(* Change the font size.*)
SetFont('Helvetica', 24, [bold]);
PicComment(PostScriptBegin, 0, NIL);
   (*******************************************)
   (*** QuickDraw representation of graphic. ***)
   (*******************************************)
   (* These calls are only executed by QuickDraw *)
   (* (i.e. non-PostScript) devices.*)
   (* The QuickDraw state is unaffected, so there’s *)
   (* no need to call PenPat again. *)
   MoveTo(250, 50);
   LineTo(750, 50);

   (********************************************)
   (*** PostScript representation of graphic. ***)
   (********************************************)
   (* These calls only executed by PostScript devices. *)
   (* Since the PostScript state was cleared, we need *)
   (* to resend the setgray operator. *)
   SendPostScript('.10 setgray');
   SendPostScript('250 50 moveto 750 50 lineto');
PicComment(PostScriptEnd, 0, NIL);
```

Although resending the `setgray` operator isn't difficult, an application may have set a lot more attributes. To avoid the overhead of resending this state, a new comment may be used. This comment is #196—`PostScriptBeginNoSave`.

When `PostScriptBeginNoSave` is used with `PostScriptEnd`, the `gsave` and `grestore` operations are not performed. This means that the application is completely responsible for the graphics state of the printer. If you are doing all of your imaging via PostScript this is not a problem. If you plan on mixing PostScript and QuickDraw, you must be very careful. Changes to attributes like line width and the transformation matrix will have a significant effect on QuickDraw drawing operations. If the comment is used for the above example, the code will look like this:

```
(* Now illustrate the use of the PostScriptBeginNoSave  *)
(* PicComment. *)
PicComment(PostScriptBeginNoSave, 0, NIL);
   PenPat(ltGray);
   SendPostScript('.10 setgray');
PicComment(PostScriptEnd, 0, NIL);

(* At this point, the gray level of the device is 10% black *)
(* Now draw something using this state. *)
(* Draw a light gray line using QuickDraw. *)
MoveTo(50, 400);
Line(100, 100);

(* At this point, the gray level is still 10%, so we must *)
(* reset it  to black. *)
PicComment(PostScriptBeginNoSave, 0, NIL);
   PenPat(black); (* Reset QuickDraw gray level.   *)
   SendPostScript('1.0 setgray'); (* Reset PostScript gray*)
                                  (* level. *)
PicComment(PostScriptEnd, 0, NIL);
```

Note that instead of sending `PostScriptBegin` as the first operation, we now send `PostScriptBeginNoSave`. We then change the gray level to light gray in the QuickDraw world, and 10% black for PostScript. Since we used `PostScriptBeginNoSave`, sending `PostScriptEnd` does not effect the state of the printer (i.e. the gray level remains at 10%). Now we want to draw something with the new state. We first send the `PostScriptBegin` comment, which saves the state we set up, as well as disabling the QuickDraw calls on PostScript devices.

We then send a QuickDraw representation of the line, followed by `PostScriptEnd`. On QuickDraw devices, the line will be drawn using the ltGray pen pattern. On PostScript devices, the line will be drawn using 10% black. After the line has been drawn, we need to reset the state of the device for subsequent drawing operations. This is done by once again sending the `PostScriptBeginNoSave` comment, followed by the commands to reset the gray level, as well as any other attributes of the printer.

In summary, we have looked at two ways of avoiding the perils of PostScript. The first was how to use a font from PostScript while choosing it using QuickDraw. The supported method for this was demonstrated by the `SetFont` procedure. The second was how to preserve your PostScript state while still using QuickDraw to select fonts.

COMPATIBILITY: RULES OF THE ROAD

Apple's System Software Version 7.0 provides the most important test of compatibility since the introduction of the Macintosh II. This article should help you prepare for the release of System 7.0. For an overview of the most critical compatibility issues and how to address them, read on.

If you've already read too many stuffy articles full of dire warnings about compatibility, you've probably decided this one will be best suited for lining the bottom of your filing cabinet. But before doing that, consider the case of Johnny Appledweeb.

Ace Macintosh programmer for Cliff Grazer Enterprises, Johnny is currently putting in the long hours to get a spread-processor-terminal-graphics-emulator out the door. He doesn't have time to read an article like this because Cliff Grazer, his boss and President of CGE, is all over his case. Four months ago, the company began accepting prepayment from customers who can't wait for Johnny's program to reach their local stores. Those customers are now beating down the doors.

Although Cliff is desperate to get the product out, he has required certain levels of performance. The application must be kept under the 1megabyte limit, for example, and must keep up at 19.2 kbaud through the modem port. Finally, at the last minute, legal decides to require copy protection. Once the application ships, reviews are excellent, customers are happy, and sales are good. Cliff is ecstatic and gives Johnny a big raise. Johnny has time to relax a bit and maybe even catch up on some reading. But this article doesn't interest him because he's a crack programmer, and his application works fine. Bottom of the filing cabinet time.

Six months later, Johnny comes back from his well-earned vacation to find that Apple has introduced new machines and released a new version of the system software. Cliff's hopping mad because of reports of compatibility problems and complaints from angry customers. As Johnny begins to look into the problems, he has a vague recollection of some article he saw on compatibility. He rummages around, finds the article, and quickly discovers it addresses his problems. But it's too late for the customers. They don't understand compatibility, but they do understand that the application they have been using every day no longer works. Cliff doesn't

DAVE RADCLIFFE, "Technical Sherpa," has been with Apple about a year and a half, putting his chemistry degree from Washington University to work in A/UX® and MPW™ technical support. Actually, he discovered his true calling while working with the computers in the UCLA chemistry research labs. When asked how he's changing the world one person at a

really understand compatibility either. What he understands is he now has the expense of shipping updated versions to keep his customers happy.

Johnny might have saved himself and others a lot of trouble if he'd spent a few minutes with this article right away. Sure, it probably would have meant a delay in the first release of the application, but it might also have made a second release unnecessary. Johnny's a good programmer, and he's aware of almost everything in this article, but if a single sentence had helped avoid problems, the article would have been worth Johnny's time.

It may be worth your time as well to check out the compatibility of your current application's features with System 7.0. The road gets a little dry and dusty from here on, so grab a cold one and we'll get down to business. This article focuses on specific areas of Macintosh programming where compatibility might trip you up today or in the future. It isn't meant to be a guide to Macintosh programming, so if you need additional information on a topic, such as implementation details, refer to *Inside Macintosh*, volumes I-V, and the Macintosh Technical Notes.

DEFENSIVE PROGRAMMING

Murphy was clearly a computer engineer. If anything can go wrong with your application, it will, as most of us learn the hard way. Once you recognize that users always stress your program in ways you never thought possible, you acquire defensive programming habits.

TESTING

Always test return values for possible errors because you never know when some unusual situation will arise. Assume that data structures will change. The Memory Manager is an example of a manager whose data structures are changing, as described later in this article. Avoid any portion of a data structure marked "Unused"—its use is reserved for Apple.

MEMORY ALLOCATION

If you treat the Memory Manager with a little courtesy and respect, your application will live a long and happy life. Keep in mind the strengths as well as the limitations of the Memory Manager and listen to what it tells you. Believe it when it returns a nil handle to tell you of memory allocation failure. Every application's memory needs are different, and as you design your application, think about how memory you allocate will be used. A little planning can ease the Memory Manager's task by reducing the number of Memory Manager calls and minimizing fragmentation and thrashing.

You should ask yourself a few simple questions about the memory you allocate in the heap. Is this memory you will need frequently? Rather than frequently allocating and releasing the memory, wouldn't it be better to allocate it once at the start of your

time, Dave replied, "home-brewed beer." In addition to home concoctions, he's into hiking, backpacking, and photography. •

application; if it is a handle, move it high in the heap with MoveHHi; and simply reuse it when necessary?

Is it memory that shouldn't move? If so, consider the use of NewPtr instead.

Is this a large block of memory used for a very short period of time? Judicious use of MultiFinder temporary memory can satisfy such needs and reduce overall heap usage, allowing you to shrink your MultiFinder size partition.

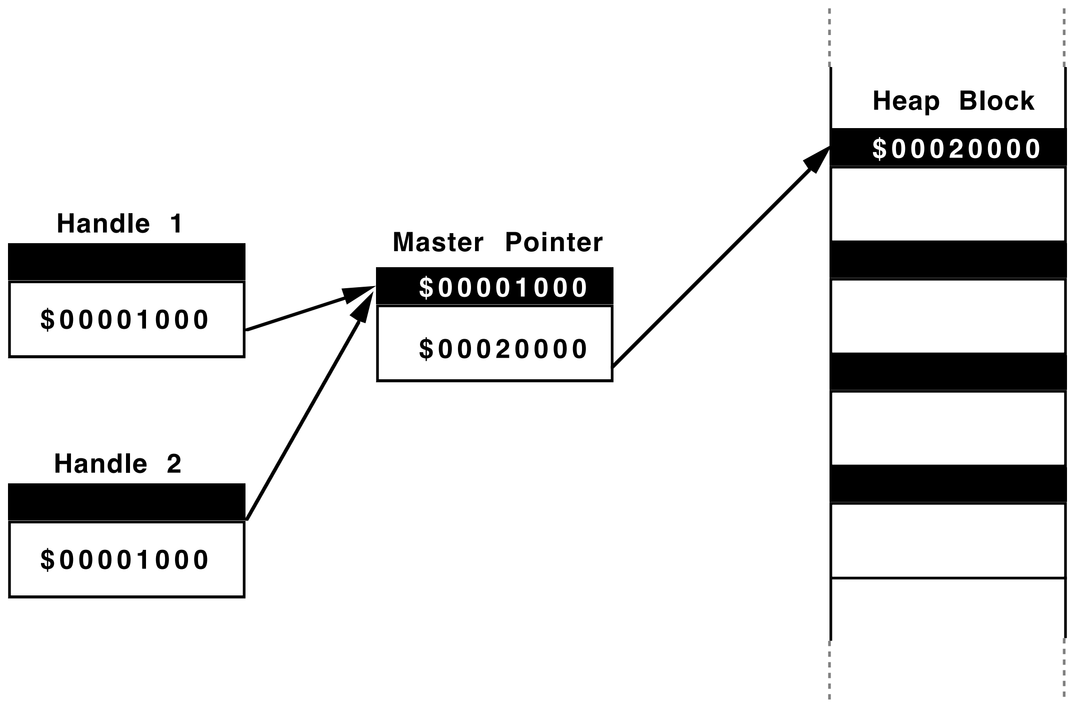
Is this memory you are willing to let the Memory Manager dispose of at its discretion, such as for a resource? Then you should consider making it purgeable. But if you've made it purgeable, be sure to check for empty handles.

Once you have your application working, be sure to stress test your use of the Memory Manager. You can do this by using your debugger to force heap scrambling and purging. You can also simulate low memory conditions by running your application in a small MultiFinder partition.

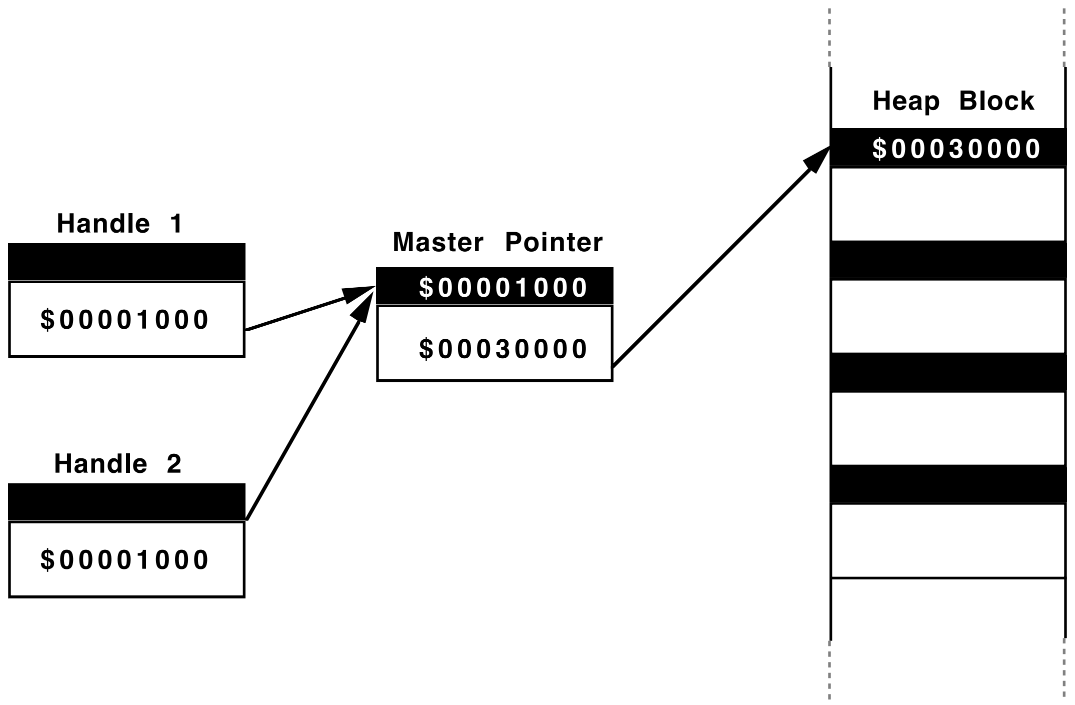
32-BIT CLEANLINESS

Another way to treat the Memory Manager with kindness and respect is to practice 32-bit cleanliness. Being 32-bit clean may be the single most important compatibility issue facing developers. To understand what 32-bit cleanliness means, let's take a closer look at Macintosh memory management. The Memory Manager maintains free-form memory structures called heap zones. It allocates memory blocks of various sizes within these zones to satisfy memory allocation requests by the system and applications. Occasionally, heaps will become full or fragmented and the Memory Manager will need to rearrange or purge blocks in a zone to create enough contiguous space to satisfy a memory allocation request. To minimize confusion that could occur when blocks are rearranged, the Memory Manager uses indirect references called handles to refer to relocatable blocks in the heap.

The Memory Manager maintains a series of master pointers referring to blocks in memory. A handle is a pointer to a master pointer, as shown in the following illustrations.

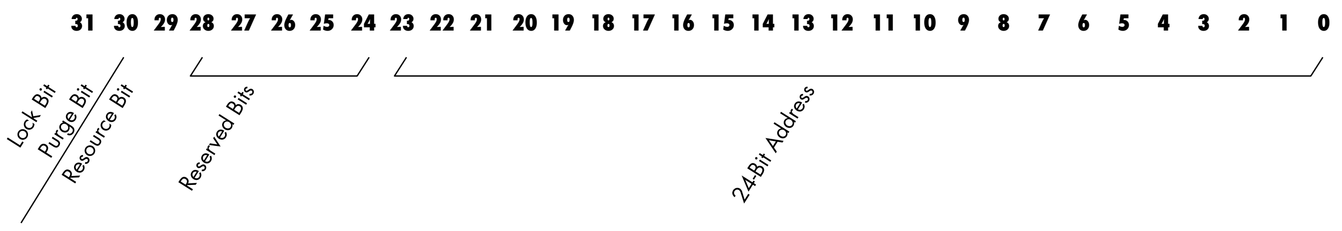


In the example in the first illustration, two independent handles refer to the same heap block at address $20000 via the master pointer at address $1000. Only the master pointer should be referring to the heap block. Now, suppose the system needs to relocate the heap block to address $30000. The second illustration shows the state of the system after relocating the block.



The master pointer is now correctly set to point to the new block. The master pointer is the only thing the Memory Manager had to update. The original handles 1 and 2 still correctly refer to the heap block because they refer to the master pointer, which has the correct location of the heap block.

The classic Macintosh has what is referred to as a 24-bit memory management system. To the hardware, only the lower 24 bits of a 32-bit address are significant. The upper 8 bits are always ignored in a hardware address reference. The Memory Manager maintains certain information about heap blocks, such as whether they are locked in memory and cannot be moved or whether they can be purged from memory to free up space in the heap. The original Macintosh Memory Manager took advantage of the unused upper 8 bits of the address in a master pointer to maintain flags about heap blocks. The illustration shows the master pointer structure of the 24-bit Memory Manager.



AVOIDING COMMON PROBLEMS

The most common violation of 32-bit cleanliness involves direct manipulation of Memory Manager flags. In a 32-bit system, all 32 bits of an address are valid, and in the case of a master pointer, the flags bits are stored elsewhere. The system provides traps for setting and cleaning these flags: `HLock/HUnlock`, `HPurge/HNoPurge` and `HSetRBit/HClrRBit`. There are also traps for getting and setting all the flags at once: `HGetState` and `HSetState`. Some applications have taken advantage of knowledge of the master pointer structure to set and clear the flag bits directly. Setting flag bits directly on a 32-bit system means you are not changing the flags, but changing the address itself, and suddenly your master pointer is pointing to a completely different location in memory.

The issue of 32-bit cleanliness is not limited to proper use of master pointer flags. Every address reference must assume all 32-bits are valid. If you have used any of the upper 8 bits of pointers or handles for anything other than part of an address, you must find an alternate representation for that information.

Two other places you can be bitten by 32-bit violations are in window definition functions (WDEFs) and control definition functions (CDEFs). The original Macintosh Window Manager stored the window variation code in the upper 8 bits of the handle to the window definition procedure. If you are using custom WDEFs and need to access the window variation code, use the `GetWVariant` trap. Similarly, use `GetCVariant` to retrieve the variant control value for a control that was formerly stored in the high bits of the control defproc handle.

Using pre-System 7.0 software, including A/UX 1.1, it is impossible to write a strictly clean CDEF. The problem with custom CDEFs is that the `calcCRgns` message uses the high bit of the region handle as a flag. *Inside Macintosh*, volume I, page 331 incorrectly advises you to “clear the high byte (not just the high bit) of the region handle before attempting to update the region.” Rather, you should clear only the high bit (not the high byte). This makes the reasonable assumption, given the current system software, that the handle represents only a 31-bit address and clearing the high bit is not harmful.

With System 7.0, the Control Manager has a new way of telling your CDEF to calculate control regions. Two new messages have been defined, `calcCntlRgn` and `calcThumbRgn`, with values of 10 and 11 respectively. With a 32-bit Memory Manager in operation, the Control Manager, which previously would have used `calcCRgns`, will now use one of the new messages. With a 24-bit Memory Manager operating, `calcCRgns` will still be used, so you must continue to support that method.

CREATING VALID HANDLES

Just as the master pointer structure will change in System 7.0, other Memory Manager structures will be subject to change. As a precaution, you should not access Memory Manager data structures directly or attempt to "walk the heap" yourself.

Since a handle is a pointer to a pointer, it is possible for an application to create a handle itself, a so-called fake handle. If you pass a fake handle to any Memory Manager routine, the Memory Manager will assume it is a valid handle under its control and may try to relocate or dispose of it. You should never pass a fake handle to any Macintosh trap, because you never know when that trap may itself call the Memory Manager.

Prior to System 7.0, handles allocated with `MFTempNewHandle` trap were not true handles and could not be passed, directly or indirectly, to Memory Manager traps. They were to be treated as fake handles. Under System 7.0, this is no longer true; the Memory Manager knows how to manage such memory.

Remember that MultiFinder temporary memory is just that, temporary. It should be allocated, used, and released as quickly as possible, preferably within one event loop cycle. With System 7.0, you can use the `HPurge` Memory Manager trap to mark handles as purgeable. You can continue to use the memory as long as MultiFinder does not need it for another application. But be sure to check for empty handles to ascertain if your memory has been purged.

USING STRIPADDRESS

One of the keys to 32-bit cleanliness is proper use of the `StripAddress` trap. `StripAddress` is necessary because handle flags in master pointers can create dirty address references. When a 24-bit Memory Manager is operating, `StripAddress` clears the high byte of the address, and returns a clean address. The operation of `StripAddress` is simple enough. What is not always so clear is when use of `StripAddress` is necessary or even appropriate.

To understand the operation of `StripAddress`, consider, again, the second illustration. Imagine that a 24-bit Memory Manager is in operation and you've called `HLock` to lock the handle. The value of the master pointer will now be $80030000 because `HLock` has set the lock bit in the master pointer as indicated in the third illustration. In normal operation, you never need to concern yourself with that high byte because the hardware ignores it. In other words, the hardware quietly strips the address for you. But suppose you are writing a driver that needs to access a NuBus board. To do that, you need to switch the hardware to 32-bit addressing mode using `SwapMMUMode`. Now, suddenly, the hardware is no longer ignoring that high byte, so to access the address properly you first need to call `StripAddress` to clean up the address.

Another situation in which `StripAddress` is necessary is comparing master pointers. In the previous example, comparing the value of the master pointer before and after calling `HLock` would lead you to conclude the master pointer is now pointing to a different block because the comparison looks at all 32 bits. To be sure you are comparing the relevant portions, namely the addresses, call `StripAddress` before comparing master pointers.

If a 32-bit Memory Manager is in operation, `StripAddress` will return the address unchanged, because all 32 bits of the address are valid. If you have used `StripAddress` correctly, you need never worry whether a 24-bit or 32-bit Memory Manager is operating, because `StripAddress` does the right thing.

Finally, do you need to call `StripAddress` on other addresses, such as handles? No, because there should be no extraneous bits set in the high byte of handles. If you are using the high byte of handles for your own purposes, go directly to the beginning of this section on 32-bit cleanliness. Do not pass Go; do not collect $200.

FILE ACCESS

Use the File Manager for all your file access. Avoid assumptions about the underlying file and directory structure. Not only has the Macintosh file system changed in the past, but you might not even be accessing a Macintosh volume. Foreign file systems such as DOS, ProDOS®, High Sierra ISO 9660, and Unix are all supported. If your application is running under A/UX, there may be no Macintosh volumes. These file system differences create many subtle problems. For example, Unix filenames are case-sensitive, whereas Macintosh names are not. Unix uses '/' as a pathname delimiter, while Macintosh uses ':'. Different file systems may have different restrictions on the length of filenames. Always use `SFGetFile` and `SFPutFile`. Not only will this ensure maximum compatibility across file systems, but it will be comforting to your users that your application looks and behaves like other Macintosh applications.

PRINTING

Apple is working to make printing easier for Macintosh programmers in the near future. Meanwhile, we can offer some help in two areas that often cause problems when printing: handling print records and using PostScript.

HANDLING PRINT RECORDS

Some applications set fields in the print record to change the default settings of items in the print dialogs. Rather than modify these fields, applications should just save the print record after the user has configured it. The best method for saving the record is to save it as a resource in your document's resource fork. Since a valid handle already points to the print record, creating a resource is easy:

```
/* This is an example of saving a print record into a resource file.  Saving the      */
/* print record in document resource files provides a method of retaining the         */
/* user setting from the last print job. For example, if a user elects to print a     */
/* document using landscape orientation, that information is stored in the print      */
/* record. If the record is saved with the document, the orientation information      */
/* will be available for the next time the document is printed.  When the 'Page       */
/* Setup' dialog is presented, the user's choices from the last time the document     */
/* was printed will be displayed as defaults.  This provides a convenient, device     */
/* independent method for saving print job information.                               */
/*  NOTE: Information from the Page Setup dialog is saved into the print record.      */
/*  Information from the Print dialog (i.e. # of copies, page range...) is            */
/*  considered to be per job information, and is not saved.  This method             */
/*  will not allow you to provide new defaults for the PrJobDialog.                   */
/*                                                                                    */
/*  Version              When          When         What                              */
/*  1.0                  7/18/89       Zz           First release.                    */
/*                                                                                    */
```

```
#include <Values.h>
#include <Types.h>
#include <Resources.h>
#include <QuickDraw.h>
#include <Fonts.h>
#include <Events.h>
#include <Windows.h>
#include <Menus.h>
#include <TextEdit.h>
#include <Dialogs.h>
#include <Desk.h>
#include <ToolUtils.h>
#include <Memory.h>
#include <SegLoad.h>
#include <Files.h>
#include <OSUtils.h>
#include <OSEvents.h>
#include <DiskInit.h>
#include <Packages.h>
#include <Printing.h>
#include <Traps.h>

/* POPT = Print OPTions.  This type can be anything   */
/* but to avoid confusion with Printing Manager       */
/* resources, the following types should NOT be       */
/* used: PREC, PDEF, & POST...                  */
#define gPRResType 'POPT'

/* This can also be any value.  Since there should    */
/* only be one print record per document, the ID is   */
/* a constant.                      */
#define gPRResID   128

/* Resource name. */
#define gPRResName "\pPrint Record"

/* Define the globals for this program... */
THPrint gPrintRecordHdl;
short   gTargetResFile;
```

```
/* ReportError                     */
/*                                 */
/* This procedure is responsible for reporting an error to the user.  This is     */
/* done by first converting the error code passed in theError into a message      */
/* that  can be displayed for the user.  See Technical Note #161, "When to call   */
/* PrOpen and PrClose".                                                           */
void ReportError(theError)
OSErr theError;
{
   /* Real programs handle errors by displayed comprehensible error messages.     */
   /* This is NOT a real program...                                               */
   if (theError != noErr)
      SysBeep(10);
}

/* InitializePrintRecord                                                          */
/*                                                                                */
/* This procedure is responsible for initializing a newly created print record.  */
/* It begins by calling PrintDefault to fill in default values, and then presents */
/* the standard 'Page Setup' dialog allowing the user to specify page setup       */
/* options.  The modified print record is then returned.                          */
void InitializePrintRecord(thePrintRecord)
THPrint thePrintRecord;
{
   Boolean     ignored;

   PrOpen();
   if (PrError() == noErr) {
      PrintDefault(thePrintRecord);
      ignored = PrStlDialog(thePrintRecord);
   }
   PrClose();
}
/* SavePrintRecord                                                                */
/*                                                                                */
/*  This procedure is responsible for saving a print record into a resource file. */
/* On entry, the print record should be initialized, and the resource file should */
/* be open with permission to write.                                              */
```

```
void SavePrintRecord(thePrintRecord, theResFile)
THPrint thePrintRecord;
short theResFile;
{
   short   currentResFile;
   Handle  existingResHdl;
   Handle  newResHdl;
   OSErr   theError;

   /* First save the currently selected resource file (before calling UseResFile).*/
   currentResFile = CurResFile();

   /* Now select the target resource file.                                        */
   UseResFile(theResFile);
   theError = ResError();
   if (theError == noErr) {
      existingResHdl = GetResource(gPRResType, gPRResID);
      if (existingResHdl != NULL) {
   /* There is already a print record resource in this file, so we need to        */
   /* delete it before adding the new one.                                        */
        RmveResource(existingResHdl);
        theError = ResError();
        if (theError == noErr) {
        /* If the resource was successfully removed, dispose of its memory        */
        /* and update the resource file.                                          */
          DisposHandle(existingResHdl);
          UpdateResFile(theResFile);
        }
      }
      if (theError == noErr) {
      /* Okay, now we have successfully opened the file, and deleted any          */
      /* previously saved print record resources.  Finally we can add the new     */
      /* one...                                                                   */
      /* Since the Resource Manager is going to keep the handle we pass it,       */
      /* we need to make a copy before calling AddResource.  We'll let the        */
      /* system do it for us by calling HandToHand.                               */
```

```
        newResHdl = (Handle)thePrintRecord;
        theError = HandToHand(&newResHdl);
        if (theError == noErr) {
          AddResource(newResHdl, gPRResType, gPRResID, gPRResName);
          theError = ResError();
          if (theError == noErr)
               UpdateResFile(theResFile);
          theError = ResError();
        }
      }
   }
   if (theError != noErr)
      ReportError(theError);

   /* Be polite and restore the original resource file to the top of the chain.   */
   UseResFile(currentResFile);
}


/* GetPrintRecord                                                               */
/*                                                                              */
/*  This function is responsible for loading a resource containing a valid print */
/* record.  On entry theResFile should be open with permission to read.         */
THPrint GetPrintRecord(theResFile)
short theResFile;
{
   short   currentResFile;
   Handle  theResource;
   OSErr   theError;
   currentResFile = CurResFile();
   UseResFile(theResFile);
   theError = ResError();
   if (theError == noErr) {
      theResource = GetResource(gPRResType, gPRResID);
      theError = ResError();
      if (theError == noErr) {
        PrOpen();
        theError = PrError();
        if (theError == noErr) {
          if (PrValidate((THPrint)theResource)) ;
        }
        PrClose();
      }
   }
   if (theError != noErr)
      ReportError(theError);
   UseResFile(currentResFile);
   return((THPrint)theResource);
}
/* TestPrintRecord                                                       */
/*                                                                       */
/*  This procedure is used to test a print record.  It will print a line of text */
/* using the options specified in thePrintRecord passed.  On exit, a line of text */
/* will have been printed.                                               */
void TestPrintRecord(thePrintRecord)
THPrint thePrintRecord;
{
   GrafPtr     currentPort;
   TPPrPort    thePMPort;
   OSErr   theError;
   TPrStatus   thePMStatus;
   GetPort(&currentPort);
   PrOpen();
   if (PrError() == noErr) {
      if (PrJobDialog(thePrintRecord)) {
        thePMPort = PrOpenDoc(thePrintRecord, NULL, NULL);
        if (PrError() == noErr) {
          PrOpenPage(thePMPort, NULL);
          if (PrError() == noErr) {
               SetPort(&thePMPort->gPort);

               MoveTo(100, 100);
               DrawString("\pThis is a test...");
```

```
            }
              PrClosePage(thePMPort);
            }
            PrCloseDoc(thePMPort);
            if (((*thePrintRecord)->prJob.bJDocLoop == bSpoolLoop) && (PrError() == noErr))
              PrPicFile(thePrintRecord, NULL, NULL, NULL, &thePMStatus);
          }
      }
      theError = PrError();                 /* Any errors?                          */
      PrClose();          /* Close the Printing Manager before attempting */
                  /* to report the error.           */
      if (theError != noErr)                /* If there was an error during printing...*/
         ReportError(theError);             /* ...report the error to the user.     */
      SetPort(currentPort);
   }
   main()
   {
      InitGraf(&qd.thePort);
      InitFonts();
      InitWindows();
      InitMenus();
      TEInit();
      InitDialogs(NULL);
      InitCursor();

      /* Get the ID of our resource file.  Since we were just opened, the         */
      /* CurResFile() will be ours.  In a real application, the resource file ID  */
      /* would be the ID of your application's document file.                     */
      gTargetResFile = CurResFile();

      /* Create a valid print record                                              */
      gPrintRecordHdl = (THPrint)NewHandle(sizeof(TPrint));
      if (gPrintRecordHdl != NULL) {
         /* Okay, we got a print record, now initialize it.                       */
         InitializePrintRecord(gPrintRecordHdl);
```

```
        /* Now save the print record into the resource file.     */
        SavePrintRecord(gPrintRecordHdl, gTargetResFile);

        /* Now that it's saved, kill it off.  We'll restore it by  */
        /* calling GetPrintRecord.                                    */
        DisposHandle((Handle)gPrintRecordHdl);
        gPrintRecordHdl = NULL;

        /* Now get the print record from the file.  Since the     */
        /* record will be loaded as a resource handle anyway, let  */
        /* GetPrintRecord allocate the handle.                        */
        gPrintRecordHdl = GetPrintRecord(gTargetResFile);
        if (gPrintRecordHdl != NULL) {
          /* Now use the print record to see if the information we */
          /* saved was preserved...                                 */
          TestPrintRecord(gPrintRecordHdl);
        } else
          ReportError(MemError());
    } else
        ReportError(MemError());

    /* Kill the print record (if it was created) and go home...   */
    if (gPrintRecordHdl != NULL)
        DisposHandle((Handle)gPrintRecordHdl);
}
```

There are several points to remember when using this technique. Use a resource type not used by the Printing Manager so it doesn't become confused. Types to avoid include '`PREC`', '`PDEF`' and '`POST`'. Remember that lowercase resource types are reserved for use by Apple. You also should not make assumptions about the size of the record. Use `GetHandleSize` if you really need to know. This allows for the record to grow in size in the future. Finally, when rereading the record from your document, be sure to pass it to `PrValidate` before using it in case the user has changed printers or print drivers since last printing the document.

USING POSTSCRIPT

Some applications prefer to bypass QuickDraw and print using PostScript instead. This often results in poor or nonexistent support for printers such as the ImageWriter and LaserWriter II SC. It also means relying on a method for determining which printer is in use, such as checking the `wDev` field in the `TPrStl` record.

One method for printing PostScript without relying on the type of printer being used is using the `TextIsPostScript PicComment`:

```
PicComment (PostScriptBegin, 0, NIL);
PicComment (TextIsPostScript, 0, NIL);
DrawString (ThePostScript);
PicComment (PostScriptEnd, 0, NIL);
```

The problem with this technique is that because non-PostScript printers ignore the TextIsPostScript PicComment, `DrawString`, which is a QuickDraw procedure, literally sends `ThePostScript` to the printer, resulting in garbage being printed. A better technique is using the `PostScriptHandle PicComment`. Because this comment is only understood by PostScript drivers, it avoids the QuickDraw/ PostScript interaction just described:

```
PicComment (PostScriptBegin, 0, NIL);
PicComment (PostScriptHandle, GetHandleSize (ThePostScript),
    ThePostScript);
PicComment (PostScriptEnd, 0, NIL);
```

Further problems occur with applications that never print using QuickDraw but only use PostScript. Some versions of the LaserWriter® driver assume that if they see no QuickDraw, nothing was printed on the page and no output occurs. This can be avoided by embedding some nonprinting QuickDraw in your code. Immediately after calling `PrOpenPage`, issue the following calls:

```
PenSize (0,0);
MoveTo (10, 10);
Line (0,0);
PenSize (1,1);
```

This technique also solves a problem with background printing. In this case, the Printing Manager starts off each page with an empty default clipping region. Without seeing any valid QuickDraw calls, this region is never altered and your nice PostScript output is clipped entirely off the page. For more details on printing, see the article on "The Perils of PostScript" in this issue.

FONTS

System 7.0 will introduce an alternate way of dealing with fonts. While this new technology won't cause problems for most applications, you should be aware of a few issues. Any application that allows user font selection will be affected by the new outline font technology. The most obvious feature is that any size font is now available. That means a list of point sizes in a menu is no longer sufficient. If you currently combine font selection and font size selection in a dialog box, be sure to include an editable field that allows the user to type in any point size. If you now have a list of common sizes in a menu, include an “Other…” menu item that displays a similar dialog box with an editable field.

Since Apple introduced the LaserWriter, there has been a problem about where to get font metrics. The most compatible method is simply to call `FontMetrics` and read the metrics from the width table. For one reason or another, however, applications have seen fit to read metrics directly from the '`FOND`' resource. The addition of outline fonts adds another layer of complexity. Outline fonts will store metric information in the '`sfnt`'. Accessing metrics in the '`FOND`' could give invalid data. If you are currently accessing the '`FOND`' directly, you will have to revise to take advantage of '`sfnt`'s.

INTERNATIONAL SUPPORT

You can greatly expand the market for your product if you do not make assumptions about your user’s language. Following a few simple rules can make your application much easier to localize. Don’t simply assume, like many C programmers, that a character is one byte. Using the C routine `strcmp`, for example, to sort strings can give completely wrong results in languages other than English. Use `IUCompStr` instead. Determine the local conventions for decimal point, thousands separator, list separator, and time cycle from the appropriate international resource when performing input and output. Script Manager 2.0 routines, if available, can make this even easier by doing the right thing for you automatically. For example, the `Str2Format` routine can take input in one language and convert it to a canonical form that can be used by `Format2Str` to output the string for a user in a completely different country.

LOWER-LEVEL ISSUES

Higher-level issues, such as the ones just discussed, are likely to affect all applications. But a lot of code that gets written needs to work at a lower level—either accessing memory in strange ways or depending on tricks in assembly language, for example. The remainder of this article will take a look at some of those issues.

LOW MEMORY GLOBALS

Applications should avoid reliance on low-memory globals. In particular, undocumented low-memory globals must be avoided since they are most likely to change. But even dependence on well-known globals can be avoided. For example, the `TickCount` trap returns the same value as the low memory global `Ticks`. `TickCount` is supported under A/UX, while `Ticks` is not, so use of the trap guarantees compatibility. In general, if a trap is available, always use it. And if a glue routine is available, you should use it as well. Then if a change is necessary, you need only update your development system and recompile to implement the change. For the same reason, use of glue routines is also good advice for assembly-language programming.

There is an exception to this rule. The Journaling Driver (see IM I-261) patches key Event Manager traps : `GetMouse`, `Button`, `GetKeys`, and `TickCount`. The Journaling Driver is now used exclusively by MacroMaker™, and unfortunately the driver's patches are not reentrant. This means you cannot safely use these traps in interrupt or VBL code. If you experience strange system hangs only when MacroMaker is installed, this is probably the cause and your code should instead reference the appropriate low-memory globals for the information you need.

SELF-MODIFYING CODE

Applications that use self-modifying code can present serious compatibility issues. There are two kinds of self-modifying code. The first kind involves actually changing machine instructions on the fly. Such code, popular in copy protection schemes, crashes and burns on Macintoshes that use an instruction cache. For example, after a sequence of instructions has been executed and cached by the Macintosh II, some code comes along, modifies the original instructions, and tries to execute them again. But the CPU says, "Ah ha! I already know what these instructions are" and tries to execute the cached instructions, which is not what the programmer originally intended. Fortunately, the Macintosh II and natural selection have made such self-modifying code virtually extinct.

A second, subtler form of self-modifying code keeps variables in the code segment itself. A typical example is the use of `DC.W` or `DC.L` directives to allocate variables in the same segment as the actual code. Such code avoids the earlier problem because it is not actually modifying instructions. The catch is that future operating systems may make 'CODE' segments read-only, and when that code tries to write to its variables, it will fail. Of course, read-only use of such data, such as storing string constants within code segments, is valid. It's fine to do this when no alternative is available. You won't crash in the foreseeable future.

A variety of small tasks, such as VBL tasks and completion routines, run asynchronously on the Macintosh. Because they are executed asynchronously, they cannot be assured that register A5, which by convention points to the application's global variables, is valid when they are called. A common technique used in this case was to store a copy of A5 in with the code so these routines could use the saved value to access global variables.

It's possible to avoid such self-modifying code, as the following MPW sample code illustrates. The trick here is that in creating a VBL task you must pass a record describing the task to the system. When the VBL task is invoked, the system sets up register A0 to point to the start of this record. While the record itself does not contain storage for A5, it's simple to embed the VBL task record into a larger record, or in this case a C struct, that does have room for A5, or anything else you deem important, such as a handle. An inline function called at the start of the VBL task converts A0 into a pointer to the record. Then the task can access anything it needs.

```
#include <Events.h>
#include <OSEvents.h>
#include <OSUtils.h>
#include <Dialogs.h>
#include <Packages.h>
#include <Retrace.h>
#include <Traps.h>

#define  INTERVAL    6
#define rInfoDialog  140
#define rStatTextItem      1

/*
 *  These are globals which will be referenced from our VBL Task
 */
long     gCounter;   /* Counter incremented each time our VBL gets called */

/*
 *  Define a struct to keep track of what we need.  Put theVBLTask into the
 *  struct first because its address will be passed to our VBL task in A0
 */
struct VBLRec {
   VBLTask           theVBLTask;       /* the VBL task itself */
   long  VBLA5;      /* saved CurrentA5 where we can find it */
};
typedef struct VBLRec VBLRec, *VBLRecPtr;

/*
 * GetVBLRec returns the address of the VBLRec associated with our VBL task.
 *  This works because on entry into the VBL task, A0 points to the theVBLTask
 *  field in the VBLRec record, which is the first field in the record and
 *  is the address we return.  Note that this method works whether the VBLRec
 *  is allocated globally, in the heap (as long as the record is locked in
 *  memory) or if it is allocated on the stack as is the case in this example.
 *  In the latter case this is OK as long as the procedure which installed the
 *  task does not exit while the task is running.  This trick allows us to get
 *  to the saved A5, but it could also be used to get to anything we wanted to
 *  store in the record.
  */
```

```
VBLRecPtr GetVBLRec ()
     = 0x2008;             /* MOVE.L  A0,D0 */

/*
 *  DoVBL is called only by StartVBL ()
 */
void DoVBL (VRP)
VBLRecPtr  VRP;
{
     gCounter++;                      /* Show we can set a global */
     VRP->theVBLTask.vblCount = INTERVAL; /* Set ourselves to run again */
}

/*
 *  This is the actual VBL task code.  It uses GetVBLRec to get our VBL record
 *  and properly set up A5.  Having done that, it calls DoVBL to increment a
 *  global counter and sets itself to run again.  Because of the vagaries of
 *  MPW C 3.0 optimization, it calls a separate routine to actually access
 *  global variables.  See Tech Note #208 - "Setting and Restoring A5" for the
 *  reasons for this, as well as for a description of SetA5.
 */
void StartVBL ()

{
     long       curA5;
     VBLRecPtr  recPtr;

     recPtr = GetVBLRec ();           /* First get our record */
     curA5 = SetA5 (recPtr->VBLA5);   /* Get the saved A5 */

                   /* Now we can access globals */
     DoVBL (recPtr);                  /* Call another routine to do actual work */

     (void) SetA5 (curA5); /* Restore old A5 */
}
```

```
/*
 *  InstallVBL creates a dialog just to demonstrate that the global variable
 *  is being updated by the VBL Task.  Before installing the VBL, we store
 *  our A5 in the actual VBL Task record, using SetCurrentA5 described in
 *  Tech Note #208.  We'll run the VBL, showing the counter being incremented,
 *  until the mouse button is clicked.  Then we remove the VBL Task, close the
 *  dialog, and remove the mouse down events to prevent the application from
 *  being inadvertently switched by MultiFinder.
 */
void InstallCVBL ()
{
   VBLRec       theVBLRec;
   DialogPtr    infoDPtr;
   DialogRecord           infoDStorage;
   Str255       numStr;
   OSErr        theErr;
   Handle       theItemHandle;
   short        theItemType;
   Rect         theRect;

   gCounter = 0;           /* Initialize our global counter */
   infoDPtr = GetNewDialog (rInfoDialog, (Ptr) &infoDStorage, (WindowPtr) -1);
   DrawDialog (infoDPtr);
   GetDItem (infoDPtr, rStatTextItem, &theItemType, &theItemHandle, &theRect);

   /*
    *  Store the current value of A5 in the MyA5 field.  For more
    *  information on SetCurrentA5, see Tech Note #208 - "Setting and
    *  Restoring A5".
    */
   theVBLRec.VBLA5 = SetCurrentA5 ();
   /* Set the address of our routine */
   theVBLRec.theVBLTask.vblAddr = (VBLProcPtr) StartVBL;
   theVBLRec.theVBLTask.vblCount = INTERVAL; /* Frequency of task, in ticks */
   theVBLRec.theVBLTask.qType = vType;   /* qElement is a VBL task */
   theVBLRec.theVBLTask.vblPhase = 0;
```

```
        /* Now install the VBL task */
        theErr = VInstall ((QElemPtr)&theVBLRec.theVBLTask);

        if (!theErr) {
             do {
                 NumToString (gCounter, numStr);
                 SetIText (theItemHandle, numStr);
             } while (!Button ());
             theErr = VRemove ((QElemPtr)&theVBLRec.theVBLTask);
                     /* Remove it when done */
        }

        /* Finish up */
        CloseDialog (infoDPtr);    /* Get rid of our dialog */
        FlushEvents (mDownMask, 0); /* Flush all mouse down events */
    }
```

PRIVILEGED INSTRUCTIONS

Under the current Macintosh operating system, the CPU operates in the supervisor state and applications are allowed to use any and all 680x0 instructions, with the lone exception of the Test And Set (`TAS`) instruction, which is not supported by the hardware. The A/UX operating system forces applications to run in the user state, and applications that use privileged instructions reserved for the supervisor state will fail. Examples of such instructions are `MOVE`, `ANDI`, and `EORI` instructions with the status register (SR) as either the source or the destination. Typically, these instructions are used to alter the condition code register (CCR), which is the low byte of the SR. Using these instructions with the CCR as the source or destination instead of the SR will accomplish the same thing without causing your application grief. Certain floating point instructions such as `FSAVE` and `FRESTORE` are also privileged and should be avoided. As we mentioned, A/UX does not allow the use of privileged instructions and is a good test of compatibility in this case.

DIRECT HARDWARE ACCESS

If you think you need direct access to hardware, let Apple know. It may be acceptable on other personal computers to access hardware directly, but it is decidedly anti-social on the Macintosh and absolutely verboten under operating systems with multi-user protection like A/UX. Beware of schemes for copy protection or performance enhancement that rely on direct hardware access. Macintosh hardware has changed in the past, and it will change in the future. Each new machine may mean yet another revision of your application.

TRAP PATCHING

Trap patching is very useful for overriding or enhancing system trap handling. It is used by the system, for example, to correct errors in the Macintosh ROM. Many applications also use it to provide additional functionality. Because it is very difficult to anticipate all the possible side effects of your patch, maintaining compatibility is difficult, too. Before writing a patch, you should decide if it's absolutely essential. Often the results you need can be achieved without the patch.

Suppose, for example, you decide to patch `ExitToShell`. This may sound like an excellent way for your program to get one last chance at closing files or doing whatever other cleanup is necessary before exiting. Whether `ExitToShell` is called in response to a user's Quit command or because of some fatal error condition, your patch would always have a chance to clean up. But rather than having `ExitToShells` all over your code, you could achieve the same result by calling a single, common exit routine that performed the cleanup and then called `ExitToShell`.

If you absolutely must trap patch, here are some general guidelines. Don't make assumptions about the format of the trap dispatch table. In particular, don't try to read or write entries in the trap dispatch table directly—use `GetTrapAddress` and `SetTrapAddress` instead. If your patch only applies to your application, install it in your application heap. Otherwise, install it in the system heap. Application heap patches will be swapped out by MultiFinder when your application is switched out. Because system heap patches will apply to all applications that use the trap, use them only when absolutely necessary.

You cannot assume that a valid A5 world exists when your patch is invoked. Register A5 points to the base of an application's global variables, and A5 world refers to an application's global address space. MultiFinder maintains different A5 worlds for each running application. Your patch cannot assume when it is called that A5 points to your application's global variables. If it needs access to global variables, you must save a copy of A5 before installing your patch. Then the patch needs to preserve the current value of A5, set the saved value, and restore the original A5 on exit. (See Technical Note #208.) Your patch should avoid use of the Memory Manager if the trap could be invoked at interrupt time or if memory could move during your patch.

Finally, you must not tail patch. In a normal patch, your code completes its task and then invokes the standard trap code to complete the patch. In a tail patch, your code regains control after the standard trap code completes. The problem with this technique is that many of the ROM patches are themselves tail patches, and they rely on knowledge of the caller to accomplish their task. If the ROM patch expects to be called from a ROM address, but is instead called by your patch code, it can become confused. If you JSR to invoke the standard trap code, then you are tail patching. The correct way is to JMP to the starting address of the code.

IN CONCLUSION

It may be useful to know that Apple's implementation of Unix, A/UX, offers a major test for compatibility with System 7.0. A/UX provides a very different environment for Macintosh applications, but applications that follow the compatibility guidelines work without alteration under A/UX. If your application works correctly under A/UX, it stands a very good chance of working correctly under System 7.0.

If you've gotten this far, you are likely to avoid Johnny Appledweeb's fate. You obviously are seriously concerned for your customers and willing to go that extra step to minimize future compatibility problems. It may seem at times that Apple goes out of its way to stretch its own rules, but that is not the case. It is simply impossible to foresee all future hardware and software changes. Incompatibility is unfortunately an ongoing battle. Your part of that battle goes beyond this article and requires you to keep abreast of changes as Apple announces them.

DEBUGGING DECLARATION ROMS

Through system software, the Macintosh can read the declaration ROMs in NuBus and pseudo-NuBus slots, like those in the Macintosh SE/30. This article tells you what you must know about NuBus addressing and the structure of correct declaration ROMs to successfully debug the ROM. It walks you through the structure of an example declaration ROM and gives common errors and strategies for debugging declaration ROMs.

The Slot Manager's flexibility in providing a layer between the hardware and higher-level software benefits developers and customers alike. Users can easily expand the Macintosh II family and the Macintosh SE/30 with additional hardware that goes in slots. The Macintosh card architecture lets them plug new cards into the Macintosh without worrying about using the right slot, setting dip switches, or running system configuration software. As a developer, you may need to know more about the architecture that makes this self-configuring environment possible.

THE SLOT MANAGER AND DECLARATION ROMS

Part of the Macintosh operating system, the Slot Manager can find the ROM on each expansion card installed in a system and identify the card's special capabilities. It makes use of predefined data structures called slot Resources (`sResources`) to initialize and configure a card and report the card's location. Each card installed in a Macintosh expansion slot needs a declaration ROM, also known as a configuration ROM, with information for using the card's hardware. The expansion hardware could be as simple as a memory card that needs to publish its address ranges or as complicated as a video card with initialization code, a driver, and declaration data.

In addition to letting the system determine what hardware is available, the Slot Manager frees applications from being dependent on a particular type of hardware. In other words, the Slot Manager helps insulate an application from the hardware by being able to locate underlying, intermediate driver software that will know about and talk to the specific hardware. The application can be free of the details and need only deal with higher-level functions. The degree of insulation depends on the software and data structures in the declaration ROM.

MARK BAUMWELL is a low-level O/S sort of guy. He started with Apple in 1981 after a stint with Zilog as a test engineer. After three years in the Lisa division, Mark made the move to Macintosh DTS where he has fulfilled his lifelong dream of being a firefighter. He professes to be "outdoorsy," and getting an airplane pilot's license is his

The Slot Manager:

- locates and lists the cards with declaration ROMs.
- defines a uniform structure for information in the declaration ROM and a set of library routines to access the information.
- includes routines to allow host applications to transparently access information in the ROMs without regard to NuBus or byte lanes.
- allows ROM initialization code to run on the host CPU during the host's startup sequence.
- allows cards to have drivers from their declaration ROMs loaded into the host CPU.
- initializes and manipulates the parameter RAM on the host CPU for the card during startup.

When applications are insulated from particular hardware implementations, they don't have to be revised for each new version of a vendor's board, or even for compatible boards made by competitors. Besides reducing maintenance work for the developer, information hiding of this sort saves wear and tear on customers.

Suppose a customer owns an application and a card and happens to buy another board, from the same or a different vendor, with even slightly different hardware. The difference might be a change in address or meaning of some register or memory location. The customer has to mix and match applications or drivers or INITs to boards. This is not very Macintosh-like, and the board manufacturer is sure to be savaged by the customers and the press. Matching various card-specific versions of software and different revisions of hardware can be a headache for distributors and dealers. Including card-specific software on each card's ROM in a universally accessible structure greatly simplifies installation and maintenance.

USING SRESOURCES

Don't confuse `sResources` on expansion cards with standard Macintosh resources. The small s indicates a slot resource as opposed to a real Macintosh resource. Although related conceptually, `sResources` are different and may contain anything from code to data—for example, icons, special fonts, or vendor-defined data. In fact, feel free to substitute "data structure" for "`sResource`" as you read.

Each card has one special `sResource` called a board `sResource` and usually one additional `sResource` for each function the card can perform, although additional supporting `sResources` are possible. An `sResource` affiliated with a function is called a functional `sResource` and gives information about that particular

current passion. He claims that nothing we could say would sully his reputation more than it has already been. We tried, but Apple Legal wouldn't approve it. Would you fly with this guy? •

function, usually to high-level applications interested in accessing the function.

Most cards perform only one function. For example, a modem card might perform only a modem function, a video card might just do video, and so on. These cards would have only one functional `sResource` , along with the required board sResource. However, it is possible to build a multifunction expansion card—a card with a parallel, serial, and modem port, for example. In this case, the card's declaration ROM would have three functional `sResources` —one each for the card's functions. In addition to other, optional `sResource`s, it would also have the required board `sResource`.

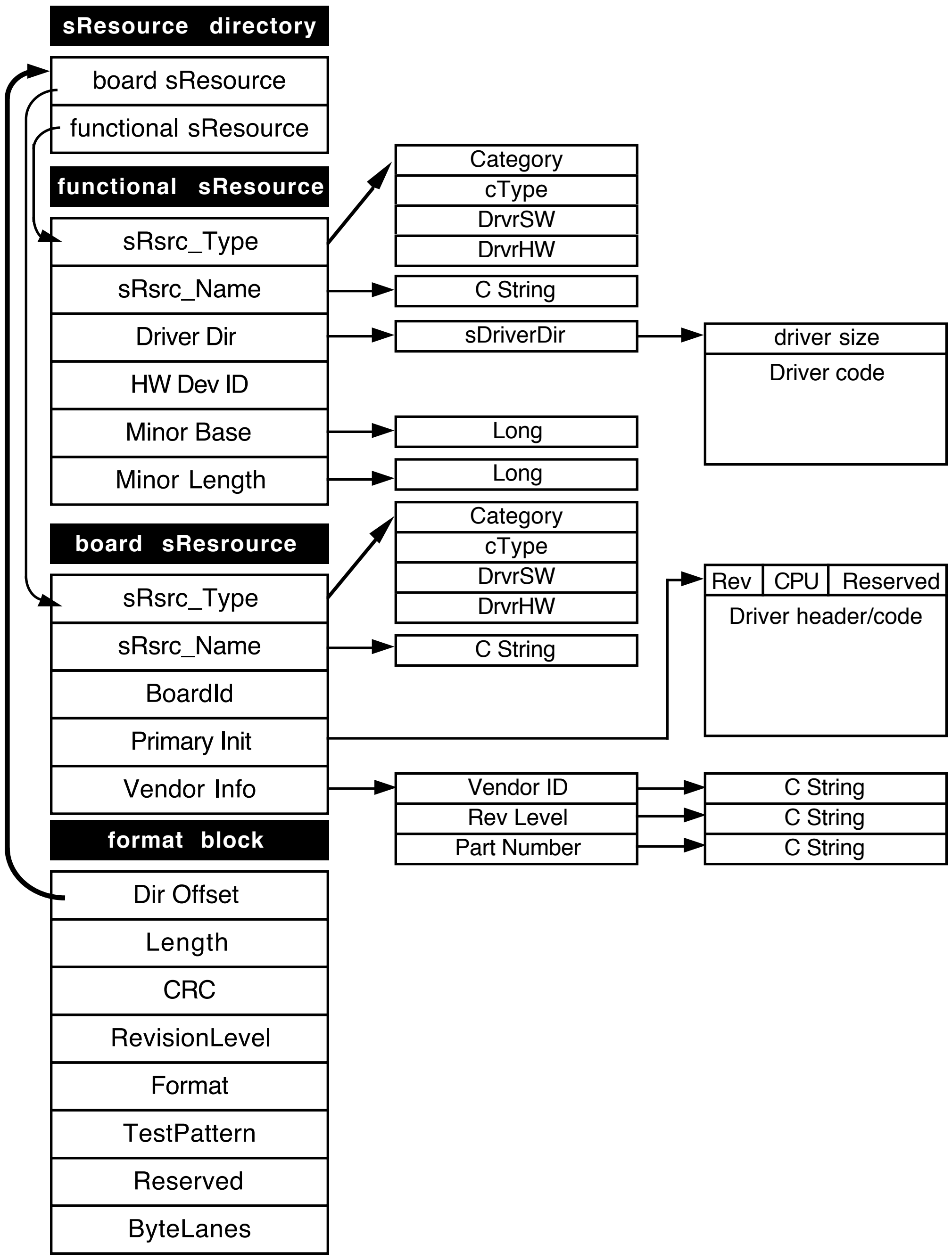
A high-level program may need to be able to find and use certain kinds of hardware in the Macintosh slots. For example, QuickDraw works with video cards made by different vendors. QuickDraw finds each video card by looking for the card's functional sResource that says it can do QuickDraw-compatible video.

AN EXAMPLE DECLARATION ROM

As far as the Slot Manager is concerned, at the startup scan of the cards a valid declaration ROM must have proper structures for the format block, `sResource` directory, and the board `sResource` . If any of these structures is in error, the Slot Manager marks the slot as empty, and no Slot Manager calls to that slot will work. Though other `sResources` or data structures may have errors, the Slot Manager doesn't check them at startup. These errors will show up during later calls to the Slot Manager by applications, INITs, drivers, and so forth.

We will look at these key structures in an example declaration ROM and will discuss some common errors developers make. The sample skeleton ROM has the required board `sResource` and one functional `sResource`.

The ROM can be divided into four major structures: the `sResource` directory, functional `sResource` , board `sResource` , and format block, as shown in the illustration. Let's look at these structures in more detail.


sResource directory
board sResource
functional sResource
functional sResource
sRsrc_Type
sRsrc_Name
Driver Dir
HW Dev ID
Minor Base
Minor Length
Category
cType
DrvrSW
DrvrHW
C String
sDriverDir
driver size
Driver code
Long
Long
board sResrource
sRsrc_Type
sRsrc_Name
BoardId
Primary Init
Vendor Info
Category
cType
DrvrSW
DrvrHW
C String
Rev
CPU
Reserved
Driver header/code
Vendor ID
Rev Level
Part Number
C String
C String
C String
format block
Dir Offset
Length
CRC
RevisionLevel
Format
TestPattern
Reserved
ByteLanes

THE SRESOURCE DIRECTORY

The source code begins with some includes and equates, followed by the `sResource` directory. A directory is a list of the `sResources` in the ROM. In the example, we have two sResources, the required board `sResource` and one functional `sResourc`. The directory looks like this:

```
_sRsrcDir     OSLstEntry      sRsrcBoard,_sRsrcBoard
                              ;References Board sResource.
              OSLstEntry      sRsrcFun,_sRsrcFun
                              ;References functional sResource.
              DatLstEntry     endOfList,0
                              ;End of the list.
```

The `OSLstEntry` and `DatLstEntry` items are assembler macros, defined in the MPW `ROMEqu.a` file. These macros make creating declaration ROMs easier, since most declaration ROM structures fall into two different formats:

- an ID byte followed by three bytes representing a 24-bit offset or
- an ID byte followed by a 24-bit data value

A directory contains both of these formats. The first format is used for all `sResource` entries in a directory. Each `sResource` entry consists of one byte containing the `sResource` identification number, and three bytes containing the offset to the `sResource` itself.

The offset list entry (`OSLstEntry`) macro is used to conveniently calculate and fill in these types of entries. It takes two arguments: the ID byte and a label designating the destination. The macro puts the first argument as is into the high byte, calculates the 24-bit signed offset value to the destination label, and puts it into the next three bytes. In our example, the first entry of the directory looks like this:

```
_sRsrcDir   OSLstEntry   sRsrcBoard,_sRsrcBoard    ;References Board
                                                   ;sResource.
```

The `_sRsrcDir` label designates the start of the directory. This label is needed because the offset to the directory will be calculated later. The first argument of the macro, `sRsrcBoard`, is equated to 1 (in the equates near the top of the source code file), and so a $01 will be put into the first byte. The second argument, `_sRsrcBoard`, is the label designating the start of the board `sResource`. The macro calculates the offset from the present point in the macro to the label and puts the resultant offset in the next three bytes. The `_sRsrcBoard` structure is `$000C` bytes away from the directory entry, so the offset is `$000C`. Putting them together, the complete directory entry for the board sResource looks like this in hexadecimal:

```
   $01000C
```

A similar calculation for the functional sResource is done with the offset from the directory to the _sRsrcFun label.

The second format is used in many places in declaration ROMs. It is commonly used for the end-of-list entry, which marks the end of the list of directories and sResources. This entry always has an ID byte value of $FF followed by three bytes of zero. It can also be used to hold small pieces of data that fit into three bytes or less.

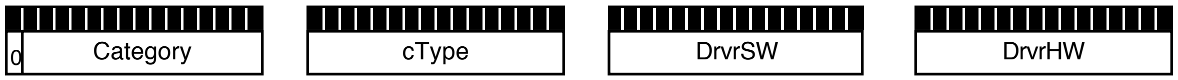
It is convenient to use the data list entry (DatLstEntry) macro for these types of entries. DatLstEntry is similar to OSLstEntry but simpler. It takes two arguments: the ID byte and the desired data value. It puts the first argument into the high byte and the data value into the next three bytes.

SRESOURCES IN GENERAL

Before looking at our example sResources, let's examine the structure of sResources in general. Every sResource includes an sRsrc_Type entry whose fields identify the sResource. Applications and drivers use the sRsrc_Type entry of each sResource to identify it and the function it performs.

The sRsrc_Type entry is comprised of an ID byte (always a $01), followed by an offset to the sResource type format. For this discussion, we will only look at the Apple defined format for the type format entry, indicated by the leading bit being a zero, since virtually all developers use it.

The sResource type format is a 64-bit value, separated into four fields of 16 bits each. The entry looks like this:



The type format fields have constant, fixed values for the board sResource, so let's look at the values for the more general case of functional sResources. The type format is hierarchical in nature, and the four fields can be considered to be "nested" under each other, with the Category being at the top of the hierarchy. While some of the fields have been predefined, new values can be and often are defined to suit developers' products.

A board can perform broad categories of possible functions, which are represented by the Category field of the type format. Within each Category are subset types that are represented by the cType value. Nested farther in the hierarchy are subset software driver identifiers (the DrvrSW value). Finally, under each DrvrSW entry, there are hardware identifiers (the `DrvrHW` value). The hierarchical relationship looks like:

```
Category
      cType
            DrvrSW
                  DrvrHW
```

A given `Category` can have multiple `cType` interfaces for it, and each of those `cTypes` can have its own nested, underlying software interfaces. Many different pieces of hardware can belong to a given software architecture. Equates for many categories have been already defined, such as Display, Network, Communication, and CPU. Further, subtypes for some of these common categories have been defined, as well as software interfaces to go with some subtypes.

Let's see how this works with a common family of cards: video cards. A category for display functions has been defined (`CatDisplay EQU $0001`). Under it, a subtype for video displays has been defined (`TypVideo EQU $0000`). Since Apple has defined a driver and firmware interface for video display cards that are QuickDraw compatible, there is a software driver definition as well (`DrSWApple EQU $0001`). Now let's say a developer wants to make a QuickDraw-compatible video card — the Amalgamated VaporWare Widget video board. The developer gets a hardware identifier from Developer Technical Support—let's say DTS assigns the developer `DrHwWidget EQU $4321`—and creates a functional `sResource` with an `sRsrc_Type` of

```
      CatDisplay     EQU $0001
      TypVideo       EQU $0000
      DrSwApple      EQU $0001
      DrHwWidget     EQU $4321
```

or in the complete type format:

```
      0001000000014321
```

Now QuickDraw will recognize the card, because it looks for type formats that match `CatDisplay/TypVideo/DrSwApple`. During the search, QuickDraw will only look for a match down to its software architecture level and will mask off the hardware identifier. It does not care about the hardware identifier, because it knows the driver will deal with the underlying hardware. Notice that even though the first three entries in the type format will be the same for all QuickDraw-compatible video cards, the different hardware identifiers will make the entries unique. This is useful for the driver of the Widget card, which very much cares about the underlying hardware. It will want to locate the card and will do so by doing a match of the whole type format, including the hardware identifier.

Developers can take advantage of this if they want to have applications use their software/hardware architecture. By publishing the software interface and type format values, a developer can make a board that others can write applications for.

Besides the `sRsrc_Type` entry, `sResources` must also have a name entry (`sRsrc_Name`), which contains an ID byte (always `$02`), followed by an offset to a null-terminated string (a `C string`). In addition to these, the board `sResource` must have a board ID value (`BoardId`). All other entries defined for `sResources` and the board `sResource` are optional. .

Now, let's take a look at the two `sResources` in detail.

THE BOARD SRESOURCE

Like the directory entries, our board sResource uses the macros we discussed earlier to calculate and fill in the various entries. Labels such as `sRsrcType` and `sRsrcName` are defined in the MPW `ROMEqu.a` file. Others, such as the board ID, are in the declaration ROM source code. The first part of our board `sResource` looks like this:

```
;==============================================================
;                     The Board sResource
;==============================================================
_sRsrcBoard    OSLstEntry     sRsrcType,_BoardType
                              ;References Rsrc_Type entry
               OSLstEntry     sRsrcName,_BoardName
                              ;References Rsrc_Name entry
               DatLstEntry    boardId,TheBoardId
                              ;boardId **ASSIGNED BY MACDTS**
               OSLstEntry     primaryInit,_sPInitRec
                              ;Refs Primary init record.
               OSLstEntry     vendorInfo,_VendorInfo
                              ;References Vendor info list.
               DatLstEntry    endOfList,0
                              ;End of the list.
```

The `_sRsrcType` entry for the board `sResource` points to the board `sResource` type format. The type format is always the same for the board `sResource`—that's how the board `sResource` is identified. The type format for the board `sResource` always has this definition: .

```
_BoardType  DC.W  CatBoard    ;ALWAYS $0001 for bd sResource
            DC.W  TypBoard    ;ALWAYS $0000 for bd sResource
            DC.W  DrSwBoard   ;ALWAYS $0000 for bd sResource
            DC.W  DrHwBoard   ;ALWAYS $0000 for bd sResource
```

Put together into the full 64 bits, it looks like this:

```
        $0001000000000000
```

The `_sRsrcName` entry points to the name string (a `C string`), which should be the official product name of the board:

```
_BoardName           DC.L        'OFFICIAL PRODUCT NAME'
;The name of the Board - should be official product name
```

(At the beginning of the source code, there is a STRING C directive, to automatically generate c strings.)

After the name comes the required board ID, which, being a 16-bit value, can be filled in using a `DatLstEntry` macro. Board IDs are assigned by Macintosh DTS. To get board ID, contact Macintosh DTS with the following information:

- the company name and address (mailing and electronic addresses, if possible)
- the name of the person in the company responsible for the board (and a phone number, if possible)
- the functions the board will perform
- the official product name for the board (or a code name)
- whether or not the board will have a software driver other than one that has been predefined (like Apple's video driver)
- whether or not the driver will be in ROM

DTS will assign the board ID and any necessary functional `sResource` information. This information goes into a database, which is kept strictly confidential. There is a HyperCard® stack on the Developer Services CD and on AppleLink that makes sending in this information easier.

Next, the board `sResource` contains an entry for the primary initialization code. We have defined one, but it is in a separate file called `PrimaryInit.a`, which is referenced with an INCLUDE directive:

```
;--------------------------------------------------------------
;             Primary Init Record (if needed)
;--------------------------------------------------------------
_sPInitRec    DC.L      _EndsPInitRec-_sPInitRec
                        ;physical Block Size
              INCLUDE   'PrimaryInit.a'
                        ;Primary Init Code
_EndsPInitRec EQU       *     ;End of block
```

```
              STRING    C     ;Restore to 'c' string type.
```

The following is optional vendor data. It is up to the developer to decide what, if anything, goes in the VendorInfo entries. This example shows the way Apple typically uses the vendor information entries.

```
;---------------------------------------------------------------
;             Vendor Information record
;---------------------------------------------------------------
_VendorInfo  OSLstEntry   VendorId,_VendorId   ;References
                                ;the Vendor
             ;Id.
             OSLstEntry   RevLevel,_RevLevel   ;References
                                ;the Revision
             ;Level.
             OSLstEntry   PartNum,_PartNum     ;References
                                ;the Part
             ;Number.
             DatLstEntry  endOfList,0          ;End of the
                                ;list.
_VendorId    DC.L         'COMPANY NAME'       ;The Vendor
                                ;Id.  Most
             ;vendors use
                                               ;company name
_RevLevel    DC.L         'Release-1.0'        ;The Revision
                                               ;Level
_PartNum     DC.L         '12-3456'            ;The Part
                                ;Number
```

THE FUNCTIONAL SRESOURCE

In our example, our card has only one function, so our ROM has just one functional sResource. For this example, we have defined a nonexistent set of Category, subtype, software, and driver identifiers, which normally would be replaced by the ones assigned by DTS. The functional sResource entry looks like this:

```
;===============================================================
;             The Functional sResource
;===============================================================
_sRsrcFun     OSLstEntry     sRsrcType,_FunType
                             ;References sRsrc_Type
              OSLstEntry     sRsrcName,_FunName
                             ;References sRsrc_Name
              OSLstEntry     sRsrcDrvrDir,_FunDrvrDir
                             ;References sResource driver dir
```

```
               DatLstEntry     sRsrcHWDevId,1
                               ;The hardware device Id.
               OSLstEntry      MinorBaseOS,_MinorBase
                               ;References Minor Base Offset.
               OSLstEntry      MinorLength,_MinorLength
                               ;References Minor Base Length.
               DatLstEntry     endOfList,0
                               ;End of the list.
```

The type format for our fictional function looks like this:

```
_FunType       DC.W            CatExCat       ;<Category>
               DC.W            TypExTyp       ;<Type>
               DC.W            DrSwExSw       ;<DrSw>
               DC.W            DrHwExHw       ;<DrHw>
```

The `sRsrc_Names` for functional `sResources` follow a convention of concatenating the equates for the `sRsrc_Type` but stripping off the prefixes and separating the type format fields by underscore characters. Since our type is `CatExCat/TypExType/DrSwExSw/DrHwExHw`, the `sRsrc_Name` becomes:

```
_FunName       DC.L            'ExCat_ExType_ExSW_ExHW'
```

The driver directory identifies the type of driver and the driver itself. In the example, the driver is compatible with the Macintosh OS but contains Motorola 68020 code. The driver itself is in a separate file and is referenced by an INCLUDE directive.

```
;-------------------------------------------------------------
;              Driver directory (if there's an on-board driver)
;-------------------------------------------------------------
_FunDrvrDir    OSLstEntry      sMacOS68020,_sMacOS68020
                               ;References Macintosh-OS
                               ;68020 driver.
               DatLstEntry     endOfList,0
                               ;End of the list.
                               ;Driver-1 (68020).
_sMacOS68020   DC.L            _End020Drvr-_sMacOS68020
                               ;The physical Block Size
               INCLUDE         'NameofDrvrSrcCodeFile.a'
                               ;The  driver code
_End020Drvr    EQU             *
                               ;The end of the driver.
```

```
                    STRING          C
```

The hardware device ID field (HWDevID) is optional and defined by the vendor. It can be used to indicate that an sResource is associated with a particular piece of hardware. This would be used in the case of cards that had multiple "hardware areas"—say, multifunction cards—that could be considered to be separate hardware devices. The field could be used to group certain sResources with the various devices. In this case, the functional sResources would have different HWDevID values depending on which hardware device on the card they describe.

For example, you have a card with two serial ports, which you label port 1 and port 2. You have three functional sResources—an asynchronous serial sResource, a MIDI sResource, and a network sResource. Let's say the async serial sResource is assigned to port 1. It is assigned HWDevID=1. Now let's say the network sResource can only be used with port 2. It is assigned HWDevID=2. Similarly, the MIDI sResource can only be used on port 1. It will also be assigned HWDevID=1. Now, by looking at the HWDevID fields, a driver or card software can tell which piece of hardware it is using in case different hardware on the card has different characteristics it must handle. If an sResource does not describe a hardware device, then the HWDevID field may be omitted.

The MajorBaseOS, MinorBaseOS, MajorLength, and MinorLength fields describe where the hardware area starts and how large it is. For example, a video sResource might have the MinorBaseOS be an offset to the starting address of the video frame buffer. The MinorLength field would tell how large it is. Other cards might use the MinorBaseOS to indicate where its hardware control registers are.

Use Major vs. Minor depending whether you want to reference the area using super slot space or NuBus slot space addresses.

In the example, let's say this function has some RAM memory in NuBus slot space that we would like to reference:

```
_MinorBase     DC.L      defMinorBase        ;RAM Offset
_MinorLength   DC.L      defMinorLength      ;RAM length
```

THE FORMAT BLOCK

Declaration ROMs are recognized by the presence of the ROM's format block, which occupies the highest address of the ROM's slot address space. This is our example format block:

```
ORG   ROMSize-FHeaderRec.fhBlockSize
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;     Format Block
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      DC.L    (_sRsrcDir-*)**$00FFFFFF
              ;Offset to sResource directory
      DC.L    ROMSize
              ;Length of declaration data
      DC.L    0
              ;CRC-can be patched by MPW crc tool
      DC.B    romRevision
              ;Revision level
      DC.B    AppleFormat
              ;Format
      DC.L    TestPattern
              ;Test pattern
      DC.B    0
              ;Reserved byte (must be zero)
      DC.B    $E1
              ;ByteLanes: 1110 0001 (bytelane 0)
ENDP
END
```

The first entry calculates the offset to the beginning of the directory, using the directory label. This must be a signed 24-bit value.

The ROM size, revision level, format, test pattern, and reserved values are declared in the source and in the included MPW `ROMEqu.a` file. The CRC value can be generated and patched in by utilities such as MPW tools. Apple supplies two MPW tools on the Developer Services CD and AppleLink called Data and CRC. Data takes the assembled source code file, strips off the CODE 0 resource, and puts the CODE 1 segment (now the actual ROM image) into a data file. This will be convenient for later downloading to a ROM burner. The CRC tool takes the ROM image, calculates the CRC value, and inserts it into the proper place.

The last field in the block is the `ByteLanes` field, a signature byte that identifies which of the four NuBus byte lanes the ROM image appears on. The Slot Manager attempts to read a valid value in each of the four byte lanes at the end of the slot space. If the Slot Manager is unable to read a valid field, then an error is posted for this slot. If a valid `ByteLanes` value is read, this information is used to confirm a special test pattern, and perform a ROM validity check. If all verification passes, then the system can utilize the offset to the `sResource` directory. Note that the Slot Manager attempts to read the format block in both the 1 MB and 16 MB NuBus slot spaces. If any of these verification checks fails, the slot is marked as empty or invalid, and all Slot Manager calls to that slot will return errors.

POTENTIAL PROBLEMS

Now that we've covered the source code to the example ROM, let's look at some common problems developers experience. When trying to assemble the source code, if one or more of the arguments to the `OSLstEntry` or `DatLstEntry` macro is incorrectly defined or just left out, you will get an assembler error in the middle of the macro, and the assembler will complain with the error message:

```
Invalid arithmetic operation on relocatable id
```

This message was generated as a result of the assembler's inability to resolve one of the two arguments to the macro. If you get this error, check both arguments and make sure the labels are correctly defined. The first argument must be equated to something in your source or the development system include files, and the second argument must be a label that exists in the source code. Please be aware that some of the predefined equates (in the assembler include files) changed from MPW 2.0 to 3.0. For instance, to improve readability, some IDs had the underscores in the middle removed (`Cat_Board` became `CatBoard` in a directory, for example).

Another error can arise from a bug in the macro defined up to and including MPW version 3.0. Most declaration ROM sources are arranged in a sequence like ours: the directory comes first in the source code (and so is lowest in memory), followed by the `sResources`, and finally the format block, which is at the very end of the source listing.

Structures referenced by `sResources` are usually defined after the `sResources`. That is, usually things are referenced in a forward manner and come later in the source code.

Laying out the `sResources` this way, the macro works fine. However, if you want to have the macro calculate a negative offset to a structure, to reference something that comes earlier in the source code, you may run into trouble. The following macro definition:

```
MACRO
OSLstEntry       &Id,&Offset
DC.L             (&Id<<24)+&Offset-*
ENDM
```

can be fixed by changing it to:

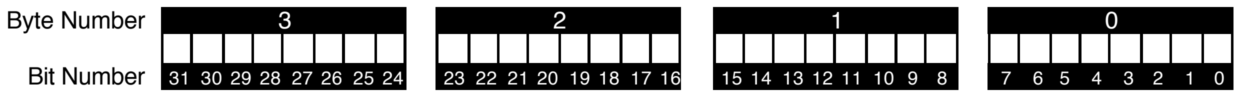
```
MACRO
OSLstEntry      &Id,&Offset
DC.L            (&Id<<24) ++ ((&Offset-*) ** $00FFFFFF)
ENDM
```

This correctly masks off the high byte of the 24-bit offset and thus allows the full range of positive as well as negative offsets to structures.

Often, the source code to the ROM will build, but because of errors in the declaration ROM data structures, the Slot Manager will fail to recognize the ROM, or will generate errors while looking at certain structures. When this happens, looking at the error generated and manually disassembling the ROM will usually find the error. This requires understanding how the ROM appears from a debugger.

DISSEMBLING THE ROM
Declaration ROMs often occupy only one or two of the four NuBus byte lanes, meaning you have to translate your assembly listing by hand. This is because the assembler generates the listing as though the ROM occupies all four byte lanes (that is, as though it would reside in RAM). To translate from the ROM listing to the actual physical addresses the ROM occupies requires knowledge of byte lanes, which are often misunderstood.

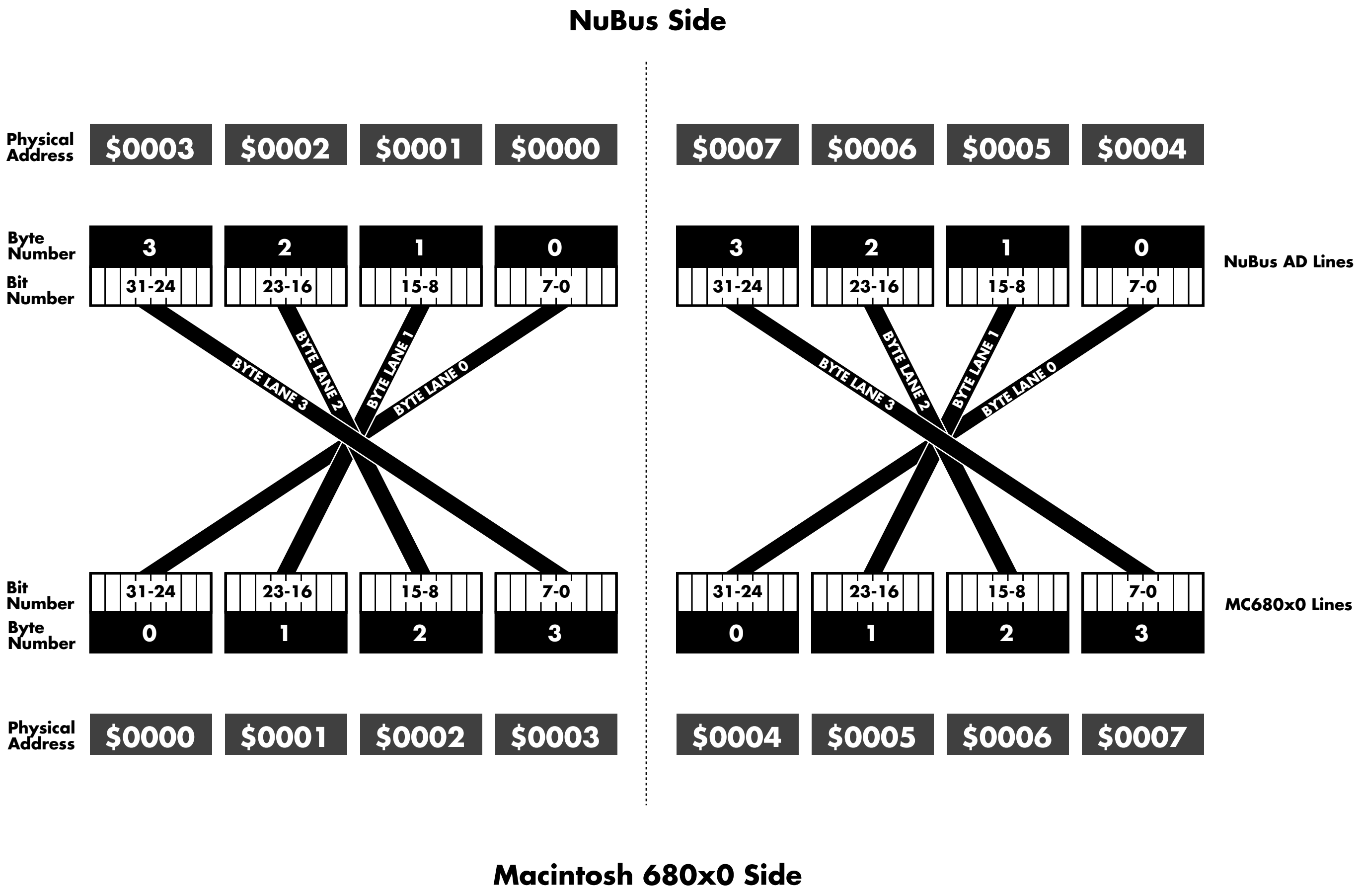
The NuBus bus width is 32 bits, or, very importantly, four bytes. Think of each group of four bytes as a chunk. A chunk on the NuBus would look like this:



The four bytes of each chunk are identified by the byte number as shown in the illustration. Byte number 3 on the NuBus side—the most significant—contains NuBus address and data bits 31-24, byte number 2 contains A/D bits 23-16, byte number 1 contains A/D bits 15-8, and byte number 0—the least significant—contains A/D bits 7-0. Bytes whose address modulo 4 equals 0 are carried on byte number 0, those whose address equals 1 are carried on byte number 1, whose address equals 2 are carried on byte number 2, and whose address equals 3 are carried on byte number 3.

This address-to-byte-number mapping is conveniently set up for an Intel-type processor, which carries the most significant bits on a higher numbered address. The Macintosh uses a Motorola type processor, which has the most significant bits on a lower numbered address. In order to preserve consistency of byte addressing, Apple does byte swapping from the NuBus to the Motorola 680x0 CPU.

To see this more clearly, let’s expand the byte lanes diagram from the address space chapter of *Designing Cards and Drivers for the Macintosh Family.*



The diagram looks quite complicated. Fortunately, once you understand the key concepts, it’s not. The addressing of bytes **within a chunk** is in reverse order on the NuBus and 680x0 sides. However, the address range of a chunk is the same when viewed from the NuBus or 680x0 side. The hardware interface between the CPU on the motherboard swaps the bytes of a chunk when going to and from the NuBus.

Now that we understand the byte and address translation between our CPU and the NuBus, let’s look at part of our assembled ROM listing. The best place to start is at the format block at the “top” or highest physical address of the ROM, since this is where the Slot Manager starts looking at startup time to find valid declaration ROMs. Assembled, the format block looks like:

```
00FEC     ;+++++++++++++++++++++++++++++++++++++++++++++++++
00FEC     ;            Format/Header Block
00FEC     ;+++++++++++++++++++++++++++++++++++++++++++++++++
00FEC     00FF F014    DC.L      (_sRsrcDir-*)**$00FFFFFF
                       ;Offset to sResource directory
00FF0     0000 1000    DC.L      ROMSize
                       ;Length of declaration data
00FF4     0000 0000    DC.L      0
                       ;CRC-can be patched by MPW crc tool
00FF8     01           DC.B      romRevision
                       ;Revision level
00FF9     01           DC.B      AppleFormat
                       ;Format
00FFA     5A93 2BC7    DC.L      TestPattern    ;Test pattern
00FFE     00           DC.B      0
                       ;Reserved byte (must be zero)
00FFF     E1           DC.B      $E1
                       ;ByteLanes: 1110 0001 (byte lane 0)
01000
```

The Slot Manager will start scanning from the highest address, looking for a `ByteLanes` value. From there, it will look for confirmation of the `ByteLanes` value by looking for the reserved values, the test pattern, proper format, and revision values, down to the CRC calculation. If there is a problem with the `ByteLanes` value or the way the card has been built, this Slot Manager check will fail. At this point, you should load up a debugger and look at the format block. Assuming the board is in slot `$B`, the above format block (residing on byte lane 3) might look like this in memory (as seen from MacsBug):

```
  00BFFFB0  0000 0000 FF00 0000  F000 0000 1400 0000  ••••••••••••••••
  00BFFFC0  0000 0000 0000 0000  1000 0000 0000 0000  •••••••• •••••••
  00BFFFD0  9D00 0000 8600 0000  3400 0000 FE00 0000  ••••••••4•••••••
  00BFFFE0  0100 0000 0100 0000  5A00 0000 9300 0000  ••••••••Z•••••••
  00BFFFF0  2B00 0000 C700 0000  0000 0000 E100 0000  +•••••••••••••••
```

The lowest address is at the upper left, and the highest address is at the lower right, with increasing addresses going from left to right. Note that the MacsBug listing shows an example CRC value ($9D8634FE) that was calculated and patched in after the ROM was assembled.

SLOT MANAGER ERRORS

During evaluation at startup or in response to application/driver Slot Manager calls, a number of errors can be returned by the Slot Manager. Often this is due to an incorrect `ByteLanes` value or bad `sResources`. The error returned usually helps to narrow down the problem. You can look at the error, then manually track through `sResources` in the ROM. This requires disassembling and "playing Slot Manager" much as we did above. Drawing a diagram like the one in the front of the article with the addresses and values can often help. However, please note that many Slot Manager calls make other Slot Manager calls, and the error returned may reflect an error returned by one of those calls.

If an error or crash occurs in the ROM before a debugger is loaded (during the primary initialization routine, for example), you can defer the driver or primary initialization until after the boot process has begun and a debugger has been loaded. Do this by first making stubs for the driver and/or primary initialization, or deleting them entirely. Then run them from a high-level application, which can have the primary initialization or drivers in the application or possibly in separate files.

In order to keep the driver from being loaded until after the boot, you might have to temporarily change the functional `sResource`'s type format. This is needed in the case of video boards, for example, since the startup code looks specially for video boards, runs their primary INITs, and opens the drivers. In this case, change the `sRsrc_Type` to something other than `CatDisplay`, `TypVideo`, `DrSwApple` so that the start code won't identify the video display function. Changing `CatDisplay` to `CatNonsense EQU $0000` would do the trick.

SUMMARY

Declaration ROMs store system-recognizable structures as well as vendor-specific data. Debugging declaration ROMs is complicated by the fact that the declaration ROM sits on the other side of the Nubus, and you have to translate the information you get. But using the techniques discussed in this article should make building, interpreting, and debugging declaration ROMS a little easier.

THE APPLE II DEVELOPMENT DYNAMO

The new Dynamo 8-bit Development Package from DTS makes developing software for the Apple II family easier and faster. This article highlights the capabilities of the package and describes how it meets developers' needs. Dynamo includes a run-time module with a macro interface and an Apple II application loader. It offers routines for handling strings, variables, arrays, and integer math.

Because of the speed and memory limitations of 8-bit Apple IIs, Apple's opinion that assembly language is the best choice for development has not changed over the years. Assembly language, however, places a heavy burden on the developer.

The more popular high-level languages depend on a processor being able to handle a large stack as part of its instruction set. Because the 6502 doesn't support a large stack, high-level languages on the Apple II generally implement their stacks in software and pay a considerable penalty in speed and memory.

Given the disadvantage of a processor not specifically designed with high-level languages in mind, the authors of the available Apple II languages have done an admirable job. These languages are very useful for many applications. Programs that require lots of speed and memory, however, can't afford the overhead of a high-level language.

WHAT DO YOU REALLY NEED?

As a developer, you don't need a specific high-level language or a particular tool set. You do need what successful languages and tool sets were designed for: to provide help in meeting fundamental development objectives. Developers need to produce code:

- that is fast
- that takes the least amount of development time

ERIC SOLDAN, Apple II DTS engineer, specializes in Toolboxes, printing, and making trouble. He's been with Apple since 8/8/88, a day he claims the Chinese think is an extremely good day ("eight"pronounced in Chinese means "prosperity"). At the University of Missouri-Rolla, he studied math and computer science, with a minor in beer. When he's not making trouble, he's tending his own enterprise,

- that consumes the least memory
- that is easy to read
- that is reliable

How well does assembly language meet these needs? Assembly produces the fastest code possible. This is especially important on the Apple II, where your code may be running at one megahertz, and every cycle counts.

It is difficult to write good assembly language code quickly. All the housekeeping demands painstaking attention to detail and carefully constructed code.

For code compactness, assembly language is excellent. It is actually difficult to develop assembly code that consumes as much memory as a good compiler.

Assembly language is hard to read. If you need to understand the code a year from now, you had better provide very good comments and plenty of them. Also, you may not be the only one who will need to understand your code.

Assembly is not well known for bug-free code production. You build the ifs, loops and data constructs yourself from assembly language statements. There is no compiler to check your syntax.

After pondering developers' needs, DTS implemented a new environment for Apple II assembly language to help make 8-bit development easier and faster, called Dynamo. The core of Dynamo is a small library of run-time routines. Dynamo has macro definitions that generate very short code fragments that use the library routines. Given that the library is all assembly language, and the macros generate the minimum code necessary to interface to the library, the code stays small and fast. Dynamo handles things that are usually cumbersome in assembly language—namely variable, string, and array management, as well as integer math. The macros add a high-level flavor to the environment, and you get speedy development of readable and dependable code. Let's take a closer look at how the different aspects of Dynamo work.

MANAGING VARIABLES

In developing Dynamo, it was important to look at typical operations within a program. We studied how long it took to code these in assembly, how big the code was, and how long the code took to execute. One such operation is assigning a value to a variable. Some examples (in Pascal) look like this:

```
aardvark := buffalo;
cat := 1000;
dog := elephant * fish / goat + 12345;
```

Educational Software Systems, a business he shares with his wife. Other interests include racquet sports, chess, and piano. •

For this development environment, variables are two-byte integers. There is no support for floating point. For that, you could use SANE or various other solutions. •

If you had a collection of run-time routines to do the real work, the main line of code would just do things like determine what happens to which variables. We looked at several ideas and started coming up with assembly code that looked like the following:

```
    ldx        #aardvark     ;load pointer to aardvark
    ldy        #buffalo      ;load pointer to buffalo
    jsr        varcpy        ;move buffalo to aardvark (16 bits)
```

The x-register is loaded with a constant (from 0 to 254) that represents the variable aardvark. The y-register is loaded with a constant that represents the variable buffalo. These constants are like pointers; they are offsets into a variable table. Then we call a routine to copy the value of the 16-bit variable buffalo to the 16-bit variable aardvark. The routine looks like this:

```
varcpy  lda    varspace,y         ;move low 8 bits
    sta        varspace,x
    lda        varspace+1,y       ;move high 8 bits
    sta        varspace+1,x
    rts
```

Varspace is some location in memory where the integer variables reside. Since we are indexing into it with a 1-byte index, the maximum size of this space is 256 bytes, and since integers take 2 bytes, the maximum number of variables is 128. This 128 variable limit may seem small, but a typical program just doesn’t have that many simple variables.

Now, by using a macro to become part of the interface, this can become:

```
    _varcpy               aardvark,buffalo
```

This means “copy the value of the variable buffalo to the variable aardvark.” It isn't Pascal, but it isn’t trying to be.

Remember that the variable names represent 8-bit numbers. They are declared by equating them to values from 0 to 254. Don’t mistake the variable names for addresses. Saying:

```
temp    equ               30
        lda               #27
        sta               temp
        lda               #00
        sta               temp+1
```

will store the number 27 in zero page location 30, something you don’t want to do. The `_varcpy` routine takes the argument temp as an index into the variable table

Dynamo MPW requires a Macintosh that can run the Macintosh Programmer’s Workshop (MPW) and an 8-bit Apple II or Apple IIGS with a 3.5" drive. You’ll also need MPW, MPW IIGS Tools, MPW IIGS Assembler, and ProDOS with the Dynamo package. •

that starts at varspace. To explicitly store 27 in variable temp without using the Dynamo routines, you would say:

```
temp    equ     30
        lda     #27
        sta     varspace+temp             ;set low byte to 27
        lda     #00
        sta     varspace+temp+1           ;and high byte to zero
```

As long as you use the Dynamo interface to deal with variables, you can treat them as if the name refers to the value.

Now, does this code meet our five objectives?

It isn't as fast as it could be. The fastest code would be:

```
    lda         buffalo
    sta         aardvark
    lda         buffalo+1
    sta         aardvark+1
```

This code, although faster, takes more memory. If we were copying an integer from one place in zero-page to another, the fast code would be 8 bytes. If we were copying an integer not in zero-page, it would be 12 bytes. The slower way would be only 7 bytes. The _varcpy routine takes up some space, but it is in memory only once so it doesn't really count.

Although we lost some speed, the code got smaller. It also became easier to read and faster to write and to debug. A good compromise, given that it is more readable, and is inherently more reliable.

Using these routines and macros, some sample code that assigns some variables, adds, multiplies, and divides looks like this:

```
aardvark          equ                   0   ;declare 16-bit variables
buffalo           equ                   2
cat               equ                   4
dog               equ                   6
elephant          equ                   8
fish              equ                   10
goat              equ                   12
    _varcpy       aardvark,buffalo      ;aardvark := buffalo
    _set          cat,#1000             ;cat := 1000
    _varcpy       dog,elephant          ;dog := elephant
    _mulvar       ,fish                 ;dog := dog*fish
    _divvar       goat                  ;goat := goat/dog
    _add            ,#12345              ;goat := goat+12345
```

Notice that dog doesn't have to be mentioned on the `_mulvar` line. All the Dynamo library routines preserve the x-register, so the x-register still has the constant for dog in it. If there is no destination variable, `_mulvar` generates code that leaves the x-register alone.

MANAGING ARRAYS

We typically store large blocks of data in arrays, which is why the 128 simple variable limit is not so bad. Arrays are a little trickier than variables, and a pointer-based system works best. When you calculate a base pointer to a row, the row elements will be in linear order from there on in memory. So, for a two-dimensional array, tell Dynamo what row to work with, and treat that row as if it were a one-dimensional array. The array routines are good for up to four dimensions. Increasing this is rather easy, but the overhead goes up slightly for each dimension you add.

Here's code for a four-dimensional array:

```
    _array     #$4000,w,#3,#4,#5,#6     ;activate array at $4000
    _index     #1,#3,#4
    _getw      value1,#3  ;value1 := array[1,3,4,3]
    _putw      value2,#5  ;array[1,3,4,5] := value2
```

The `_array` macro defines the size of the array elements, the number of elements in each dimension, and the location of the array. This example has a four-dimensional array whose dimensions are 3 x 4 x 5 x 6. The array starts at address `$4000`, and the element size is a word. `_array` is an assembly-time macro and does its calculating at assembly-time. This means the arguments must be constants like literals or equates. If the first argument had an * instead of a #, it would be used as the address of a pointer to the base address. `_index` is a run-time macro and can use literals or variables.

The `_index` macro indexes into the array, up to but not including the final subscript or index. The `_index` macro is used to calculate a pointer to a row of data. Once this is done, access to the row is as if the array were linear, and you can make multiple accesses to the array. Since the address of the row does not have to be recalculated, the last subscript is simply used as an index into the row of data.

This example uses the `_getw macro` to get a word element of the row and place the value into a variable. Element #3 (from zero) of the row is moved to the variable value1. Finally, the value of the variable `value2` is placed in element #5 of the row.

`_getb` and `_putb` work with byte elements, and `_vgetb`, `_vgetw`, `_vputb`, and _vputw use a variable as an index instead of a constant:

You can also use a variable value when calculating an index to a row of the array—for example, to get an element from the array stuff[] and save it in thing. For you C dudes, this looks like:

```
thing = stuff[color][size][3][weight];
```

Using Dynamo and the macro interface, this is:

```
_array     #stuff,w,#5,#6,#7,#8        ;activate array "stuff"
_vindex    color,size
_index     ,,#3
_vgetw     thing,weight
```

You can mix variable and constant subscripts, as long as you remember the commas as place holders on the second line. Also, if the first index hasn't changed, you don't need to mention it.

There can be only one active array at a time, and _array sets the active array. Instead of putting code in-line to activate an array, you can define simple routines to set active arrays.

```
        jsr       mat1               ;set mat1 active
        _vindex   row                ;set pointer to rowth row
        _vgetw    thing,column       ;thing := mat1[row,col]
        jsr       mat2               ;set mat2 active
        _vindex   row                ;set pointer to rowth row
        _vputw    thing,column       ;mat2[row,col] := thing
        rts
mat1loc           equ                $1000       ;storage for mat1
mat2loc           equ                $1080       ;storage for mat2
mat1    _array    #mat1loc,w,#8,#8
        rts
mat2    _array    #mat2loc,w,#8,#8
        rts
```

The # in #mat1loc and #mat2loc means use the value of mat1loc and mat2loc as base addresses for the arrays. If the #s are replaced with *s, you get another level of indirection, and the contents of mat1loc and mat2loc would be used as the array location instead.

All of these tricks add up to very efficient and flexible array access in assembly language.

MANAGING STRINGS

String management works much like variable management. The x-register holds the constant representing the destination string. You can perform operations on a destination string, like copying another string into it, reading string data into it, appending another string, appending some portion of a string, printing the string, and so forth. Just like the variable management routines, the x-register is always preserved. Of course, all of these functions are done with macros for readability.

PUTTING IT ALL TOGETHER

These simple things make programming in assembly language immensely easier. The only cost is some loss of speed. If a particular routine needs to be as fast as possible, you can still write it in straight assembly code. After all, you are using an assembler.

The Dynamo runtime library is very small. A breakdown of the various routine types is as follows:

initialization & char output routines:	154
integer variables & intmath routines:	787
random generator routines:	139
string handling/output routines:	505
read data (ints & strings) routines:	77
multi-dimension array handling:	358
TOTAL	2020

These values apply only if you use all the routines in a particular area. The linker only includes what you use.

The new MPW IIgs Cross-Development System is another step toward a better development environment. It is the most powerful development environment available for the Apple II and IIgs. Having the most powerful system is important. Developers are the ultimate power users. Developers spend an incredible amount of time using computers. The less time they spend editing code and waiting for assemblies, the more time they have for real development work. MPW lets you keep several windows of source code open at the same time. And since the Mac is not used to test the software, you don't have to boot out of your development environment every time you test your program. You can look at main code and subroutines or data structures while your program is running on your Apple II. Also, the speed of the system reduces development time. There is nothing wrong with developing Apple II software on an Apple II. It just takes longer. If you can afford a Mac and the MPW IIGS Cross-Development System, you should really consider developing with it. It should pay for itself very quickly in terms of development time dollars.

One last statement: Dynamo was not difficult to develop, so if it isn't perfect for your development needs, develop your own macro interface. Just remember, you can keep memory use down and speed up by working in assembly language.

The source code and user's manual for Dynamo are included on develop, the CD and the Apple II source code disks from APDA. •

Apple II Q&A

Q

How can I get back to my program from the Init version of GSBug?

A

The new version of GSBug, available from APDA in a beta version, comes with an Init file which installs the debugger to be present in the background, invocable with the keystrokes control-command-option-Escape. The command to quit the application version of GSBug ("Q") does not work from the Init version; the correct command is "R" (for "resume"). If you break on a tool call, be sure to take tool breaks out before doing any tracing if you hope to not die a different horrible death than the one your application would normally have given you.

Q

I've heard about a version of SANE for eight-bit Apple II computers, but I can't find it anywhere. Is it still sold?

A

SANE is considered part of the System Software and is distributed by Apple's Software Licensing group. You may contact them at (408) 974-4667 or through AppleLink address SW.LICENSE. Although older versions of SANE were sold as part of Apple's old "WorkBench" series, the current version should be obtained from Software Licensing. Even if you own an older copy of SANE you wish to use in your program, it still must be licensed from Software Licensing before distributing it. SANE is built in to all IIgs and Macintosh computers.

Q

What file type should I use for my program's files?

A

Apple II Developer Technical Support assigns file type and auxiliary type combinations to developers by request. Apple II DTS must assign file types and auxiliary types, rather than arbitrate as Macintosh DTS does, since the range of Apple II file types is much more limited. Please refer to "About Apple II File Type Notes" (included on CD or available from APDA) for information on how to submit a request for an assignment, as well as for a complete listing of all currently assigned file type and auxiliary type assignments.

Q

What is "FASTFONT" and how can I use it?

A

FASTFONT is a new disk file for System Software 5.0. It contains a pre-shifted version of the ROM font, Shaston 8. QuickDraw will load FASTFONT from the Fonts directory at QDStartUp time (if present) and use it for greatly increased text drawing speed. Currently, the System does not support different or multiple FastFonts, and no special work is needed by the application to take advantage of the present capability.

Q

What is ExpressLoad and how do I use it?

A

ExpressLoad works with the System Loader to load specially prepared (or "Expressed") files much faster than the System Loader does. Files may be prepared to work with ExpressLoad by using the APW tool "Express", the MPW IIgs tool "ExpressIIgs", or a linker that can automatically create Expressed files. If your file is not Expressed, it will work just fine with System Software 5.0; it just won't load as fast as Expressed files will. Similarly, Expressed files will load properly when ExpressLoad is not present. There are, however, some considerations that should be made when working with Expressed files. These are detailed in the Apple IIgs Technical Note "ExpressLoad Philosophy".

Q

My program uses option-key equivalents for certain functions, and they no longer work under System Software 5.0. How come?

A

(This is the kind of specific question with lots of details that DTS really likes.) System Software 5.0 includes a new key translation feature very similar to that found on the Macintosh. The feature allows special characters to be typed by pressing option-key keystokes. (For example, the *ƒ* character can be generated by typing option-f.) This will interfere with programs that already use option-key equivalents. Your program will not get the keydown event for option-f; you'll get an unmodified keydown event for the ASCII code for "*ƒ*". This feature may be controlled through new Event Manager calls, and may also be deactivated by using the "Translation" option of the "Alphabet" CDev in the Control Panel NDA.

Q

I noticed the Finder is now using application-specific strings for the "kind" of a file. How can I use this capability?

A

The Finder on System Software 5.0 uses a new data structure known as a File Type Descriptor to allow a string to be matched with a particular file type and auxiliary type. Like icon files, multiple File Type Descriptor files may be used, so strings may be "added" to the Finder's vocabulary. Details on the data structure and the Finder's implementation are in Apple II File Type Note for File Type $42.

Q

I'm drawing into a large offscreen bitmap (PixMap), but anything drawn outside the 640 by 480 pixel screen area doesn't get written to the PixMap. Why not?

A

When you create a new port with OpenPort or OpenCPort the visRgn is initialized to the rectangular region defined by screenBits.bounds

(IM I:163). If your port has a large portRect, any drawing will be clipped to the visRgn and you will lose any drawing outside of the screenBits.bounds rectangle.

To correct this set the visRgn of the port to coincide with your port's portRect after creating the port.

Also note that OpenPort initializes the clipRgn to a wide-open rectangular region (-32768, -32768, 32767, 32767). Some operations (i.e.OpenPicture) can fail with this setup, so you should try setting clipRgn to a smaller rectangle.clipRgn to a wide-open rectangular region (-32768, -32768, 32767, 32767). Some operations (i.e.OpenPicture) can fail with this setup, so you should try setting clipRgn to a smaller rectangle.

Q

What is Printing Manager error –8133?

A

Printing Manager error -8133 occurs when the PostScript interpreter of the LaserWriter (or any other PostScript printer) generates a PostScript error. A description of the PostScript command that caused the error will be displayed in the status window. This error often occurs when an application is sending PostScript directly to the printer, and that PostScript contains an error. To debug this kind a problem, you should look at the PostScript generated by the driver. To do this, hold down the Command-F key right after clicking okay in the Print dialog. A file named PostScript0 will be created in the current directory.

Q

I'm confused about the changes to FPRead in AFP version 2.0. How do I use the NewLine mask?

A

The difference between AFP 1.1 and AFP 2.0 as far as the NewLine Mask is concerned is that, in AFP 1.1 the only legal values of Newline Mask are $00 and $FF, whereas in AFP 2.0, all values of Newline Mask are allowed. The Newline Mask is logically ANDed with a copy of each byte read. If the result matches the Newline char, the read terminates. The Newline character is returned as the last byte of data that was read from the fork.

Q

How do I implement file range locking?

A

HFS doesn't provide for file-range locking. AppleShare has additional structures to implement locking, but there is no way for you to implement locking with HFS. We are working on this limitation.

Q

Why does enabling fractional font widths with FractEnable(TRUE) disable Font Substitution?

A

When you call FractEnable(TRUE), you are telling the Printing Manager that you always want to use the fractional width information in the FOND resource of a font. This fractional information is specified for a 1 point font, and can be scaled for any desired size. When you tell the Printing Manager to use this width information, the normal font width information (stored in the ROM of the LaserWriter) is ignored. Font Substitution provides the ability to substitute high quality PostScript fonts for fonts that have no PostScript equivalent. For example, a document laid out in Geneva will be printed with Helvetica if Font Substitution is enabled. When this substitution occurs, the Printing Manager makes adjustments to the intercharacter spacing of the line to make the printed Helvetica version match the width of the Geneva version displayed on the screen. When you enable fractional font widths for a document that uses Geneva, the Printing Manager is being told to print Helvetica characters on the printer using fractional widths from the Geneva screen font. If the Printing Manager placed the Helvetica characters using the fractional widths, formatting problems could occur. For example, some of the Helvetica characters may be wider than their Geneva equivalents, causing character collisions. To avoid this problem, the LaserWriter driver disables the Font Substitution option when fractional fonts ar enabled. This way, WYSIWYG s maintained. If you want to use fractional fonts, then you should format your document using fonts that have PostScript versions available on the LaserWriter.

Q

Inside Mac says that there is a 3K limit on CopyBits. Is this still true?

A

The CopyBits limit is obsolete; there is no longer a 3K limit. The limit depends on the amount of RAM in your system. CopyBits tries to use the stack to do all of the copying. In most cases CopyBits is able to copy entire screen shots at one time. You might run into problems if you don't have enough stack to hold two times the rowBytes of your source, but even in this case CopyBits will attempt to find the memory it needs.

Q

How do I order an MCP card and software? What do I get when I order it?

A

You can order the Macintosh Coprocessor Platform (MCP) card and software through Apple Software Licensing. The card comes with the MCP platform and software which contains the appropriate libraries and header files. On the card is the 68000, ROM (256K), RAM (512K), and the NuBus Logic to drive the card in both master and slave modes. There is also blank space on the board, left there for your communications hardware. Basic documentation is also included.

Q

Can I get a list of all board IDs?

A

No—that information is confidential. MacDTS registers board ID and functional sResource equates so developers don't use equates that are already in use, but they can't distribute the list because the database contains information on unreleased products.

However, even if the list could be distributed, any program that depended on the information in it would be obsolete as soon as a new board came out.

It is recommended you use the Slot Manager's ability to find certain cards or functions. That way, you only need to write your code once, and it will work with newer boards. That's why QuickDraw can find video cards years after it was frozen in ROM. It does so by calling the Slot Manager and looking for boards that perform the QuickDraw compatible video function.

Q

How do you catch a penguin?

A

Before we answer that question, perhaps a little biology is in order. Penguins, by necessity, must be both water and air tight, or they would freeze their little bergies off in the Antarctic seas. Well, that's a problem if you happen to be warm blooded, because there's really no way to sweat (since penguins don't have hands, they were never able to invent Gor-tex®). To handle this problem, they have capillaries in their feet that swell when warm and act as a radiator.

So, to catch a penguin, all you have to do is chase it across the ice. The running will heat the penguin, which in turn, will heat the penguin's feet. Once the penguin gets hot enough, he will stick to the ice (what scientists call "the tongue to the sled effect"). All you have to do then is walk over and pick them up.

PLEASE NOTE:

When picking a penguin up we can't overemphasize how critical it is to use slightly warm water to thaw the feet first. Penguin podiatrists are expensive, and very hard to find.

Q

I'd like to write James Brown in jail. Where do I write to?

A

You can write the Godfather of Soul at:

James Brown, prisoner ID #155413
Broad Rivers Correctional Institute
4460 Broad Rivers Road
Columbia, SC 29210

Brown is serving concurrent six year and six year & 3 month terms for a wild, two-state car chase which happened September 24, 1988. He won't be eligible for parole until 1992.

INDEX